1 General [intro]

1.1 Scope [intro.scope]

1 This International Standard specifies requirements for processors of the C++ programming language. The first such requirement is that they implement the language, and so this International Standard also defines C++. Other requirements and relaxations of the first requirement appear at various places within the Standard.

2 C++ is a general purpose programming language based on the C programming language as described in ISO/IEC 9899:1990 Programming Languages C (1.2). In addition to the facilities provided by C, C++ provides additional data types, classes, templates, exceptions, inline functions, operator overloading, function name overloading, references, free store management operators, function argument checking and type conversion, and additional library facilities. These extensions to C are summarized in C.1. The differences between C++ and ISO C[1] are summarized in C.2. The extensions to C++ since 1985 are summarized in C.1.2.

3 Clauses 17 through 27 (the *library clauses*) describe the Standard C++ library, which provides definitions for the following kinds of entities: macros (16.3), values (3), types (8.1, 8.3), templates (14), classes (9), functions (8.3.5), and objects (7).

4 For classes and class templates, the library clauses specify partial definitions. Private members (11) are not specified, but each implementation shall supply them to complete the definitions according to the description in the library clauses.

5 For functions, function templates, objects, and values, the library clauses specifiy declarations. Implementations shall supply definitions consistent with the descriptions in the library clauses.

6 The names defined in the library have namespace scope (7.3). A C++ translation unit (2.1) obtains access to these names by including the appropriate standard library header (16.2).

7 The templates, classes, functions, and objects in the library have external linkage (3.5). An implementation provides definitions for standard library entities, as necessary, while combining translation units to form a complete C++ program (2.1).

1.2 Normative references [intro.refs]

1 The following standards contain provisions which, through reference in this text, constitute provisions of this International Standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

— ISO/IEC 2382 *Dictionary for Information Processing Systems.*

— ISO/IEC 9899:1990, *C Standard*

— ISO/IEC 9899:1990/DAM 1, *Amendment 1 to C Standard*

2 The library described in Clause 7 of the C Standard and Clause 7 of Amendment 1 to the C standard is hereinafter called the *Standard C Library*.[1]

[1] With the qualifications noted in clauses 17 through 27, and in subclause C.4, the Standard C library is a subset of the Standard C++ library.

1.3 Definitions [intro.defs]

1 For the purposes of this International Standard, the definitions given in ISO/IEC 2382 and the following definitions apply.

— **argument:** An expression in the comma-separated list bounded by the parentheses in a function call expression, a sequence of preprocessing tokens in the comma-separated list bounded by the parentheses in a function-like macro invocation, the operand of `throw`, or an expression in the comma-separated list bounded by the angle brackets in a template instantiation. Also known as an "actual argument" or "actual parameter."

— **diagnostic message:** A message belonging to an implementation-defined subset of the implementation's message output.

— **dynamic type:** The *dynamic type* of an expression is determined by its current value and can change during the execution of a program. If a pointer (8.3.1) whose static type is "pointer to class `B`" is pointing to an object of class `D`, derived from `B` (10), the dynamic type of the pointer is "pointer to `D`." References (8.3.2) are treated similarly.

— **ill-formed program:** input to a C++ processor that is not a well-formed program (*q. v.*).

— **implementation-defined behavior:** Behavior, for a correct program construct and correct data, that depends on the implementation and that each implementation shall document. The range of possible behaviors is delineated by the standard.

— **implementation limits:** Restrictions imposed upon programs by the implementation.

— **locale-specific behavior:** Behavior that depends on local conventions of nationality, culture, and language that each implementation shall document.

— **multibyte character:** A sequence of one or more bytes representing a member of the extended character set of either the source or the execution environment. The extended character set is a superset of the basic character set.

— **parameter:** an object or reference declared as part of a function declaration or definition in the catch clause of an exception handler that acquires a value on entry to the function or handler, an identifier from the comma-separated list bounded by the parentheses immediately following the macro name in a function-like macro definition, or a *template-parameter*. A function can be said to "take arguments" or to "have parameters." Parameters are also known as a "formal arguments" or "formal parameters."

— **signature:** The signature of a function is the information about that function that participates in overload resolution (13.3): the types of its parameters and, if the function is a non-static member of a class, the CV-qualifiers (if any) on the function itself and whether the function is a direct member of its class or inherited from a base class. The signature of a template function specialization includes the types of its template arguments (14.10.4).

— **static type:** The *static type* of an expression is the type (3.9) resulting from analysis of the program without consideration of execution semantics. It depends only on the form of the program and does not change.

— **undefined behavior:** Behavior, such as might arise upon use of an erroneous program construct or of erroneous data, for which the standard imposes no requirements. Permissible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message). Note that many erroneous program constructs do not engender undefined behavior; they are required to be diagnosed.

— **unspecified behavior:** Behavior, for a correct program construct and correct data, that depends on the

[2] Function signatures do not include return type, because that does not participate in overload resolution.

implementation. The implementation is not required to document which behavior occurs. [*Note:* usually, the range of possible behaviors is delineated by the standard. —*end note*]

— **well-formed program:** a C++ program constructed according to the syntax rules, diagnosable semantic rules, and the One Definition Rule (3.1).

2 Clause 17.1 defines additional terms that are used only in the library clauses (17–27).

1.4 Syntax notation [syntax]

1 In the syntax notation used in this International Standard, syntactic categories are indicated by *italic* type, and literal words and characters in `constant width` type. Alternatives are listed on separate lines except in a few cases where a long set of alternatives is presented on one line, marked by the phrase "one of." An optional terminal or nonterminal symbol is indicated by the subscript "*opt*," so

$\{\ expression_{opt}\ \}$

indicates an optional expression enclosed in braces.

2 Names for syntactic categories have generally been chosen according to the following rules:

— *X-name* is a use of an identifier in a context that determines its meaning (e.g. *class-name*, *typedef-name*).

— *X-id* is an identifier with no context-dependent meaning (e.g. *qualified-id*).

— *X-seq* is one or more *X*'s without intervening delimiters (e.g. *declaration-seq* is a sequence of declarations).

— *X-list* is one or more *X*'s separated by intervening commas (e.g. *expression-list* is a sequence of expressions separated by commas).

1.5 The C++ memory model [intro.memory]

1 The fundamental storage unit in the C++ memory model is the *byte*. A byte is at least large enough to contain any member of the basic execution character set and is composed of a contiguous sequence of bits, the number of which is implementation-defined. The least significant bit is called the *low-order* bit; the most significant bit is called the high-order bit. The memory accessible to a C++ program is one or more contiguous sequences of bytes. Every byte has a unique address.[3)]

2 [*Note:* the representation of types is described in 3.9.]

1.6 The C++ object model [intro.object]

1 The constructs in a C++ program create, refer to, access, and manipulate objects. An *object* is a region of storage and, except for bit-fields (9.7), occupies one or more contiguous bytes of storage. An object is created by a *definition* (3.1), by a *new-expression* (5.3.4) or by the implementation (12.2) when needed. The properties of an object are determined when the object is created. An object can have a *name* (3). An object has a *storage duration* (3.7) which influences its *lifetime* (3.8). An object has a type (3.9). The term *object type* refers to the type with which the object is created. The object's type determines the number of bytes that the object occupies and the interpretation of its content. Some objects are *polymorphic* (10.3); the implementation generates information carried in each such object that makes it possible to determine that object's type during program execution. For other objects, the meaning of the values found therein is determined by the type of the *expression*s (5) used to access them.

3) An implementation is free to disregard this requirement as long as doing so has no perceptible effect on the execution of the program. Thus, for example, an implementations is free to place any variable in an internal register that does not have an address as long as the program does not do anything that depends on the address of the variable.

2 Objects can contain other objects, called *sub-objects*. A sub-object can be a *member sub-object* (9.2) or a *base class sub-object* (10). An object that is not a sub-object of any other object is called a *complete object*. For every object `x`, there is some object called *the complete object of* `x`, determined as follows:

— If `x` is a complete object, then `x` is the complete object of `x`.

— Otherwise, the complete object of `x` is the complete object of the (unique) object that contains `x`.

3 C++ provides a variety of built-in types and several ways of composing new types from existing types.

4 Certain types have implementation-defined *alignment* restrictions. An object of one of those types shall appear only at an address that is compatible with its alignment restriction.

1.7 Processor compliance [intro.compliance]

1 Every conforming C++ processor shall, within its resource limits, accept and correctly execute well-formed C++ programs, and shall issue at least one diagnostic message when presented with any ill-formed program that contains a violation of any diagnosable semantic rule or of any syntax rule, except as noted herein.

2 If an ill-formed program contains no diagnosable errors. diagnosable errors, this International Standard places no requirement on processors with respect to that program.

3 The set of "diagnosable semantic rules" consists of all semantic rules in this International Standard except for those rules containing an explicit notation that "no diagnostic is required."

4 Two kinds of implementations are defined: *hosted* and *freestanding*. For a hosted implementation, this standard defines the set of available libraries. 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 libraries that includes certain language-support libraries (17.3.1.3).

5 In this International Standard, the examples, the notes, the footnotes, and the non-normative annexes are not part of the normative Standard. Each example is introduced by "[*Example:*" and terminated by "]". Each note is introduced by "[*Note:*" and terminated by "]".

1.8 Program execution [intro.execution]

1 The semantic descriptions in this International Standard define a parameterized nondeterministic abstract machine. This International Standard places no requirement on the structure of conforming processors. In particular, they need not copy or emulate the structure of the abstract machine. Rather, conforming processors are required to emulate (only) the observable behavior of the abstract machine as explained below.

2 Certain aspects and operations of the abstract machine are described in this International Standard as implementation-defined (for example, `sizeof(int)`). These constitute the parameters of the abstract machine. Each implementation shall include documentation describing its characteristics and behavior in these respects, which documentation defines the instance of the abstract machine that corresponds to that implementation (referred to as the ‘‘corresponding instance’’ below).

3 Certain other aspects and operations of the abstract machine are described in this International Standard as unspecified (for example, order of evaluation of arguments to a function). In each case the Standard defines a set of allowable behaviors. These define the nondeterministic aspects of the abstract machine. An instance of the abstract machine can thus have more than one possible execution sequence for a given program and a given input.

4 Certain other operations are described in this International Standard as undefined (for example, the effect of dereferencing the null pointer).

5 A conforming processor executing a well-formed program shall produce the same observable behavior as one of the possible execution sequences of the corresponding instance of the abstract machine with the same program and the same input. However, if any such execution sequence contains an undefined operation, this International Standard places no requirement on the processor executing that program with that input (not even with regard to operations previous to the first undefined operation).

6 The observable behavior of the abstract machine is its sequence of reads and writes to volatile data and calls to library I/O functions.[4]

7 Accessing an object designated by a volatile lvalue, modifying an object, modifying a file, or calling a function that does any of those operations are all *side effects*, which are changes in the state of the execution environment. Evaluation of an expression might produce side effects. At certain specified points in the execution sequence called sequence points, all side effects of previous evaluations shall be complete and no side effects of subsequent evaluations shall have taken place.[5]

8 Once the execution of a function begins, no expressions from the calling function are evaluated until execution of the called function is completed.[6]

9 In the abstract machine, all expressions are evaluated as specified by the semantics. An actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no needed side effects are produced (including any caused by calling a function or accessing a volatile object).

10 When the processing of the abstract machine is interrupted by receipt of a signal, only the values of objects as of the previous sequence point may be relied on. Objects that may be modified between the previous sequence point and the next sequence point need not have received their correct values yet.

11 An instance of each object with automatic storage duration is associated with each entry into its block. Such an object exists and retains its last-stored value during the execution of the block and while the block is suspended (by a call of a function or receipt of a signal).

12 The least requirements on a conforming implementation are:

— At sequence points, volatile objects are stable in the sense that previous evaluations are complete and subsequent evaluations have not yet occurred.

— At program termination, all data written into files shall be identical to one of the possible results that execution of the program according to the abstract semantics would have produced.

— The input and output dynamics of interactive devices shall take place in such a fashion that prompting messages actually appear prior to a program waiting for input. What constitutes an interactive device is implementation-defined.

— More stringent correspondences between abstract and actual semantics may be defined by each implementation.

13 Define a *full-expression* as an expression that is not a subexpression of another expression.

14 [*Note:* certain contexts in C++ cause the evaluation of a full-expression that results from a syntactic construct other than *expression* (5.18). [*Example:* in 8.5 one syntax for *initializer* is

(*expression-list*)

but the resulting construct is a function call upon a constructor function with *expression-list* as an argument list; such a function call is a full-expression. For another example in 8.5, another syntax for *initializer* is

= *initializer-clause*

but again the resulting construct might be a function call upon a constructor function with one *assignment-expression* as an argument; again, the function call is a full-expression.]]

15 [*Note:* that the evaluation of a full-expression can include the evaluation of subexpressions that are not lexically part of the full-expression. [*Example:* subexpressions involved in evaluating default argument expressions (8.3.6) are considered to be created in the expression that calls the function, not the expression that

[4] An implementation can offer additional library I/O functions as an extension. Implementations that do so should treat calls to those functions as ‘‘observable behavior’’ as well.

[5] Note that some aspects of sequencing in the abstract machine are unspecified; the preceding restriction upon side effects applies to that particular execution sequence in which the actual code is generated.

[6] In other words, function executions do not interleave with each other.

defines the default argument.]]

16 There is a sequence point at the completion of evaluation of each full-expression[7].

17 When calling a function (whether or not the function is inline), there is a sequence point after the evaluation of all function arguments (if any) which takes place before execution of any expressions or statements in the function body. There is also a sequence point after the copying of a returned value and before the execution of any expressions outside the function[8]. Several contexts in C++ cause evaluation of a function call, even though no corresponding function call syntax appears in the translation unit. [*Example:* evaluation of a `new` expression invokes one or more allocation and constructor functions; see 5.3.4. For another example, invocation of a conversion function (12.3.2) can arise in contexts in which no function call syntax appears.] The sequence points at function-entry and function-exit (as described above) are features of the function calls as evaluated, whatever the syntax of the expression that calls the function might be.

18 In the evaluation of each of the expressions

```
a && b
a || b
a ? b : c
a , b
```

using the builtin meaning of the operators in these expressions (5.14, 5.15, 5.16, 5.18) there is a sequence point after the evaluation of the first expression[9].

[7] As specified in 12.2, after the "end-of-full-expression" sequence point, a sequence of zero or more invocations of destructor functions takes place, in reverse order of the construction of each temporary object.
[8] The sequence point at the function return is not explicitly specified in ISO C, and can be considered redundant with sequence points at full-expressions, but the extra clarity is important in C++. In C++, there are more ways in which a called function can terminate its execution, such as the throw of an exception, as discussed below.
[9] The operators indicated in this paragraph are the builtin operators, as described in Clause 5. When one of these operators is overloaded (13) in a valid context, thus designating a user-defined operator function, the expression designates a function invocation, and the operands form an argument list, without an implied sequence point between them.

2 Lexical conventions [lex]

1 A C++ program need not all be translated at the same time. The text of the program is kept in units called *source files* in this standard. A source file together with all the headers (17.3.1.2) and source files included (16.2) via the preprocessing directive `#include`, less any source lines skipped by any of the conditional inclusion (16.1) preprocessing directives, is called a *translation unit*. Previously translated translation units can be preserved individually or in libraries. The separate translation units of a program communicate (3.5) by (for example) calls to functions whose identifiers have external linkage, manipulation of objects whose identifiers have external linkage, or manipulation of data files. Translation units can be separately translated and then later linked to produce an executable program. (3.5).

2.1 Phases of translation [lex.phases]

1 The precedence among the syntax rules of translation is specified by the following phases.[10)]

1 Physical source file characters are mapped to the source character set (introducing new-line characters for end-of-line indicators) if necessary. Trigraph sequences (2.2) are replaced by corresponding single-character internal representations.

2 Each instance of a new-line character and an immediately preceding backslash character is deleted, splicing physical source lines to form logical source lines. A source file that is not empty shall end in a new-line character, which shall not be immediately preceded by a backslash character.

3 The source file is decomposed into preprocessing tokens (2.3) and sequences of white-space characters (including comments). A source file shall not end in a partial preprocessing token or partial comment[11)]. Each comment is replaced by one space character. New-line characters are retained. Whether each nonempty sequence of white-space characters other than new-line is retained or replaced by one space character is implementation-defined. The process of dividing a source file's characters into preprocessing tokens is context-dependent. [*Example:* see the handling of `<` within a `#include` preprocessing directive.]

4 Preprocessing directives are executed and macro invocations are expanded. A `#include` preprocessing directive causes the named header or source file to be processed from phase 1 through phase 4, recursively.

5 Each source character set member and escape sequence in character literals and string literals is converted to a member of the execution character set.

6 Adjacent character string literal tokens are concatenated and adjacent wide string literal tokens are concatenated.

7 White-space characters separating tokens are no longer significant. Each preprocessing token is converted into a token. (See 2.5). The resulting tokens are syntactically and semantically analyzed and translated. The result of this process starting from a single source file is called a *translation unit*.

[10)] Implementations must behave as if these separate phases occur, although in practice different phases might be folded together.

[11)] A partial preprocessing token would arise from a source file ending in one or more characters of a multi-character token followed by a "line-splicing" backslash. A partial comment would arise from a source file ending with an unclosed `/*` comment, or a `//` comment line that ends with a "line-splicing" backslash.

8 The translation units that will form a program are combined. All external object and function references are resolved. Library components are linked to satisfy external references to functions and objects not defined in the current translation. All such translator output is collected into a program image which contains information needed for execution in its execution environment.

2.2 Trigraph sequences [lex.trigraph]

1 Before any other processing takes place, each occurrence of one of the following sequences of three characters ("*trigraph sequences*") is replaced by the single character indicated in Table 1.

Table 1—trigraph sequences

trigraph	*replacement*	*trigraph*	*replacement*	*trigraph*	*replacement*
`??=`	`#`	`??(`	`[`	`??<`	`{`
`??/`	`\`	`??)`	`]`	`??>`	`}`
`??'`	`^`	`??!`	`|`	`??-`	`~`

2 [*Example:*

```
??=define arraycheck(a,b) a??(b??) ??!??! b??(a??)
```

becomes

```
#define arraycheck(a,b) a[b] || b[a]
```

—*end example*]

2.3 Preprocessing tokens [lex.pptoken]

preprocessing-token:
header-name
identifier
pp-number
character-literal
string-literal
preprocessing-op-or-punc
each non-white-space character that cannot be one of the above

1 Each preprocessing token that is converted to a token (2.5) shall have the lexical form of a keyword, an identifier, a literal, an operator, or a punctuator.

2 A *preprocessing token* is the minimal lexical element of the language in translation phases 3 through 6. The categories of preprocessing token are: *header names*, *identifiers*, *preprocessing numbers*, *character literals*, *string literals*, *preprocessing-op-or-punc*, and single non-white-space characters that do not lexically match the other preprocessing token categories. If a `'` or a `"` character matches the last category, the behavior is undefined. Preprocessing tokens can be separated by *white space*; this consists of comments (2.6), or *white-space characters* (space, horizontal tab, new-line, vertical tab, and form-feed), or both. As described in Clause 16, in certain circumstances during translation phase 4, white space (or the absence thereof) serves as more than preprocessing token separation. White space can appear within a preprocessing token only as part of a header name or between the quotation characters in a character literal or string literal.

3 If the input stream has been lexically analyzed into preprocessing tokens up to a given character, the next preprocessing token is the longest sequence of characters that could constitute a preprocessing token, even if that would cause further lexical analysis to fail.

4 [*Example:* The program fragment `1Ex` is parsed as a preprocessing number token (one that is not a valid floating or integer literal token), even though a parse as the pair of preprocessing tokens `1` and `Ex` might produce a valid expression (for example, if `Ex` were a macro defined as `+1`). Similarly, the program fragment `1E1` is parsed as a preprocessing number (one that is a valid floating literal token), whether or not `E` is a macro name.]

5 [*Example:* The program fragment `x+++++y` is parsed as `x ++ ++ + y`, which, if `x` and `y` are of built-in types, violates a constraint on increment operators, even though the parse `x ++ + ++ y` might yield a correct expression.]

2.4 Alternative tokens [lex.digraph]

1 Alternative token representations are provided for some operators and punctuators[12].

2 In all respects of the language, each alternative token behaves the same, respectively, as its primary token, except for its spelling[13]. The set of alternative tokens is defined in Table 2.

Table 2—alternative tokens

alternative	*primary*	*alternative*	*primary*	*alternative*	*primary*
`<%`	`{`	`and`	`&&`	`and_eq`	`&=`
`%>`	`}`	`bitor`	`|`	`or_eq`	`|=`
`<:`	`[`	`or`	`||`	`xor_eq`	`^=`
`:>`	`]`	`xor`	`^`	`not`	`!`
`%:`	`#`	`compl`	`~`	`not_eq`	`!=`
`%:%:`	`##`	`bitand`	`&`		

2.5 Tokens [lex.token]

token:
- *identifier*
- *keyword*
- *literal*
- *operator*
- *punctuator*

1 There are five kinds of tokens: identifiers, keywords, literals,[14] operators, and other separators. Blanks, horizontal and vertical tabs, newlines, formfeeds, and comments (collectively, "white space"), as described below, are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and literals.

2.6 Comments [lex.comment]

1 The characters `/*` start a comment, which terminates with the characters `*/`. These comments do not nest. The characters `//` start a comment, which terminates with the next new-line character. If there is a formfeed or a vertical-tab character in such a comment, only white-space characters shall appear between it and the new-line that terminates the comment; no diagnostic is required. The comment characters `//`, `/*`, and `*/` have no special meaning within a `//` comment and are treated just like other characters. Similarly, the comment characters `//` and `/*` have no special meaning within a `/*` comment.

[12] These include "digraphs" and additional reserved words. The term "digraph" (token consisting of two characters) is not perfectly descriptive, since one of the alternative preprocessing-tokens is `%:%:` and of course several primary tokens contain two characters. Nonetheless, those alternative tokens that aren't lexical keywords are colloquially known as "digraphs".
[13] Thus `[` and `<:` behave differently when "stringized" (16.3.2), but can otherwise be freely interchanged.
[14] Literals include strings and character and numeric literals.

2.7 Identifiers [lex.name]

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
```

1 An identifier is an arbitrarily long sequence of letters and digits. The first character is a letter; the underscore _ counts as a letter. Upper- and lower-case letters are different. All characters are significant.

2.8 Keywords [lex.key]

1 The identifiers shown in Table 3 are reserved for use as keywords, and shall not be used otherwise in phases 7 and 8:

Table 3—keywords

asm	do	inline	short	typeid
auto	double	int	signed	typename
bool	dynamic_cast	long	sizeof	union
break	else	mutable	static	unsigned
case	enum	namespace	static_cast	using
catch	explicit	new	struct	virtual
char	extern	operator	switch	void
class	false	private	template	volatile
const	float	protected	this	wchar_t
const_cast	for	public	throw	while
continue	friend	register	true	
default	goto	reinterpret_cast	try	
delete	if	return	typedef	

2 Furthermore, the alternative representations shown in Table 4 for certain operators and punctuators (2.4) are reserved and shall not be used otherwise:

Table 4—alternative representations

bitand	and	bitor	or	xor	compl
and_eq	or_eq	xor_eq	not	not_eq	

3 In addition, identifiers containing a double underscore (_ _) or beginning with an underscore and an upper-case letter are reserved for use by C++ implementations and standard libraries and shall be avoided by users; no diagnostic is required.

4 The lexical representation of C++ programs includes a number of preprocessing tokens which are used in the syntax of the preprocessor or are converted into tokens for operators and punctuators:

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
```

After preprocessing, each *preprocessing-op-or-punc* is converted to a single token in translation phase 7 (2.1).

5 [*Note:* Certain implementation-defined properties, such as the type of a `sizeof` (5.3.3) expression, the ranges of fundamental types (3.9.1), and the types of the most basic library functions are defined in the standard headers `<limits>`, `<cstddef>`, and `<new>` (_lib.support_). —*end note*]

2.9 Literals [lex.literal]

1 There are several kinds of literals.[15)]

literal:
  integer-literal
  character-literal
  floating-literal
  string-literal
  boolean-literal

2.9.1 Integer literals [lex.icon]

integer-literal:
  decimal-literal integer-suffix$_{opt}$
  octal-literal integer-suffix$_{opt}$
  hexadecimal-literal integer-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
```

15) The term “literal” generally designates, in this International Standard, those tokens that are called “constants” in ISO C.

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
```

1 An integer literal consisting of a sequence of digits is taken to be decimal (base ten) unless it begins with `0` (digit zero). A sequence of octal digits[16)] starting with `0` is taken to be an octal integer (base eight). A sequence of digits preceded by `0x` or `0X` is taken to be a hexadecimal integer (base sixteen). The hexadecimal digits include `a` or `A` through `f` or `F` with decimal values ten through fifteen. [*Example:* the number twelve can be written `12`, `014`, or `0XC`.]

2 The type of an integer literal depends on its form, value, and suffix. If it is decimal and has no suffix, it has the first of these types in which its value can be represented: `int`, `long int`, `unsigned long int`. If it is octal or hexadecimal and has no suffix, it has the first of these types in which its value can be represented: `int`, `unsigned int`, `long int`, `unsigned long int`. If it is suffixed by `u` or `U`, its type is the first of these types in which its value can be represented: `unsigned int`, `unsigned long int`. If it is suffixed by `l` or `L`, its type is the first of these types in which its value can be represented: `long int`, `unsigned long int`. If it is suffixed by `ul`, `lu`, `uL`, `Lu`, `Ul`, `lU`, `UL`, or `LU`, its type is `unsigned long int`.

3 A program is ill-formed if it contains an integer literal that cannot be represented by any of the allowed types.

2.9.2 Character literals [lex.ccon]

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

16) The digits `8` and `9` are not octal digits.

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

1 A character literal is one or more characters enclosed in single quotes, as in `'x'`, optionally preceded by the letter `L`, as in `L'x'`. Single character literals that do not begin with `L` have type `char`, with value equal to the numerical value of the character in the machine's character set. Multicharacter literals that do not begin with `L` have type `int` and implementation-defined value.

2 A character literal that begins with the letter `L`, such as `L'ab'`, is a wide-character literal. Wide-character literals have type `wchar_t`.[17] Wide-character literals have implementation-defined values, regardless of the number of characters in the literal.

3 Certain nongraphic characters, the single quote `'`, the double quote `"`, `?`, and the backslash `\`, can be represented according to Table 5.

Table 5—escape sequences

new-line	NL (LF)	\n
horizontal tab	HT	\t
vertical tab	VT	\v
backspace	BS	\b
carriage return	CR	\r
form feed	FF	\f
alert	BEL	\a
backslash	\	\\
question mark	?	\?
single quote	’	\’
double quote	"	\"
octal number	*ooo*	\ooo
hex number	*hhh*	\x*hhh*

If the character following a backslash is not one of those specified, the behavior is undefined. An escape sequence specifies a single character.

4 The escape `\`*ooo* consists of the backslash followed by one or more octal digits that are taken to specify the value of the desired character. The escape `\x`*hhh* consists of the backslash followed by `x` followed by one or more hexadecimal digits that are taken to specify the value of the desired character. There is no limit to the number of digits in either sequence. A sequence of octal or hexadecimal digits is terminated by the first character that is not an octal digit or a hexadecimal digit, respectively. The value of a character literal is implementation-defined if it exceeds that of the largest `char` (for ordinary literals) or `wchar_t` (for wide literals).

[17] They are intended for character sets where a character does not fit into a single byte.

2.9.3 Floating literals [lex.fcon]

floating-literal:
  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`

1 A floating literal consists of an integer part, a decimal point, a fraction part, an `e` or `E`, an optionally signed integer exponent, and an optional type suffix. The integer and fraction parts both consist of a sequence of decimal (base ten) digits. Either the integer part or the fraction part (not both) can be missing; either the decimal point or the letter `e` (or `E`) and the exponent (not both) can be missing. The type of a floating literal is `double` unless explicitly specified by a suffix. The suffixes `f` and `F` specify `float`, the suffixes `l` and `L` specify `long double`.

2.9.4 String literals [lex.string]

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

1 A string literal is a sequence of characters (as defined in 2.9.2) surrounded by double quotes, optionally beginning with the letter `L`, as in `"..."` or `L"..."`. A string literal that does not begin with `L` has type "array of *n* `char`" and *static* storage duration (3.7), where *n* is the size of the string as defined below, and is initialized with the given characters. Whether all string literals are distinct (that is, are stored in nonoverlapping objects) is implementation-defined. The effect of attempting to modify a string literal is undefined.

2 A string literal that begins with `L`, such as `L"asdf"`, is a wide string literal. A wide string literal has type "array of *n* `wchar_t`," where *n* is the size of the string as defined below.

3 Adjacent string literals are concatenated. Adjacent wide string literals are concatenated. If a string literal token is adjacent to a wide string literal token, the behavior is undefined. Characters in concatenated strings are kept distinct. [*Example:*

```
"\xA" "B"
```

contains the two characters `'\xA'` and `'B'` after concatenation (and not the single hexadecimal character `'\xAB'`).]

4 After any necessary concatenation `'\0'` is appended so that programs that scan a string can find its end. The size of a string is the number of its characters including this terminator. Within a string, the double quote character `"` shall be preceded by a `\`.

5 Escape sequences in string literals have the same meaning as in character literals (2.9.2).

2.9.5 Boolean literals [lex.bool]

boolean-literal:
    `false`
    `true`

1 The Boolean literals are the keywords `false` and `true`. Such literals have type `bool` and the given values. They are not lvalues.

3 Basic concepts [basic]

1 [*Note:* this clause presents the basic concepts of the C++ language. It explains the difference between an *object* and a *name* and how they relate to the notion of an *lvalue*. It introduces the concepts of a *declaration* and a *definition* and presents C++'s notion of *type*, *scope*, *linkage*, and *storage duration*. The mechanisms for starting and terminating a program are discussed. Finally, this clause presents the fundamental types of the language and lists the ways of constructing *compound* types from these.

2 This clause does not cover concepts that affect only a single part of the language. Such concepts are discussed in the relevant clauses. —*end note*]

3 An *entity* is a value, object, subobject, base class subobject, array element, variable, function, set of functions, instance of a function, enumerator, type, class member, template, or namespace.

4 A *name* is a use of an identifier (2.7) that denotes an entity or *label* (6.6.4, 6.1).

5 Every name that denotes an entity is introduced by a *declaration*. Every name that denotes a label is introduced either by a `goto` statement (6.6.4) or a *labeled-statement* (6.1). Every name is introduced in some contiguous portion of program text called a *declarative region* (3.3), which is the largest part of the program in which that name can possibly be valid. In general, each particular name is valid only within some possibly discontiguous portion of program text called its *scope* (3.3). To determine the scope of a declaration, it is sometimes convenient to refer to the *potential scope* of a declaration. The scope of a declaration is the same as its potential scope unless the potential scope contains another declaration of the same name. In that case, the potential scope of the declaration in the inner (contained) declarative region is excluded from the scope of the declaration in the outer (containing) declarative region.

6 [*Example:* in

```
int j = 24;

int main()
{
        int i = j, j;

        j = 42;
}
```

the identifier `j` is declared twice as a name (and used twice). The declarative region of the first `j` includes the entire example. The potential scope of the first `j` begins immediately after that `j` and extends to the end of the program, but its (actual) scope excludes the text between the `,` and the `}`. The declarative region of the second declaration of `j` (the `j` immediately before the semicolon) includes all the text between `{` and `}`, but its potential scope excludes the declaration of `i`. The scope of the second declaration of `j` is the same as its potential scope.]

7 Some names denote types, classes, enumerations, or templates. In general, it is necessary to determine whether or not a name denotes one of these entities before parsing the program that contains it. The process that determines this is called *name lookup* (3.4).

8 Two names denote the same entity if

— they are identifiers composed of the same character sequence; or

— they are the names of overloaded operator functions formed with the same operator; or

— they are the names of user-defined conversion functions formed with the same type.

9 An identifier used in more than one translation unit can potentially refer to the same entity in these translation units depending on the linkage (3.5) specified in the translation units.

3.1 Declarations and definitions [basic.def]

1 A declaration (7) introduces one or more names into a program and gives each name a meaning.

2 A declaration is a *definition* unless it declares a function without specifying the function's body (8.4), it contains the `extern` specifier (7.1.1) and neither an *initializer* nor a *function-body*, it declares a static data member in a class declaration (9.5), it is a class name declaration (9.1), or it is a `typedef` declaration (7.1.3), a `using` declaration(7.3.3), or a `using` directive(7.3.4).

3 [*Example:* all but one of the following are definitions:

```
int a;                           // defines a
extern const int c = 1;          // defines c
int f(int x) { return x+a; }     // defines f
struct S { int a; int b; };      // defines S
struct X {                       // defines X
    int x;                       // defines nonstatic data member x
    static int y;                // declares static data member y
    X(): x(0) { }                // defines a constructor of X
};
int X::y = 1;                    // defines X::y
enum { up, down };               // defines up and down
namespace N { int d; }           // defines N and N::d
namespace N1 = N;                // defines N1
X anX;                           // defines anX
```

whereas these are just declarations:

```
extern int a;                    // declares a
extern const int c;              // declares c
int f(int);                      // declares f
struct S;                        // declares S
typedef int Int;                 // declares Int
extern X anotherX;               // declares anotherX
using N::d;                      // declares N::d
```

—*end example*]

4 [*Note:* in some circumstances, C++ implementations implicitly define the default constructor (12.1), copy constructor (12.8), assignment operator (12.8), or destructor (12.4) member functions. [*Example:* given

```
struct C {
    string s;    // string is the standard library class (21.1.2)
};

int main()
{
    C a;
    C b = a;
    b = a;
}
```

the implementation will implicitly define functions to make the definition of `C` equivalent to

```
struct C {
    string s;
    C(): s() { }
    C(const C& x): s(x.s) { }
    C& operator=(const C& x) { s = x.s; return *this; }
    ~C() { }
};
```

—*end example*] —*end note*]

5 [*Note:* a class name can also be implicitly declared by an *elaborated-type-specifier* (7.1.5.3).]

3.2 One definition rule [basic.def.odr]

1 No translation unit shall contain more than one definition of any variable, function, class type, enumeration type or template.

2 A function is *used* if it is called, its address is taken, or it is a virtual member function that is not pure (10.4). Every program shall contain at least one definition of every function that is used in that program. That definition can appear explicitly in the program, it can be found in the standard or a user-defined library, or (when appropriate) it is implicitly defined (see 12.1, 12.4 and 12.8). If a non-virtual function is not defined, a diagnostic is required only if an attempt is actually made to call that function. If a virtual function is neither called nor defined, no diagnostic is required.

3 A non-local variable with static storage duration shall have exactly one definition in a program unless the variable has a builtin type or is an aggregate and also is unused or used only as the operand of the `sizeof` operator.

4 At least one definition of a class is required in a translation unit if the class is used other than in the formation of a pointer or reference type.

5 [*Example:* the following complete translation unit is well-formed, even though it never defines `X`:

```
struct X;       // declare X is a struct type
struct X* x1;   // use X in pointer formation
X* x2;          // use X in pointer formation
```

—*end example*]

6 There can be more than one definition of a named enumeration type in a program provided that each definition appears in a different translation unit and the names and values of the enumerators are the same.

7 There can be more than one definition of a class type in a program provided that each definition appears in a different translation unit and the definitions describe the same type.

8 No diagnostic is required for a violation of the ODR rule.

3.3 Declarative regions and scopes [basic.scope]

1 The name look up rules are summarized in 3.4.

3.3.1 Local scope [basic.scope.local]

1 A name declared in a block (6.3) is local to that block. Its scope begins at its point of declaration (3.3.9) and ends at the end of its declarative region.

2 A function parameter name in a function definition (8.4) is a local name in the scope of the outermost block of the function and shall not be redeclared in that scope.

3 The name in a `catch` exception-declaration is local to the handler and shall not be redeclared in the outermost block of the handler.

4 Names declared in the *for-init-statement*, *condition*, and controlling expression parts of `if`, `while`, `for`, and `switch` statments are local to the `if`, `while`, `for`, or `switch` statement (including the controlled statement), and shall not be redeclared in a subsequent condition or controlling expression of that statement nor in the outermost block of the controlled statement.

5 Names declared in the outermost block of the controlled statement of a `do` statement shall not be redeclared in the controlling expression.

3.3.2 Function prototype scope [basic.scope.proto]

1 In a function declaration, or in any of function declarator except the declarator of a function definition (8.4), names of parameters (if supplied) have function prototype scope, which terminates at the end of the function declarator.

3.3.3 Function scope

1 Labels (6.1) can be used anywhere in the function in which they are declared. Only labels have function scope.

3.3.4 Namespace scope [basic.scope.namespace]

1 A name declared in a named or unnamed namespace (7.3) has namespace scope. Its potential scope includes its namespace from the name's point of declaration (3.3.9) onwards, as well as the potential scope of any *using directive* (7.3.4) that nominates its namespace. A namespace member can also be used after the `::` scope resolution operator (5.1) applied to the name of its namespace.

2 A name declared outside all named or unnamed namespaces (7.3), blocks (6.3) and classes (9) has *global namespace scope* (also called *global scope*). The potential scope of such a name begins at its point of declaration (3.3.9) and ends at the end of the translation unit that is its declarative region. Names declared in the global namespace scope are said to be *global*.

3.3.5 Class scope [basic.scope.class]

1 The name of a class member is local to its class and can be used only in:

— the scope of that class (9.3) or a class derived (10) from that class,

— after the `.` operator applied to an expression of the type of its class (5.2.4) or a class derived from its class,

— after the `->` operator applied to a pointer to an object of its class (5.2.4) or a class derived from its class,

— after the `::` scope resolution operator (5.1) applied to the name of its class or a class derived from its class,

— or after a *using declaration* (7.3.3).

2 [*Note:* The scope of names introduced by friend declarations is described in 7.3.1. The scope rules for classes are summarized in 9.3.]

3.3.6 Name hiding [basic.scope.hiding]

1 A name can be hidden by an explicit declaration of that same name in a nested declarative region or derived class.

2 A class name (9.1) or enumeration name (7.2) can be hidden by the name of an object, function, or enumerator declared in the same scope. If a class or enumeration name and an object, function, or enumerator are declared in the same scope (in any order) with the same name, the class or enumeration name is hidden wherever the object, function, or enumerator name is visible.

3 In a member function definition, the declaration of a local name hides the declaration of a member of the class with the same name; see 9.3. The declaration of a member in a derived class (10) hides the declaration of a member of a base class of the same name; see 10.2.

4 If a name is in scope and is not hidden it is said to be *visible*.

3.3.7 Explicit qualification [basic.scope.exqual]

1 [*Note:* a name hidden by a nested declarative region or derived class can still be used when it is qualified by its class or namespace name using the `::` operator (5.1, 9.5, 10). A hidden global scope name can still be used when it is qualified by the unary `::` operator (5.1). —*end note*]

3.3.8 Elaborated type specifier [basic.scope.elab]

1 A class name or enumeration name can be hidden by the name of an object, function, or enumerator in local, class or namespace scope. A hidden class name can still be used when appropriately prefixed with `class`, `struct`, or `union` (7.1.5), or when followed by the `::` operator. A hidden enumeration name can still be used when appropriately prefixed with `enum` (7.1.5). [*Example:*

```
class A {
public:
    static int n;
};

int main()
{
    int A;

    A::n = 42;          // OK
    class A a;          // OK
    A b;                // ill-formed: A does not name a type
}
```

—*end example*]

2 [*Note:* the scope of class names first introduced in *elaborated-type-specifiers* is described in (7.1.5.3).]

3.3.9 Point of declaration [basic.scope.pdecl]

1 The *point of declaration* for a name is immediately after its complete declarator (8) and before its *initializer* (if any), except as noted below. [*Example:*

```
int x = 12;
{ int x = x; }
```

Here the second `x` is initialized with its own (unspecified) value.]

2 A nonlocal name remains visible up to the point of declaration of the local name that hides it. [*Example:*

```
const int  i = 2;
{ int  i[i]; }
```

declares a local array of two integers.]

3 [*Note:* for the point of declaration for an enumerator, see 7.2. For the point of declaration of a function first declared in a `friend` declaration, see 11.4. For the point of declaration of a class first declared in an *elaborated-type-specifier* or in a `friend` declaration, see 7.1.5.3. For point of instantiation of a template, see 14.3.]

4

3.4 Name look up [class.scope]

1 The name look up rules apply uniformly to all names (including *typedef-name*s (7.1.3), *namespace-name*s (7.3) and *class-name*s (9.1)) wherever the grammar allows such names in the context discussed by a particular rule. This section discusses name look up in lexical scope only; 3.5 discusses linkage issues. The notions of name hiding and point of declaration are discussed in 3.3.

2 Name look up associates the use of a name with a visible declaration (3.1) of that name. Name look up shall find an unambiguous declaration for the name (see 10.2). Name look up may associate more than one declaration with a name if it finds the name to be a function name; in this case, all the declarations shall be found in the same scope (10.2); the declarations are said to form a set of overloaded functions (13.1). Overload resolution (13.3) takes place after name look up has succeeded. The access rules (11) are considered only once name look up and function overload resolution (if applicable) have succeeded. Only after name look up, function overload resolution (if applicable) and access checking have succeeded are the attributes introduced by the name's declaration used further in expression processing (5).

3 A name used in the global scope outside of any function, class or user-declared namespace, shall be declared before it is used in global scope or be a name introduced by a `using` directive (7.3.4) that appears in global scope before the name is used.

4 A name specified after a *nested-name-specifier* is looked up in the scope of the class or namespace denoted by the *nested-name-specifier*; see 5.1 and 7.3.1.1. A name prefixed by the unary scope operator `::` (5.1) is looked up in global scope. A name specified after the `.` operator or `->` operator of a class member access is looked up as specified in 5.2.4.

5 A name that is not qualified in any of the ways described above and that is used in a namespace outside of the definition of any function or class shall be declared before its use in that namespace or in one of its enclosing namespaces or, be introduced by a `using` directive (7.3.4) visible at the point the name is used.

6 A name that is not qualified in any of the ways described above and that is used in a function that is not a class member shall be declared before its use in the block in which it is used or in one of its enclosing blocks (6.3) or, shall be declared before its use in the namespace enclosing the function definition or in one of its enclosing namespaces or, shall be introduced by a `using` directive (7.3.4) visible at the point the name is used.

7 A name that is not qualified in any of the ways described above and that is used in the definition of a class `X` outside of any inline member function or nested class definition shall be declared before its use in class `X` (9.3) or be a member of a base class of class `X` (10) or, if `X` is a nested class of class `Y` (9.8), shall be declared before the definition of class `X` in the enclosing class `Y` or in `Y`'s enclosing classes or, if `X` is a local class (9.9), shall be declared before the definition of class `X` in a block enclosing the definition of class `X` or, shall be declared before the definition of class `X` in a namespace enclosing the definition of class `X` or, be introduced by a `using` directive (7.3.4) visible at the point the name is used. [*Note:* Subclause 9.3 further describes the restrictions on the use of names in a class definition. Subclause 9.8 further describes the restrictions on the use of names in nested class definitions. Subclause 9.9 further describes the restrictions on the use of names in local class definitions.]

8 A name that is not qualified in any of the ways described above and that is used in a function that is a member function (9.4) of class `X` shall be declared before its use in the block in which it is used or in an enclosing block (6.3) or, shall be a member of class `X` (9.2) or a member of a base class of class `X` (10) or, if `X` is a nested class of class `Y` (9.8), shall be a member of the enclosing class `Y` or a member of `Y`'s enclosing classes or, if `X` is a local class (9.9), shall be declared before the definition of class `X` in a block enclosing the definition of class `X` or, shall be declared before the member function definition in a namespace enclosing the member function definition or, be introduced by a `using` directive (7.3.4) visible at the point the name is used. [*Note:* Subclause 9.4 and 9.5 further describe the restrictions on the use of names in member function definitions. Subclause 9.8 further describes the restrictions on the use of names in the scope of nested classes. Subclause 9.9 further describes the restrictions on the use of names in local class definitions.]

9 For a `friend` function (11.4) defined inline in the definition of the class granting friendship, name look up in the `friend` function definition for a name that is not qualified in any of the ways described above proceeds as described in member function definitions. If the `friend` function is not defined in the class granting friendship, name look up in the `friend` function definition for a name that is not qualified in any of the ways described above proceeds as described in nonmember function definitions.

10 A name that is not qualified in any of the ways described above and that is used in a function *parameter-declaration-clause* as a default argument (8.3.6) or that is used in a function *ctor-initializer* (12.6.2) is looked up as if the name was used in the outermost block of the function definition. In particular, the function parameter names are visible for name look up in default arguments and in *ctor-initializer*s. [*Note:* Subclause 8.3.6 further describes the restrictions on the use of names in default arguments. Subclause 12.6.2 further describes the restrictions on the use of names in a *ctor-initializer*.]

11 A name that is not qualified in any of the ways described above and that is used in the *initializer* expression of a `static` member of class `X` (9.5.2) shall be a member of class `X` (9.2) or a member of a base class of class `X` (10) or, if `X` is a nested class of class `Y` (9.8), shall be a member of the enclosing class `Y` or a member of `Y`'s enclosing classes or, be declared before the static member definition in the namespace enclosing the static member definition or in one of its enclosing namespaces or, be introduced by a `using` directive (7.3.4) visible at the point the name is used. [*Note:* Subclause 9.5.2 further describes the restrictions on the use of names in the *initializer* expression for a `static` data member. Subclause 9.8 further describes the restrictions on the use of names in nested class definitions.]

12 In all cases, the scopes are searched for a declaration in the order listed in each of the respective category above and name look up ends as soon as a declaration is found for the name.

3.5 Program and linkage [basic.link]

1 A *program* consists of one or more *translation units* (2) linked together. A translation unit consists of a sequence of declarations.

translation unit:
  declaration-seq$_{opt}$

2 A name is said to have *linkage* when it might denote the same object, function, type, template, or value as a name introduced by a declaration in another scope:

— When a name has *external linkage*, the entity it denotes can be referred to by names from scopes of other translation units or from other scopes of the same translation unit.

— When a name has *internal linkage*, the entity it denotes can be referred to by names from other scopes of the same translation unit.

— When a name has *no linkage*, the entity it denotes cannot be referred to by names from other scopes.

3 A name of namespace scope (3.3.4) has internal linkage if it is the name of

— a variable that is explicitly declared `static` or, is explicitly declared `const` and neither explicitly declared `extern` nor previously declared to have external linkage; or

— a function that is explicitly declared `static` or, is explicitly declared `inline` and neither explicitly declared `extern` nor previously declared to have external linkage; or

— the name of a data member of an anonymous union.

4 A name of namespace scope has external linkage if it is the name of

— a variable, unless it has internal linkage; or

— a function, unless it has internal linkage; or

— a class (9) or enumeration (7.2) or an enumerator; or

— a template (14).

5 In addition, a name of class scope has external linkage if the name of the class has external linkage.

6 The name of a function declared in a block scope or a variable declared `extern` in a block scope has linkage, either internal or external to match the linkage of prior visible declarations of the name in the same translation unit, but if there is no prior visible declaration it has external linkage.

7 Names not covered by these rules have no linkage. Moreover, except as noted, a name declared in a local scope (3.3.1) has no linkage. A name with no linkage (notably, the name of a class or enumeration declared in a local scope (3.3.1)) shall not be used to declare an entity with linkage. [*Example:*

```
        void f()
        {
            struct A { int x; };       // no linkage
            extern A a;                // ill-formed
        }
```

—*end example*] This implies that names with no linkage cannot be used as template arguments (14.8).

8 Two names that are the same and that are declared in different scopes shall denote the same object, function, type, enumerator, or template if

— both names have external linkage or else both names have internal linkage and are declared in the same translation unit; and

— both names refer to members of the same namespace or to members, not by inheritance, of the same class; and

— when both names denote functions or function templates, the function types are identical for purposes of overloading.

9 After all adjustments of types (during which typedefs (7.1.3) are replaced by their definitions), the types specified by all declarations of a particular external name shall be identical, except that declarations for an array object can specify array types that differ by the presence or absence of a major array bound (8.3.4), and declarations for functions with the same name can specify different numbers and types of parameters (8.3.5). A violation of this rule on type identity does not require a diagnostic.

10 [*Note:* linkage to non-C++ declarations can be achieved using a *linkage-specification* (7.5).]

3.6 Start and termination [basic.start]

3.6.1 Main function [basic.start.main]

1 A program shall contain a global function called `main`, which is the designated start of the program.

2 This function is not predefined by the implementation, it cannot be overloaded, and its type is implementation-defined. All implementations shall allow both of the following definitions of `main`:

```
        int main() { /* ... */ }
```

and

```
        int main(int argc, char* argv[]) { /* ... */ }
```

In the latter form `argc` shall be the number of arguments passed to the program from the environment in which the program is run. If `argc` is nonzero these arguments shall be supplied in `argv[0]` through `argv[argc-1]` as pointers to the initial characters of null-terminated multibyte strings (NTMBSs) and `argv[0]` shall be the pointer to the initial character of a NTMBS that represents the name used to invoke the program or `""`. The value of `argc` shall be nonnegative. The value of `argv[argc]` shall be 0. [*Note:* It is recommended that any further (optional) parameters be added after `argv`.]

3 The function `main()` shall not be called from within a program. The linkage (3.5) of `main()` is implementation-defined. The address of `main()` shall not be taken and `main()` shall not be declared `inline` or `static`. The name `main` is not otherwise reserved. [*Example:* member functions, classes, and enumerations can be called `main`, as can entities in other namespaces.]

4 Calling the function

```
void exit(int);
```

declared in `<cstdlib>` (18.3) terminates the program without leaving the current block and hence without destroying any objects with automatic storage duration (12.4). The argument value is returned to the program's environment as the value of the program.

5 A return statement in `main()` has the effect of leaving the main function (destroying any objects with automatic storage duration) and calling `exit()` with the return value as the argument. If control reaches the end of `main` without encountering a `return` statement, the effect is that of executing

```
return 0;
```

3.6.2 Initialization of non-local objects [basic.start.init]

1 The initialization of nonlocal objects with static storage duration (3.7) defined in a translation unit is done before the first use of any function or object defined in that translation unit. Such initializations (8.5, 9.5, 12.1, 12.6.1) can be done before the first statement of `main()` or deferred to any point in time before the first use of a function or object defined in that translation unit. The storage for objects with static storage duration is zero-initialized (8.5) before any other initialization takes place. Objects with static storage duration initialized with constant expressions (5.19) are initialized before any dynamic (that is, run-time) initialization takes place. The order of initialization of nonlocal objects with static storage duration defined in the same translation unit is the order in which their definition appears in this translation unit. No further order is imposed on the initialization of objects from different translation units. The initialization of local static objects is described in 6.7.

2 If construction or destruction of a non-local static object ends in throwing an uncaught exception, the result is to call `terminate()` (18.6.2.3).

3.6.3 Termination [basic.start.term]

1 Destructors (12.4) for initialized static objects are called when returning from `main()` and when calling `exit()` (18.3). Destruction is done in reverse order of initialization. The function `atexit()` from `<cstdlib>` can be used to specify a function to be called at exit. If `atexit()` is to be called, the implementation shall not destroy objects initialized before an `atexit()` call until after the function specified in the `atexit()` call has been called.

2 Where a C++ implementation coexists with a C implementation, any actions specified by the C implementation to take place after the `atexit()` functions have been called take place after all destructors have been called.

3 Calling the function

```
void abort();
```

declared in `<cstdlib>` terminates the program without executing destructors for static objects and without calling the functions passed to `atexit()`.

3.7 Storage duration [basic.stc]

1 Storage duration is a property of an object that indicates the potential time extent the storage in which the object resides might last. The storage duration is determined by the construct used to create the object and is one of the following:

— static storage duration

— automatic storage duration

— dynamic storage duration

2 Static and automatic storage durations are associated with objects introduced by declarations (3.1). The dynamic storage duration is associated with objects created with `operator new` (5.3.4).

3 The storage class specifiers `static`, `auto`, and `mutable` are related to storage duration as described below.

4 References (8.3.2) might or might not require storage; however, the storage duration categories apply to references as well.

3.7.1 Static storage duration [basic.stc.static]

1 All non-local objects have *static storage duration*. The storage for these objects can last for the entire duration of the program. These objects are initialized and destroyed as described in 3.6.2 and 3.6.3.

2 Note that if an object of static storage duration has initialization or a destructor with side effects, it shall not be eliminated even if it appears to be unused.

3 The keyword `static` can be used to declare a local variable with static storage duration; for a description of initialization and destruction of local `static` variables, see 6.7.

4 The keyword `static` applied to a class data member in a class definition gives the data member static storage duration.

3.7.2 Automatic storage duration [basic.stc.auto]

1 Local objects explicitly declared `auto` or `register` or not explicitly declared `static` have *automatic storage duration*. The storage for these objects lasts until the block in which they are created exits.

2 [*Note:* These objects are initialized and destroyed as described 6.7.]

3 If a named automatic object has initialization or a destructor with side effects, it shall not be destroyed before the end of its block, nor shall it be eliminated as an optimization even if it appears to be unused.

3.7.3 Dynamic storage duration [basic.stc.dynamic]

1 Objects can be created dynamically during program execution (1.8), using *new-expression*s (5.3.4), and destroyed using *delete-expression*s (5.3.5). A C++ implementation provides access to, and management of, dynamic storage via the global *allocation functions* `operator new` and `operator new[]` and the global *deallocation functions* `operator delete` and `operator delete[]`.

2 These functions are always implicitly declared. The library provides default definitions for them (18.4.1). A C++ program shall provide at most one definition of any of the functions `::operator new(size_t)`, `::operator new[](size_t)`, `::operator delete(void*)`, and/or `::operator delete[](void*)`. Any such function definitions replace the default versions. This replacement is global and takes effect upon program startup (3.6). Allocation and/or deallocation functions can also be declared and defined for any class (12.5).

3 Any allocation and/or deallocation functions defined in a C++ program shall conform to the semantics specified in this subclause.

3.7.3.1 Allocation functions [basic.stc.dynamic.allocation]

1 Allocation functions can be static class member functions or global functions. They can be overloaded, but the return type shall always be `void*` and the first parameter type shall always be `size_t` (5.3.3), an implementation-defined integral type defined in the standard header `<cstddef>` (18). For these functions, parameters other than the first can have associated default arguments (8.3.6).

2 The function shall return the address of a block of available storage at least as large as the requested size. The order, contiguity, and initial value of storage allocated by successive calls to an allocation function is unspecified. The pointer returned is suitably aligned so that it can be assigned to a pointer of any type and then used to access such an object or an array of such objects in the storage allocated (until the storage is

explicitly deallocated by a call to a corresponding deallocation function). Each such allocation shall yield a pointer to storage (1.5) disjoint from any other currently allocated storage. The pointer returned points to the start (lowest byte address) of the allocated storage. If the size of the space requested is zero, the value returned shall be nonzero and shall not pointer to or within any other currently allocated storage. The results of dereferencing a pointer returned as a request for zero size are undefined.[18)]

3 If an allocation function is unable to obtain an appropriate block of storage, it can invoke the currently installed `new_handler`[19)] and/or throw an exception (15) of class `bad_alloc` (18.4.2.1) or a class derived from `bad_alloc`.

4 If the allocation function returns the null pointer the result is implementation-defined.

3.7.3.2 Deallocation functions [basic.stc.dynamic.deallocation]

1 Like allocation functions, deallocation functions can be static class member functions or global functions.

2 Each deallocation function shall return `void` and its first parameter shall be `void*`. For class member deallocation functions, a second parameter of type `size_t` may be added. If both versions are declared in the same class, the one-parameter form is the usual deallocation function and the two-parameter form is used for placement delete (5.3.4). If the second version is declared but not the first, it is the usual deallocation function, not placement delete.

3 The value of the first parameter supplied to a deallocation function shall be zero, or refer to storage allocated by the corresponding allocation function (even if that allocation function was called with a zero argument). If the value of the first argument is zero, the call to the deallocation function has no effect. If the value of the first argument refers to a pointer already deallocated, the effect is undefined.

4 A deallocation function can free the storage referenced by the pointer given as its argument and renders the pointer *invalid*. The storage can be made available for further allocation. An invalid pointer contains an unusable value: it cannot even be used in an expression.

5 If the argument is non-zero, the value of a pointer that refers to deallocated space is *indeterminate*. The effect of dereferencing an indeterminate pointer value is undefined.[20)]

3.7.4 Duration of sub-objects [basic.stc.inherit]

1 The storage duration of member subobjects, base class subobjects and array elements is that of their complete object (1.6).

3.8 Object Lifetime [basic.life]

1 The *lifetime* of an object is a runtime property of the object. The lifetime of an object of type `T` begins when:

— storage with the proper alignment and size for type `T` is obtained, and

— if `T` is a class type with a non-trivial constructor (12.1), the constructor call has completed.

The lifetime of an object of type `T` ends when:

— if `T` is a class type with a non-trivial destructor (12.4), the destructor call starts, or

— the storage which the object occupies is reused or released.

18) The intent is to have `operator new()` implementable by calling `malloc()` or `calloc()`, so the rules are substantially the same. C++ differs from C in requiring a zero request to return a non-null pointer.

19) A program-supplied allocation function can obtain the address of the currently installed `new_handler` (18.4.2.2) using the `set_new_handler()` function (18.4.2.3).

20) On some architectures, it causes a system-generated runtime fault.

2 [*Note:* The lifetime of an object of POD type starts as soon as storage with proper size and alignment is obtained, and its lifetime ends when the storage which the object occupies is reused or released. Subclause 12.6.2 describes the lifetime of base and member subobjects.]

3 The properties ascribed to objects throughout this International Standard apply for a given object only during its lifetime. In particular, except as noted during object construction (12.6.2) and destruction (12.7), before the lifetime of the object starts and after its lifetime ends the value of the storage which the object occupies is indeterminate and, for an object of non-POD class type, referring to a non-static data member, calling a non-static member function or converting the object to a base class subobject results in undefined behavior.

4 [*Note:* The behavior of an object under construction and destruction might not be the same as the behavior of an object whose lifetime has started and not ended. Subclauses 12.6.2 and 12.7 describe the behavior of an object during the construction and destruction phases.]

5 A program may end the lifetime of any object by reusing the storage which the object occupies or by explicitly calling the destructor for an object of a class type with a non-trivial destructor. For an object of a class type with a non-trivial destructor, the program is not required to call the destructor explicitly before the storage which the object occupies is reused or released; however, if there is no explicit call to the destructor or if a *delete-expression* (5.3.5) is not used to release the storage, the destructor is not implicitly called and any program that depends on the side effects produced by the destructor has unspecified behavior.

6 After the lifetime of an object has ended and while the storage which the object occupied still exists, any pointer to the original object can be used but only in limited ways. Such a pointer still points to valid storage and using the pointer as a pointer to the storage where the object was located, as if the pointer were of type `void*`, is well-defined. However, using the pointer to refer to the original object is no longer valid. In particular, such a pointer cannot be dereferenced; for a non-POD class type `T`, a pointer of type `T*` that points to the original object cannot be the operand of a `static_cast` (5.2.8) (except when the conversion is to `void*` or `char*`) and cannot be the operand of a `dynamic_cast` (5.2.6); if `T` is a class with a non-trivial destructor, such a pointer cannot be used as the operand of a *delete-expression.* [*Example:*

```
struct B {
        virtual void f();
        void mutate();
        virtual ~B();
};

struct D1 : B { void f(); };
struct D2 : B { void f(); };

void B::mutate() {
        new (this) D2;  // reuses storage - ends the lifetime of '*this'
        f();            // undefined behavior
        ... = this;     // ok, 'this' points to valid memory
}

void g() {
        void* p = malloc(sizeof(D1) + sizeof(D2));
        B* pb = new (p) D1;
        pb->mutate();
        &pb;            // ok: pb points to valid memory
        void* q = pb;   // ok: pb points to valid memory
        pb->f();        // undefined behavior, lifetime of *pb has ended
}
```

—*end example*]

7 If, after the lifetime of an object has ended and while the storage which the object occupied still exists, a new object is created at the storage location which the original object occupied, a pointer that pointed to the original object will automatically refer to the new object and, once the lifetime of the new object has

started, can be used to manipulate the new object, if:

— the storage for the new object exactly overlays the storage location which the original object occupied, and

— the new object is of the same type as the original object (ignoring the top-level cv-qualifiers), and

— the original object was a complete object of type `T` and the new object is a complete object of type `T` (that is, they are not base class subobjects). [*Example:*

```
struct C {
        int i;
        void f();
        const C& operator=( const C& );
};

const C& C::operator=( const C& other)
{
        if ( this != &other )
        {
                this->~C();          // lifetime of '*this' ends
                new (this) C(other); // new object of type C created
                f();                 // well-defined
        }
        return *this;
}

C c1;
C c2;
c1 = c2; // well-defined
c1.f();  // well-defined; c1 refers to a new object of type C
```

—*end example*]

8 If a program ends the lifetime of an object of type `T` with static (3.7.1) or automatic (3.7.2) storage duration and if `T` has a non-trivial destructor,[21] the program must ensure that an object of the original type occupies that same storage location when the implicit destructor call takes place; otherwise the behavior of the program is undefined. This is true even if the block is exited with an exception. [*Example:*

```
struct B {
        ~B();
};
void h() {
        B b;
        new (&b) T;
} // undefined behavior at block exit
```

—*end example*]

3.9 Types [basic.types]

1 This clauses imposes requirements on processors regarding the representation of types. There are two kinds of types: fundamental types and compound types. Types describe objects (1.6), references (8.3.2), or functions (8.3.5).

2 For any object type `T`, the underlying bytes (1.5) of the object can be copied (using the `memcpy` library function (17.3.1.2) into an array of `char` or `unsigned char`. The copy operation is well-defined, even if the object does not hold a valid value of type `T`. Whether or not the value of the object is later changed, if the content of the array of `char` or `unsigned char` is copied back into the object using the `memcpy`

[21] that is, an object for which a destructor will be called implicitly -- either upon exit from the block for an object with automatic storage duration or upon exit from the program for an object with static storage duration.

library function, the object shall subsequently hold its original value. [*Example:*

```
#define N sizeof(T)
char buf[N];
T obj;  // obj initialized to its original value
memcpy(buf, &obj, N);
        // between these two calls to memcpy,
        // obj might be modified
memcpy(&obj, buf, N);
        // at this point, each subobject of obj of scalar type
        // holds its original value
```

—*end example*]

3 For any scalar type `T`, if two pointers to `T` point to distinct `T` objects `obj1` and `obj2`, if the value of `obj1` is copied into `obj2`, using the `memcpy` library function, `obj2` shall subsequently hold the same value as `obj1`. [*Example:*

```
T* t1p;
T* t2p;
        // provided that t2p points to an initialized object ...
memcpy(t1p, t2p, sizeof(T));
        // at this point, every subobject of scalar type in *t1p
        // contains the same value as the corresponding subobject in
        // *t2p
```

—*end example*]

4 The *object representation* of an object of type `T` is the sequence of *N* `unsigned char` objects taken up by the object of type `T`, where *N* equals `sizeof(T)`. The *value representation* of an object is the sequence of bits in the object representation that hold the value of type `T`. The bits of the value representation determine a *value*, which is one discrete element of an implementation-defined set of values.[22]

5 Object types have *alignment requirements* (3.9.1, 3.9.2). The alignment of an object type is an implementation-defined integer value representing a number of bytes; an object is allocated at an address that is divisible by the alignment of its object type.

6 Arrays of unknown size and classes that have been declared but not defined are called *incomplete* types.[23] Also, the `void` type is an incomplete type; it represents an empty set of values. No objects shall be defined to have incomplete type. The term *incompletely-defined object type* is a synonym for *incomplete type*; the term *completely-defined object type* is a synonym for *complete type*;

7 A class type (such as "`class X`") might be incomplete at one point in a translation unit and complete later on; the type "`class X`" is the same type at both points. The declared type of an array might be incomplete at one point in a translation unit and complete later on; the array types at those two points ("array of unknown bound of `T`" and "array of N `T`") are different types. However, the type of a pointer to array of unknown size, or of a type defined by a `typedef` declaration to be an array of unknown size, cannot be completed. [*Example:*

```
class X;             // X is an incomplete type
extern X* xp;        // xp is a pointer to an incomplete type
extern int arr[];    // the type of arr is incomplete
typedef int UNKA[];  // UNKA is an incomplete type
UNKA* arrp;          // arrp is a pointer to an incomplete type
UNKA** arrpp;
```

[22] The intent is that the memory model of C++ is compatible with that of ISO/IEC 9899 Programming Language C.
[23] The size and layout of an instance of an incomplete type is unknown.

```
        void foo()
        {
            xp++;              // ill-formed:  X is incomplete
            arrp++;            // ill-formed:  incomplete type
            arrpp++;           // okay: sizeof UNKA* is known
        }

        struct X { int i; };  // now X is a complete type
        int  arr[10];         // now the type of arr is complete

        X x;
        void bar()
        {
            xp = &x;           // okay; type is ‘‘pointer to X’’
            arrp = &arr;       // ill-formed: different types
            xp++;              // okay:  X is complete
            arrp++;            // ill-formed:  UNKA can’t be completed
        }
```

—*end example*]

8 [*Note:* Clause 5, 6 and 7 describe in which contexts incomplete types are prohibited.]

9 Arithmetic and enumeration types (3.9.1) and pointer types (3.9.2) are *scalar types*. Scalar types, POD class types, POD union types (9) and arrays of such types are *POD types*.

10 If two types `T1` and `T2` are the same type, then `T1` and `T2` are *layout-compatible* types. [*Note:* Layout-compatible enumerations are described in 7.2. Layout-compatible POD-structs and POD-unions are described in 9.2.]

3.9.1 Fundamental types [basic.fundamental]

1 There are several fundamental types. Specializations of the standard template `numeric_limits` (18.2) shall specify the largest and smallest values of each for an implementation.

2 Objects declared as characters (`char`) shall be large enough to store any member of the implementation’s basic character set. If a character from this set is stored in a character object, its value shall be equivalent to the integer code of that character. It is implementation-defined whether a `char` object can take on negative values. Characters can be explicitly declared `unsigned` or `signed`. Plain `char`, `signed char`, and `unsigned char` are three distinct types. A `char`, a `signed char`, and an `unsigned char` occupy the same amount of storage and have the same alignment requirements (3.9); that is, they have the same object representation. For character types, all bits of the object representation participate in the value representation. For unsigned character types, all possible bit patterns of the value representation represent numbers. These requirements do not hold for other types. In any particular implementation, a plain `char` object can take on either the same values as a `signed char` or an `unsigned char`; which one is implementation-defined.

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

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

5 For each of the signed integer types, there exists a corresponding (but different) *unsigned integer type*: “`unsigned char`”, “`unsigned short int`”, “`unsigned int`”, and “`unsigned long int,`” each of which occupies the same amount of storage and has the same alignment requirements (3.9) as the corresponding signed integer type[24] ; that is, each signed integer type has the same object

[24] See 7.1.5.2 regarding the correspondence between types and the sequences of *type-specifier*s that designate them.

representation as its corresponding unsigned integer type. The range of nonnegative values of a *signed integer* type is a subrange of the corresponding *unsigned integer* type, and the value representation of the same value in each type shall be the same.

6 Unsigned integers, declared `unsigned`, shall obey the laws of arithmetic modulo 2^n where n is the number of bits in the representation of that particular size of integer.[25)]

7 Type `wchar_t` is a distinct type whose values can represent distinct codes for all members of the largest extended character set specified among the supported locales (22.1.1). Type `wchar_t` shall have the same size, signedness, and alignment requirements (1.5) as one of the other integral types, called its *underlying type*.

8 Values of type `bool` are either `true` or `false`.[26)] There are no `signed`, `unsigned`, `short`, or `long bool` types or values. As described below, `bool` values behave as integral types. Values of type `bool` participate in integral promotions (4.5, 5.2.3). Although values of type `bool` generally behave as signed integers, for example by promoting (4.5) to `int` instead of `unsigned int`, a `bool` value can successfully be stored in a bit-field of any (nonzero) size.

9 Types `bool`, `char`, `wchar_t`, and the signed and unsigned integer types are collectively called *integral* types.[27)] A synonym for integral type is *integer type*. The representations of integral types shall define values by use of a pure binary numeration system.

10 There are three *floating point* types: `float`, `double`, and `long double`. The type `double` provides at least as much precision as `float`, and the type `long double` provides at least as much precision as `double`. The value representation of floating-point is implementation-defined. *Integral* and *floating* types are collectively called *arithmetic* types.

11 The `void` type has an empty set of values. It is used as the return type for functions that do not return a value. Objects of type `void` shall not be declared. Any expression can be explicitly converted to type `void` (5.4); the resulting expression shall be used only as an expression statement (6.2), as the left operand of a comma expression (5.18), or as a second or third operand of `?:` (5.16).

12 [*Note:* Even if the implementation defines two or more basic types to have the same value representation, they are nevertheless different types.]

3.9.2 Compound types [basic.compound]

1 There is a conceptually infinite number of compound types constructed from the fundamental types in the following ways:

— *arrays* of objects of a given type, 8.3.4;

— *functions*, which have parameters of given types and return `void` or references or objects of a given type, 8.3.5;

— *pointers* to `void` or objects or functions (including static members of classes) of a given type, 8.3.1;

— *references* to objects or functions of a given type, 8.3.2;

— *constants*, which are values of a given type, 7.1.5;

— *classes* containing a sequence of objects of various types (9), a set of functions for manipulating these objects (9.4), and a set of restrictions on the access to these objects and functions, 11;

— *unions*, which are classes capable of containing objects of different types at different times, 9.6;

25) This implies that unsigned arithmetic does not overflow.
26) Using a `bool` value in ways described by this International Standard as ‘‘undefined,’’ such as by examining the value of an uninitialized automatic variable, might cause it to behave as if is neither `true` nor `false`.
27) Therefore, enumerations (7.2) are not integral; however, enumerations can be promoted to `int`, `unsigned int`, `long`, or `unsigned long`, as specified in 4.5.

— *pointers to non-static*[28] *class members*, which identify members of a given type within objects of a given class, 8.3.3.

2 These methods of constructing types can be applied recursively; restrictions are mentioned in 8.3.1, 8.3.4, 8.3.5, and 8.3.2.

3 A pointer to objects of type `T` is referred to as a "pointer to `T`." [*Example:* a pointer to an object of type `int` is referred to as "pointer to `int`" and a pointer to an object of class `X` is called a "pointer to `X`."] Except for pointers to static members, text referring to "pointers" does not apply to pointers to members. Pointers to incomplete types are allowed although there are restrictions on what can be done with them (3.9). The value representation of pointer types is implementation-defined. Pointers to cv-qualified and cv-unqualified versions (3.9.3) of layout-compatible types shall have the same value representation and alignment requirements (3.9).

4 Objects of cv-qualified (3.9.3) or cv-unqualified type `void*` (pointer to void), can be used to point to objects of unknown type. A `void*` shall be able to hold any object pointer. A cv-qualified or cv-unqualified (3.9.3) `void*` shall have the same representation and alignment requirements as a cv-qualified or cv-unqualified `char*`.

5 Except for pointers to static members, text referring to "pointers" does not apply to pointers to members.

3.9.3 CV-qualifiers [basic.type.qualifier]

1 A type mentioned in 3.9.1 and 3.9.2 is a *cv-unqualified type*. Each cv-unqualified fundamental type (3.9.1) has three corresponding cv-qualified versions of its type: a *const-qualified* version, a *volatile-qualified* version, and a *const-volatile-qualified* version. The term *object type* (1.6) includes the cv-qualifiers specified when the object is created. The presence of a `const` specifier in a *decl-specifier-seq* declares an object of *const-qualified object type*; such object is called a *const object*. The presence of a `volatile` specifier in a *decl-specifier-seq* declares an object of *volatile-qualified object type*; such object is called a *volatile object*. The presence of both *cv-qualifiers* in a *decl-specifier-seq* declares an object of *const-volatile-qualified object type*; such object is called a *const volatile object*. The cv-qualified or cv-unqualified versions of a type are distinct types; however, they shall have the same representation and alignment requirements (3.9).[29]

2 A compound type (3.9.2) is not cv-qualified by the cv-qualifiers (if any) of the type from which it is compounded. However, any cv-qualifiers that appears in an array declaration apply to the array element type, not the array type (8.3.4).

3 Each non-function, non-static, non-mutable member of a const-qualified class object is const-qualified, each non-function, non-static member of a volatile-qualified class object is volatile-qualified and similarly for members of a const-volatile class. See 8.3.5 and 9.4.2 regarding cv-qualified function types.

4 There is a (partial) ordering on cv-qualifiers, so that a type can be said to be *more cv-qualified* than another. Table 6 shows the relations that constitute this ordering.

Table 6—relations on `const` and `volatile`

no cv-qualifier	<	`const`
no cv-qualifier	<	`volatile`
no cv-qualifier	<	`const volatile`
`const`	<	`const volatile`
`volatile`	<	`const volatile`

[28] Static class members are objects or functions, and pointers to them are ordinary pointers to objects or functions.

[29] The same representation and alignment requirements are meant to imply interchangeability as arguments to functions, return values from functions, and members of unions.

5 In this document, the notation *cv* (or *cv1*, *cv2*, etc.), used in the description of types, represents an arbitrary set of cv-qualifiers, i.e., one of {`const`}, {`volatile`}, {`const, volatile`}, or the empty set. Cv-qualifiers applied to an array type attach to the underlying element type, so the notation "*cv* `T`," where `T` is an array type, refers to an array whose elements are so-qualified. Such array types can be said to be more (or less) cv-qualified than other types based on the cv-qualification of the underlying element types.

3.9.4 Type names [basic.type.name]

1 [*Note:* Fundamental and compound types can be given names by the `typedef` mechanism (7.1.3), and families of types and functions can be specified and named by the `template` mechanism (14).]

3.10 Lvalues and rvalues [basic.lval]

1 Every expression is either an *lvalue* or an *rvalue*.

2 An lvalue refers to an object or function. Some rvalue expressions—those of class or cv-qualified class type—also refer to objects.[30)]

3 [*Note:* some builtin operators and function calls yield lvalues. [*Example:* if `E` is an expression of pointer type, then `*E` is an lvalue expression referring to the object or function to which `E` points. As another example, the function

```
        int& f();
```

yields an lvalue, so the call `f()` is an lvalue expression.]]

4 [*Note:* some builtin operators expect lvalue operands. [*Example:* builtin assignment operators all expect their left hand operands to be lvalues.] Other builtin operators yield rvalues, and some expect them. [*Example:* the unary and binary `+` operators expect rvalue arguments and yield rvalue results.] The discussion of each builtin operator in clause 5 indicates whether it expects lvalue operands and whether it yields an lvalue.]

5 Constructor invocations and calls to functions that do not return references are always rvalues. User defined operators are functions, and whether such operators expect or yield lvalues is determined by their type.

6 Whenever an lvalue appears in a context where an lvalue is not expected, the lvalue is converted to an rvalue; see 4.1, 4.2, and 4.3.

7 The discussion of reference initialization in 8.5.3 and of temporaries in 12.2 indicates the behavior of lvalues and rvalues in other significant contexts.

8 Class rvalues can have cv-qualified types; non-class rvalues always have cv-unqualified types. Rvalues always have complete types or the `void` type; lvalues may have incomplete types.

9 An lvalue for an object is necessary in order to modify the object except that an rvalue of class type can also be used to modify its referent under certain circumstances. [*Example:* a member function called for an object (9.4) can modify the object.]

10 Functions cannot be modified, but pointers to functions can be modifiable.

11 A pointer to an incomplete type can be modifiable. At some point in the program when this pointer type is complete, the object at which the pointer points can also be modified.

12 Array objects cannot be modified, but their elements can be modifiable.

13 The referent of a `const`-qualified expression shall not be modified (through that expression), except that if it is of class type and has a `mutable` component, that component can be modified (7.1.5.1).

[30)] Expressions such as invocations of constructors and of functions that return a class type do in some sense refer to an object, and the implementation can invoke a member function upon such objects, but the expressions are not lvalues.

14 If an expression can be used to modify its object, it is called *modifiable*. A program that attempts to modify an object through a nonmodifiable lvalue or rvalue expression is ill-formed.

4 Standard conversions [conv]

1 [*Note:* Expressions with a given type will be implicitly converted to other types in several contexts:

— When used as operands of operators. The operator's requirements for its operands dictate the destination type. See 5.

— When used in the condition of an `if` statement or iteration statement (6.4, 6.5). The destination type is `bool`.

— When used in the expression of a `switch` statement. The destination type is integral (6.4).

— When used as the source expression for an initialization (which includes use as an argument in a function call and use as the expression in a `return` statement). The type of the entity being initialized is (generally) the destination type. See 8.5, 8.5.3.

2 Standard conversions are implicit conversions defined for built-in types. For user-defined types, user-defined conversions are considered as well; see 12.3. In general, an implicit conversion sequence (13.3.3.1) consists of zero or more standard conversions and zero or one user-defined conversion.

3 There are some contexts where certain conversions are suppressed. For example, the lvalue-to-rvalue conversion is not done on the operand of the unary & operator. Specific exceptions are given in the descriptions of those operators and contexts.

—*end note*]

4 One or more of the following standard conversions will be applied to an expression if necessary to convert it to a required destination type.

4.1 Lvalue-to-rvalue conversion [conv.lval]

1 An lvalue (3.10) of a non-function, non-array type `T` can be converted to an rvalue. If `T` is an incomplete type, a program that necessitates this conversion is ill-formed. If `T` is a non-class type, the type of the rvalue is the cv-unqualified version of `T`. Otherwise (i.e., `T` is a class type), the type of the rvalue is `T`. [31)]

2 The value contained in the object indicated by the lvalue is the rvalue result. When an lvalue-to-rvalue conversion is done within the operand of `sizeof` (5.3.3) the value contained in the referenced object is not accessed, since that operator does not evaluate its operand.

3 [*Note:* See also 3.10.]

4.2 Array-to-pointer conversion [conv.array]

1 An lvalue or rvalue of type "array of `N` `T`" or "array of unknown bound of `T`" can be converted to an rvalue of type "pointer to `T`." The result is a pointer to the first element of the array.

31) In C++ class rvalues can have cv-qualified types (because they are objects). This differs from ISO C, in which non-lvalues never have cv-qualified types.

4.3 Function-to-pointer conversion [conv.func]

1 An lvalue of function type `T` can be converted to an rvalue of type “pointer to `T`.” The result is a pointer to the function.[32)]

2 [*Note:* See 13.4 for additional rules for the case where the function is overloaded.]

4.4 Qualification conversions [conv.qual]

1 An rvalue of type “pointer to *cv1* `T`” can be converted to an rvalue of type “pointer to *cv2* `T`” if “*cv2* `T`” is more cv-qualified than “*cv1* `T`.”

2 An rvalue of type “pointer to member of `X` of type *cv1* `T`” can be converted to an rvalue of type “pointer to member of `X` of type *cv2* `T`” if “*cv2* `T`” is more cv-qualified than “*cv1* `T`.”

3 A conversion can add type qualifiers at levels other than the first in multi-level pointers, subject to the following rules:[33)]

Two pointer types T1 and T2 are *similar* if there exists a type T and integer $N > 0$ such that:

$$T1 \text{ is } Tcv_{1,n} * \cdots cv_{1,1} * cv_{1,0}$$

and

$$T2 \text{ is } Tcv_{2,n} * \cdots cv_{2,1} * cv_{2,0}$$

where each $cv_{i,j}$ is `const`, `volatile`, `const volatile`, or nothing. An expression of type $T1$ can be converted to type $T2$ if and only if the following conditions are satisfied:

— the pointer types are similar.

— for every $j > 0$, if `const` is in $cv_{1,j}$ then `const` is in $cv_{2,j}$, and similarly for `volatile`.

— the $cv_{1,j}$ and $cv_{2,j}$ are different, then `const` is in every $cv_{2,k}$ for $0 < k < j$.

4 When a multi-level pointer is composed of data member pointers, or a mix of object and data member pointers, the rules for adding type qualifiers are the same as those for object pointers. That is, the “member” aspect of the pointers is irrelevant in determining where type qualifiers can be added.

4.5 Integral promotions [conv.prom]

1 An rvalue of type `char`, `signed char`, `unsigned char`, `short int`, or `unsigned short int` can be converted to an rvalue of type `int` if `int` can represent all the values of the source type; otherwise, the source rvalue can be converted to an rvalue of type `unsigned int`.

2 An rvalue of type `wchar_t` (3.9.1) or an enumeration type (7.2) can be converted to an rvalue of the first of the following types that can represent all the values of the source type: `int`, `unsigned int`, `long`, or `unsigned long`.

3 An rvalue for an integral bit-field (9.7) can be converted to an rvalue of type `int` if `int` can represent all the values of the bit-field; otherwise, it can be converted to `unsigned int` if `unsigned int` can represent all the values of the bit-field[34)].

4 An rvalue of type `bool` can be converted to an rvalue of type `int`, with `false` becoming zero and `true` becoming one.

5 These conversions are called integral promotions.

32) This conversion never applies to nonstatic member functions because there is no way to obtain an lvalue for a nonstatic member function.
33) These rules ensure that const-safety is preserved by the conversion.
34) If the bit-field is larger yet, it is not eligible for integral promotion. If the bit-field has an enumerated type, it is treated as any other value of that type for promotion purposes.

4.6 Floating point promotion [conv.fpprom]

1 An rvalue of type `float` can be converted to an rvalue of type `double`. The value is unchanged.

2 This conversion is called floating point promotion.

4.7 Integral conversions [conv.integral]

1 An rvalue of an integer type can be converted to an rvalue of another integer type.

2 If the destination type is unsigned, the resulting value is the least unsigned integer congruent to the source integer (modulo 2^n where n is the number of bits used to represent the unsigned type). [*Note:* In a two's complement representation, this conversion is conceptual and there is no change in the bit pattern (if there is no truncation).]

3 If the destination type is signed, the value is unchanged if it can be represented in the destination type (and bitfield width); otherwise, the value is implementation-defined.

4 If the destination type is `bool`, see 4.13. If the source type is `bool`, the value `false` is converted to zero and the value `true` is converted to one.

5 The conversions allowed as integral promotions are excluded from the set of integral conversions.

4.8 Floating point conversions [conv.double]

1 An rvalue of floating point type can be converted to an rvalue of another floating point type. If the source value can be exactly represented in the destination type, the result of the conversion is that exact representation. If the source value is between two adjacent destination values, the result of the conversion is an unspecified choice of either of those values. Otherwise, the behavior is undefined.

2 The conversions allowed as floating point promotions are excluded from the set of floating point conversions.

4.9 Floating-integral conversions [conv.fpint]

1 An rvalue of a floating point type can be converted to an rvalue of an integer type. The conversion truncates; that is, the fractional part is discarded. The behavior is undefined if the truncated value cannot be represented in the destination type. [*Note:* If the destination type is `bool`, see 4.13.]

2 An rvalue of an integer type can be converted to an rvalue of a floating point type. The result is exact if possible. Otherwise, it is an unspecified choice of either the next lower or higher representable value. Loss of precision occurs if the integral value cannot be represented exactly as a value of the floating type. If the source type is `bool`, the value `false` is converted to zero and the value `true` is converted to one.

4.10 Pointer conversions [conv.ptr]

1 An integral constant expression (5.19) rvalue that evaluates to zero (called a *null pointer constant*) can be converted to a pointer type. The result is a value (called the *null pointer value* of that type) distinguishable from every pointer to an object or function. Two null pointer values of a given type compare equal.

2 An rvalue of type "pointer to *cv* `T`," where `T` is an object type, can be converted to an rvalue of type "pointer to *cv* `void`." The result of converting a "pointer to *cv* `T`" to a "pointer to *cv* `void`" points to the start of the storage location where the object of type `T` resides, as if the object is a complete object of type `T` (that is, not a base class subobject).

3 An rvalue of type "pointer to *cv* `D`," where `D` is a class type, can be converted to an rvalue of type "pointer to *cv* `B`," where `B` is a base class (10) of `D`. If `B` is an inaccessible (11) or ambiguous (10.2) base class of `D`, a program that necessitates this conversion is ill-formed. The result of the conversion is a pointer to the base class sub-object of the derived class object. The null pointer value is converted to the null pointer value of the destination type.

4.11 Pointer to member conversions [conv.mem]

1 A null pointer constant (4.10) can be converted to a pointer to member type. The result is a value (called the *null member pointer value* of that type) distinguishable from a pointer to any member. Two null member pointer values of a given type compare equal.

2 An rvalue of type "pointer to member of `B` of type *cv* `T`," where `B` is a class type, can be converted to an rvalue of type "pointer to member of `D` of type *cv* `T`," where `D` is a derived class (10) of `B`. If `B` is an inaccessible (11) or ambiguous (10.2) base class of `D`, a program that necessitates this conversion is ill-formed. The result of the conversion refers to the same member as the pointer to member before the conversion took place, but it refers to the base class member as if it were a member of the derived class. The result refers to the member in `D`'s instance of `B`. Since the result has type "pointer to member of `D` of type *cv* `T`," it can be dereferenced with a `D` object. The result is the same as if the pointer to member of `B` were dereferenced with the `B` sub-object of `D`. The null member pointer value is converted to the null member pointer value of the destination type.[35)]

4.12 Base class conversion [conv.class]

1 An rvalue of type "*cv* `D`," where `D` is a class type, can be converted to an rvalue of type "*cv* `B`," where `B` is a base class (10) of `D`. If `B` is an inaccessible (11) or ambiguous (10.2) base class of `D`, or if the conversion is implemented by calling a constructor (12.3.1) and the constructor is not callable, a program that necessitates this conversion is ill-formed. The result of the conversion is the value of the base class sub-object of the derived class object.

4.13 Boolean conversions [conv.bool]

1 An rvalue of arithmetic, enumeration, pointer, or pointer to member type can be converted to an rvalue of type `bool`. A zero value, null pointer value, or null member pointer value is converted to `false`; any other value is converted to `true`.

35) The rule for conversion of pointers to members (from pointer to member of base to pointer to member of derived) appears inverted compared to the rule for pointers to objects (from pointer to derived to pointer to base) (4.10, 10). This inversion is necessary to ensure type safety. Note that a pointer to member is not a pointer to object or a pointer to function and the rules for conversions of such pointers do not apply to pointers to members. In particular, a pointer to member cannot be converted to a `void*`.

5 Expressions [expr]

1 [*Note:* this clause defines the syntax, order of evaluation, and meaning of expressions. An expression is a sequence of operators and operands that specifies a computation. An expression can result in a value and can cause side effects.

2 Operators can be overloaded, that is, given meaning when applied to expressions of class type (9). Uses of overloaded operators are transformed into function calls as described in 13.5. Overloaded operators obey the rules for syntax specified in this clause, but the requirements of operand type, lvalue, and evaluation order are replaced by the rules for function call. Relations between operators, such as `++a` meaning `a+=1`, are not guaranteed for overloaded operators (13.5).[36]]

3 This clause defines the operators when applied to types for which they have not been overloaded. Operator overloading shall not modify the rules for the *built-in operators*, that is, for operators applied to types for which they are defined by the language itself. However, these built-in operators participate in overload resolution; see 13.3.1.2.

4 Operators can be regrouped according to the usual mathematical rules only where the operators really are associative or commutative. Overloaded operators are never assumed to be associative or commutative. Except where noted, the order of evaluation of operands of individual operators and subexpressions of individual expressions, and the order in which side effects take place, is unspecified. Between the previous and next sequence point a scalar object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be accessed only to determine the value to be stored. The requirements of this paragraph shall be met for each allowable ordering of the subexpressions of a full expression; otherwise the behavior is undefined. [*Example:*

```
i = v[i++];       // the behavior is undefined
i = 7,i++,i++;    // 'i' becomes 9

i = ++i + 1;      // the behavior is undefined
i = i + 1;        // the value of 'i' is incremented
```

—end example]

5 If during the evaluation of an expression, the result is not mathematically defined or not in the range of representable values for its type, the behavior is undefined. [*Note:* most existing implementations of C++ ignore integer overflows. Treatment of division by zero and all floating point exceptions vary among machines, and is usually adjustable by a library function.]

6 Except where noted, operands of types `const T`, `volatile T`, `T&`, `const T&`, and `volatile T&` can be used as if they were of the plain type `T`. Similarly, except where noted, operands of type `T* const` and `T* volatile` can be used as if they were of the plain type `T*`. Similarly, a plain `T` can be used where a `volatile T` or a `const T` is required. These rules apply in combination so that, except where noted, a `T* const volatile` can be used where a `T*` is required. Such uses do not count as standard conversions when considering overloading resolution (13.3).

[36] Nor is it guaranteed for type `bool`; the left operand of `+=` shall not have type `bool`.

7 If an expression initially has the type “reference to `T`” (8.3.2, 8.5.3), the type is adjusted to “`T`” prior to any further analysis, the expression designates the object or function denoted by the reference, and the expression is an lvalue. A reference can be thought of as a name of an object.

8 An expression designating an object is called an *object-expression.*

9 User-defined conversions of class types to and from fundamental types, pointers, and so on, can be defined (12.3). If unambiguous (13.3), such conversions are applied wherever a class object appears as an operand of an operator or as a function argument (5.2.2).

10 Whenever an lvalue expression appears as an operand of an operator that expects an rvalue for that operand, the lvalue-to-rvalue (4.1), array-to-pointer (4.2), or function-to-pointer (4.3) standard conversion are applied to convert the expression to an rvalue.

11 Many binary operators that expect operands of arithmetic type cause conversions and yield result types in a similar way. The purpose is to yield a common type, which is also the type of the result. This pattern is called the “usual arithmetic conversions.”

12 The processor shall perform the following conversions on operands of arithmetic type:

— If either operand is of type `long double`, the other shall be converted to `long double`.

— Otherwise, if either operand is `double`, the other shall be converted to `double`.

— Otherwise, if either operand is `float`, the other shall be converted to `float`.

— Otherwise, the integral promotions (4.5) shall be performed on both operands.[37)]

— Then, if either operand is `unsigned long` the other shall be converted to `unsigned long`.

— Otherwise, if one operand is a `long int` and the other `unsigned int`, then if a `long int` can represent all the values of an `unsigned int`, the `unsigned int` shall be converted to a `long int`; otherwise both operands shall be converted to `unsigned long int`.

— Otherwise, if either operand is `long`, the other shall be converted to `long`.

— Otherwise, if either operand is `unsigned`, the other shall be converted to `unsigned`.

[*Note:* otherwise, the only remaining case is that both operands are `int`]

13 If the program attempts to access the stored value of an object through an lvalue of other than one of the following types the behavior is undefined:

— the dynamic type of the object,

— a cv-qualified version of the declared type of the object,

— a type that is the signed or unsigned type corresponding to the declared type of the object,

— a type that is the signed or unsigned type corresponding to a cv-qualified version of the declared type of the object,

— an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union),

— a type that is a (possibly cv-qualified) base class type of the declared type of the object,

— a `char` or `unsigned char` type.[38)]

37) As a consequence, operands of type `bool`, `wchar_t`, or an enumerated type are converted to some integral type.
38) The intent of this list is to specify those circumstances in which an object may or may not be aliased.

5.1 Primary expressions [expr.prim]

1 Primary expressions are literals, names, and names qualified by the scope resolution operator `::`.

> *primary-expression:*
> > *literal*
> > `this`
> > `::` *identifier*
> > `::` *operator-function-id*
> > `::` *qualified-id*
> > `(` *expression* `)`
> > *id-expression*

2 A *literal* is a primary expression. Its type depends on its form (2.9).

3 The keyword `this` names a pointer to the object for which a nonstatic member function (9.4.2) is invoked. The keyword `this` shall be used only inside a nonstatic class member function body (9.4) or in a constructor *mem-initializer* (12.6.2).

4 The operator `::` followed by an *identifier*, a *qualified-id*, or an *operator-function-id* is a *primary-expression*. Its type is specified by the declaration of the identifier, name, or *operator-function-id*. The result is the identifier, name, or *operator-function-id*. The result is an lvalue if the identifier, name, or *operator-function-id* is. The identifier, name, or *operator-function-id* shall be of global namespace scope. [*Note:* the use of `::` allows a type, an object, a function, or an enumerator declared in the global namespace to be referred to even if its identifier has been hidden (3.3).]

5 A parenthesized expression is a primary expression whose type and value are identical to those of the enclosed expression. The presence of parentheses does not affect whether the expression is an lvalue.

6 A *id-expression* is a restricted form of a *primary-expression* that can appear after `.` and `->` (5.2.4):

> *id-expression:*
> > *unqualified-id*
> > *qualified-id*
>
> *unqualified-id:*
> > *identifier*
> > *operator-function-id*
> > *conversion-function-id*
> > ˜ *class-name*
> > *template-id*

7 An *identifier* is an *id-expression* provided it has been suitably declared (7). [*Note:* for *operator-function-id*s, see 13.5; for *conversion-function-id*s, see 12.3.2. A *class-name* prefixed by ~ denotes a destructor; see 12.4.]

8
> *qualified-id:*
> > *nested-name-specifier* `template`$_{opt}$ *unqualified-id*

A *nested-name-specifier* that names a class (7.1.5) followed by `::`, optionally followed by the keyword `template` (14.10.1), and then followed by the name of a member of either that class (9.2) or one of its base classes (10), is a *qualified-id*. If the qualified-id refers to a non-static member, its type is the data member type or function member type (9.2); if it refers to a static member, its type is an object or function type (9.5). The result is the member. The result is an lvalue if the member is. If the *class-name* is hidden by a name that is not a type name or *namespace-name*, the *class-name* is still found and used. Where *class-name* `::` *class-name* is used, and the two *class-name*s refer to the same class, this notation names the constructor (12.1). Where *class-name* `:: ~` *class-name* is used, the two *class-name*s shall refer to the same class; this notation names the destructor (12.4).

9 A *nested-name-specifier* that names a namespace (7.3) followed by `::`, followed by the name of a member of that namespace is a *qualified-id*; names introduced by *using-directives* (7.3.4) in the namespace denoted

by the *nested-name-specifier* are ignored for the purpose of this member lookup. The type of the *qualified-id* is the type of the member. The result is the member. The result is an lvalue if the member is. If the *namespace-name* is hidden by a name that is not a type name, the *namespace-name* is still found and used.

10 Multiply qualified names, such as `N1::N2::N3::n`, can be used to refer to nested types (9.8).

11 In a *qualified-id*, if the *id-expression* is a *conversion-function-id*, its *conversion-type-id* shall denote the same type in both the context in which the entire *qualified-id* occurs and in the context of the class denoted by the *nested-name-specifier*.

12 An *id-expression* that denotes a nonstatic member of a class can only be used:

— as part of a class member access (5.2.4) in which the object-expression refers to the member's class or a class derived from that class, or

— to form a pointer to member (5.3.1), or

— in the body of a nonstatic member function of that class or of a class derived from that class (9.4.1), or

— in a *mem-initializer* for a constructor for that class or for a class derived from that class (12.6.2).

13 A *template-id* shall be used as an *unqualified-id* only as specified in clauses 14.4, 14.5, and 14.6.

5.2 Postfix expressions [expr.post]

1 Postfix expressions group left-to-right.

```
postfix-expression:
        primary-expression
        postfix-expression [ expression ]
        postfix-expression ( expression-list_opt )
        simple-type-specifier ( expression-list_opt )
        postfix-expression . template_opt id-expression
        postfix-expression -> template_opt 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
```

5.2.1 Subscripting [expr.sub]

1 A postfix expression followed by an expression in square brackets is a postfix expression. [*Note:* the intuitive meaning is that of a subscript.] One of the expressions shall have the type "pointer to `T`" and the other shall be of enumeration or integral type. The result is an lvalue of type "`T`." The type "`T`" shall be complete. The expression `E1[E2]` is identical (by definition) to `*((E1)+(E2))`. [*Note:* see 5.3 and 5.7 for details of `*` and `+` and 8.3.4 for details of arrays.]

5.2.2 Function call [expr.call]

1 There are two kinds of function call: ordinary function call and member function[39] (9.4) call. A function call is a postfix expression followed by parentheses containing a possibly empty, comma-separated list of

[39] A static member function (9.5) is an ordinary function.

expressions which constitute the arguments to the function. For ordinary function call, the postfix expression shall be a function name, or a pointer or reference to a function. For member function call, the postfix expression shall be an implicit (9.4.1, 9.5) or explicit class member access (5.2.4) whose *id-expression* is a function member name, or a pointer-to-member expression (5.5) selecting a function member. The first expression in the postfix expression is then called the *object expression*, and the call is as a member of the object pointed to or referred to. In the case of an implicit class member access, the implied object is the one pointed to by `this`. [*Note:* a member function call of the form `f()` is interpreted as `(*this).f()` (see 9.4.1).] If a function or member function name is used, the name can be overloaded (13), in which case the appropriate function shall be selected according to the rules in 13.3. The function called in a member function call is normally selected according to the static type of the object expression (see 10), but if that function is `virtual` the function actually called will be the final overrider (10.3) of the selected function in the dynamic type of the object expression [*Note:* the type of the object pointed or referred to by the current value of the object expression. Clause 12.7 describes the behavior of virtual function calls when the object-expression refers to an object under construction or destruction.]

2 The type of the function call expression is the return type of the statically chosen function (i.e., ignoring the `virtual` keyword), even if the type of the function actually called is different. This type shall be complete or the type `void`.

3 When a function is called, each parameter (8.3.5) shall be initialized (8.5.3, 12.8, 12.1) with its corresponding argument. Standard (4) and user-defined (12.3) conversions shall be performed. The value of a function call is the value returned by the called function except in a virtual function call if the return type of the final overrider is different from the return type of the statically chosen function, the value returned from the final overrider is converted to the return type of the statically chosen function.

4 [*Note:* a function can change the values of its nonconstant parameters, but these changes cannot affect the values of the arguments except where a parameter is of a non-`const` reference type (8.3.2). Where a parameter is of reference type a temporary object is introduced if needed (7.1.5, 2.9, 2.9.4, 8.3.4, 12.2). In addition, it is possible to modify the values of nonconstant objects through pointer parameters.

5 A function can be declared to accept fewer arguments (by declaring default arguments (8.3.6)) or more arguments (by using the ellipsis, `...` 8.3.5) than the number of parameters in the function definition (8.4).]

6 If no declaration of the called function is accessible from the scope of the call the program is ill-formed. [*Note:* this implies that, except where the ellipsis (`...`) is used, a parameter is available for each argument.]

7 Any argument of type `float` for which there is no parameter is converted to `double` before the call; any of `char`, `short`, or a bit-field type for which there is no parameter are converted to `int` or `unsigned` by integral promotion (4.5). Any argument of enumeration type is converted to `int`, `unsigned`, `long`, or `unsigned long` by integral promotion. An argument of a POD class type `T`, for which no corresponding parameter is declared, is passed in a manner such that the receiving function can obtain its value by an invocation of `va_arg(T)`. If an argument of a non-POD class type is passed, and there is no corresponding parameter, the behavior is undefined.

8 [*Note:* an argument of class type for which a corresponding parameter is declared is passed according to the rules above.]

9 The order of evaluation of arguments is unspecified. All side effects of argument expressions take effect before the function is entered. The order of evaluation of the postfix expression and the argument expression list is unspecified.

10 The function-to-pointer standard conversion (4.3) is suppressed on the postfix expression of a function call.

11 Recursive calls are permitted.

12 A function call is an lvalue if and only if the result type is a reference.

5.2.3 Explicit type conversion (functional notation) [expr.type.conv]

1 A *simple-type-specifier* (7.1.5) followed by a parenthesized *expression-list* constructs a value of the specified type given the expression list. If the expression list specifies a single value, the expression is equivalent (in definedness, and if defined in meaning) to the corresponding cast expression (5.4). If the expression list specifies more than a single value, the type shall be a class with a suitably declared constructor (8.5, 12.1), and the expression `T(x1, x2, ...)` is equivalent in effect to the declaration `T t(x1, x2, ...);` for some invented temporary variable `t`, with the result being the value of `t` as an rvalue.

2 The expression `T()`, where `T` is a simple-type-specifier (7.1.5.2), creates an rvalue of the specified type, whose value is determined by default-initialization (8.5).

5.2.4 Class member access [expr.ref]

1 A postfix expression followed by a dot `.` or an arrow `->`, optionally followed by the keyword `template` (14.10.1), and then followed by an *id-expression*, is a postfix expression. The postfix expression before the dot or arrow is evaluated;[40] the result of that evaluation, together with the *id-expression*, determine the result of the entire postfix expression.

2 For the first option (dot) the type of the first expression (the *object expression*) shall be "class object" (of a complete type). For the second option (arrow) the type of the first expression (the *pointer expression*) shall be "pointer to class object" (of a complete type). The *id-expression* shall name a member of that class, except that an imputed destructor can be explicitly invoked for a scalar type (12.4). If `E1` has the type "pointer to class `X`," then the expression `E1->E2` is converted to the equivalent form `(*(E1)).E2`; the remainder of this subclause will address only the first option (dot)[41].

3 If the *id-expression* is a *qualified-id*, the *nested-name-specifier* of the *qualified-id* can specify a namespace name or a class name. If the *nested-name-specifier* of the *qualified-id* specifies a namespace name, the name is looked up in the context in which the entire *postfix-expression* occurs. If the *nested-name-specifier* of the *qualified-id* specifies a class name, the class name is looked up as a type both in the class of the object expression (or the class pointed to by the pointer expression) and the context in which the entire *postfix-expression* occurs. [*Note:* by the "injection" rules, the name, if any, of each class is also considered a nested class member of that class.] These searches shall yield a single type. [*Note:* the type might be found in either or both contexts.] If the *nested-name-specifier* contains a class *template-id* (14.1), its *template-argument*s are evaluated in the context in which the entire *postfix-expression* occurs.

4 Similarly, if the *id-expression* is a *conversion-function-id*, its *conversion-type-id* shall denote the same type in both the context in which the entire *postfix-expression* occurs and in the context of the class of the object expression (or the class pointed to by the pointer expression).

5 Abbreviating *object-expression.id-expression* as `E1.E2`, then the type and lvalue properties of this expression are determined as follows. In the remainder of this subclause, *cq* represents either `const` or the absence of `const`; *vq* represents either `volatile` or the absence of `volatile`. *cv* represents an arbitrary set of cv-qualifiers, as defined in 3.9.3.

6 If `E2` is declared to have type "reference to `T`", then `E1.E2` is an lvalue; the type of `E1.E2` is `T`. Otherwise, one of the following rules applies.

— If `E2` is a static data member, and the type of `E2` is `T`, then `E1.E2` is an lvalue; the expression designates the named member of the class. The type of `E1.E2` is `T`.

— If `E2` is a (possibly overloaded) static member function, and the type of `E2` is "function of (parameter type list) returning `T`", then `E1.E2` is an lvalue; the expression designates the static member function. The type of `E1.E2` is the same type as that of `E2`, namely "function of (parameter type list) returning `T`".

[40] This evaluation happens even if the result is unnecessary to determine the value of the entire postfix expression, for example if the *id-expression* denotes a static member.
[41] Note that if `E1` has the type "pointer to class `X`", then `(*(E1))` is an lvalue.

— If `E2` is a non-static data member, and the type of `E1` is "*cq1 vq1* `X`", and the type of `E2` is "*cq2 vq2* `T`", the expression designates the named member of the object designated by the first expression. If `E1` is an lvalue, then `E1.E2` is an lvalue. Let the notation *vq12* stand for the "union" of *vq1* and *vq2* ; that is, if *vq1* or *vq2* is `volatile`, then *vq12* is `volatile`. Similarly, let the notation *cq12* stand for the "union" of *cq1* and *cq2*; that is, if *cq1* or *cq2* is `const`, then *cq12* is `const`. If `E2` is declared to be a `mutable` member, then the type of `E1.E2` is "*vq12* `T`". If `E2` is not declared to be a `mutable` member, then the type of `E1.E2` is "*cq12 vq12* `T`".

— If `E2` is a (possibly overloaded) non-static member function, and the type of `E2` is "*cv* function of (parameter type list) returning `T`", then `E1.E2` is *not* an lvalue. The expression designates a member function (of some class `X`). The expression can be used only as the left-hand operand of a member function call (9.4). The member function shall be at least as cv-qualified as `E1`. The type of `E1.E2` is "class `X`'s *cv* member function of (parameter type list) returning `T`".

— If `E2` is a nested type, the expression `E1.E2` is ill-formed.

— If `E2` is a member enumerator, and the type of `E2` is `T`, the expression `E1.E2` is not an lvalue. The type of `E1.E2` is `T`.

7 [*Note:* "class objects" can be structures (9.2) and unions (9.6). Classes are discussed in clause 9.]

5.2.5 Increment and decrement [expr.post.incr]

1 The value obtained by applying a postfix `++` is the value that the operand had before applying the operator. [*Note:* the value obtained is a copy of the original value] The operand shall be a modifiable lvalue. The type of the operand shall be an arithmetic type or a pointer to object type. After the result is noted, the value of the object is modified by adding `1` to it, unless the object is of type `bool`, in which case it is set to `true`. [*Note:* this use is deprecated.] The type of the result is the same as the type of the operand, but it is not an lvalue. See also 5.7 and 5.17.

2 The operand of postfix `--` is decremented analogously to the postfix `++` operator, except that the operand shall not be of type `bool`.

5.2.6 Dynamic cast [expr.dynamic.cast]

1 The result of the expression `dynamic_cast<T>(v)` is the result of converting the expression `v` to type `T`. `T` shall be a pointer or reference to a complete class type, or "pointer to *cv* `void`". Types shall not be defined in a `dynamic_cast`. The `dynamic_cast` operator shall not cast away constness (5.2.10).

2 If `T` is a pointer type, `v` shall be an rvalue of a pointer to complete class type, and the result is an rvalue of type `T`. If `T` is a reference type, `v` shall be an lvalue of a complete class type, and the result is an lvalue of the type referred to by `T`.

3 If the type of `v` is the same as the required result type (which, for convenience, will be called `R` in this description), or it can be converted to `R` via a qualification conversion (4.4) in the pointer case, the result is `v` (converted if necessary).

4 If the value of `v` is a null pointer value in the pointer case, the result is the null pointer value of type `R`.

5 If `T` is "pointer to *cv1* `B`" and `v` has type "pointer to *cv2* `D`" such that `B` is a base class of `D`, the result is a pointer to the unique `B` sub-object of the `D` object pointed to by `v`. Similarly, if `T` is "reference to *cv1* `B`" and `v` has type "*cv2* `D`" such that `B` is a base class of `D`, the result is an lvalue for the unique[42)] `B` sub-object of the `D` object referred to by `v`. In both the pointer and reference cases, *cv1* shall be the same cv-qualification as, or greater cv-qualification than, *cv2*, and `B` shall be an accessible nonambiguous base class of `D`. [*Example:*

[42)] The complete object pointed or referred to by `v` can contain other `B` objects as base classes, but these are ignored.

```
struct B {};
struct D : B {};
void foo(D* dp)
{
    B*  bp = dynamic_cast<B*>(dp);  // equivalent to B* bp = dp;
}
```

—*end example*]

6 Otherwise, `v` shall be a pointer to or an lvalue of a polymorphic type (10.3).

7 If `T` is "pointer to *cv* `void`," then the result is a pointer to the complete object (12.6.2) pointed to by `v`. Otherwise, a run-time check is applied to see if the object pointed or referred to by `v` can be converted to the type pointed or referred to by `T`.

8 The run-time check logically executes like this: If, in the complete object pointed (referred) to by `v`, `v` points (refers) to a `public` base class sub-object of a `T` object, and if only one object of type `T` is derived from the sub-object referred to by `v`, the result is a pointer (an lvalue referring) to that `T` object. Otherwise, if the type of the complete object has an unambiguous public base class of type `T`, the result is a pointer (reference) to the `T` sub-object of the complete object. Otherwise, the run-time check *fails*.

9 The value of a failed cast to pointer type is the null pointer value of the required result type. A failed cast to reference type throws `bad_cast` (18.5.2). [*Example:*

```
class A { virtual void f(); };
class B { virtual void g(); };
class D : public virtual A, private B {};
void g()
{
    D   d;
    B*  bp = (B*)&d;  // cast needed to break protection
    A*  ap = &d;      // public derivation, no cast needed
    D&  dr = dynamic_cast<D&>(*bp);  // succeeds
    ap = dynamic_cast<A*>(bp);       // succeeds
    bp = dynamic_cast<B*>(ap);       // fails
    ap = dynamic_cast<A*>(&dr);      // succeeds
    bp = dynamic_cast<B*>(&dr);      // fails
}

class E : public D , public B {};
class F : public E, public D {}
void h()
{
    F   f;
    A*  ap = &f;  // okay: finds unique A
    D*  dp = dynamic_cast<D*>(ap);  // fails: ambiguous
    E*  ep = (E*)ap;  // error: cast from virtual base
    E*  ep = dynamic_cast<E*>(ap);  // succeeds
}
```

—*end example*] [*Note:* Clause 12.7 describes the behavior of a `dynamic_cast` applied to an object under construction or destruction.]

5.2.7 Type identification [expr.typeid]

1 The result of a `typeid` expression is of type `const type_info&`. The value is a reference to a `type_info` object (18.5.1) that represents the *type-id* or the type of the *expression* respectively.

2 If the *expression* is a reference to a polymorphic type (10.3), the `type_info` for the complete object (12.6.2) referred to is the result.

3 If the *expression* is the result of applying unary `*` to a pointer to a polymorphic type,[43] then the pointer shall either be zero or point to a valid object. If the pointer is zero, the `typeid` expression throws the `bad_typeid` exception (18.5.3). Otherwise, the result of the `typeid` expression is the value that represents the type of the complete object to which the pointer points.

4 If the *expression* is the result of subscripting (5.2.1) a pointer, say `p`, that points to a polymorphic type,[44] then the result of the `typeid` expression is that of `typeid(*p)`. The subscript is not evaluated.

5 If the expression is neither a pointer nor a reference to a polymorphic type, the result is the `type_info` representing the (static) type of the *expression*. The *expression* is not evaluated.

6 In all cases `typeid` ignores the top-level cv-qualifiers of its operand's type. [*Example:*

```
class D { ... };
D d1;
const D d2;

typeid(d1) == typeid(d2);      // yields true
typeid(D)  == typeid(const D); // yields true
typeid(D)  == typeid(d2);      // yields true
```

—*end example*] [*Note:* Clause 12.7 describes the behavior of `typeid` applied to an object under construction or destruction.]

5.2.8 Static cast [expr.static.cast]

1 The result of the expression `static_cast<T>(v)` is the result of converting the expression `v` to type `T`. If `T` is a reference type, the result is an lvalue; otherwise, the result is an rvalue. Types shall not be defined in a `static_cast`. The `static_cast` operator shall not cast away constness. See 5.2.10.

2 Any implicit conversion (including standard conversions and/or user-defined conversions; see 4 and 13.3.3.1) can be performed explicitly using `static_cast`. More precisely, if `T t(v);` is a well-formed declaration, for some invented temporary variable `t`, then the result of `static_cast<T>(v)` is defined to be the temporary `t`, and is an lvalue if `T` is a reference type, and an rvalue otherwise. The expression `v` shall be an lvalue if the equivalent declaration requires an lvalue for `v`.

3 If the `static_cast` does not correspond to an implicit conversion by the above definition, it shall perform one of the conversions listed below. No other conversion can be performed explicitly using a `static_cast`.

4 Any expression can be explicitly converted to type "*cv* `void`." The expression value is discarded.

5 An lvalue expression of type `T1` can be cast to the type "reference to `T2`" if an expression of type "pointer to `T1`" can be explicitly converted to the type "pointer to `T2`" using a `static_cast`. That is, a reference cast `static_cast<T&>x` has the same effect as the conversion `*static_cast<T*>&x` with the built-in `&` and `*` operators. The result is an lvalue. This interpretation is used only if the original `static_cast` is not well-formed as an implicit conversion under the rules given above. This form of reference cast creates an lvalue that refers to the same object as the source lvalue, but with a different type. [*Note:* it does not create a temporary or copy the object, and constructors (12.1) or conversion functions (12.3) are not called. For example,

```
struct B {};
struct D : public B {};
D d;
// creating a temporary for the B sub-object not allowed
... (const B&) d ...
```

—*end note*]

[43] If `p` is a pointer, then `*p`, `(*p)`, `((*p))`, and so on all meet this requirement.
[44] If `p` is a pointer to a polymorphic type and `i` has integral or enumerated type, then `p[i]`, `(p[i])`, `(p)[i]`, `((((p))[((i))]))`, `i[p]`, `(i[p])`, and so on all meet this requirement.

6 The inverse of any standard conversion (4), other than the lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) conversions, can be performed explicitly using `static_cast` subject to the restriction that the explicit conversion does not cast away constness (5.2.10), and the following additional rules for specific cases:

7 A value of integral type can be explicitly converted to an enumeration type. The value is unchanged if the integral value is within the range of the enumeration values (7.2). Otherwise, the resulting enumeration value is unspecified.

8 An rvalue of type “pointer to *cv1* `B`”, where `B` is a class type, can be converted to an rvalue of type “pointer to *cv2* `D`”, where `D` is a class derived (10) from `B`, if a valid standard conversion from “pointer to *cv2* `D`” to “pointer to *cv2* `B`” exists (4.10), *cv2* is the same cv-qualification as, or greater cv-qualification than, *cv1*, and `B` is not a virtual base class of `D`. The null pointer value (4.10) is converted to the null pointer value of the destination type. If the rvalue of type “pointer to *cv1* `B`” points to a `B` that is actually a sub-object of an object of type `D`, the resulting pointer points to the enclosing object of type `D`. Otherwise, the result of the cast is undefined.

9 An rvalue of type “pointer to member of `D` of type *cv1* `T`” can be converted to an rvalue of type “pointer to member of `B` of type *cv2* `T`”, where `B` is a base class (10) of `D`, if a valid standard conversion from “pointer to member of `B` of type *cv2* `T`” to “pointer to member of `D` of type *cv2* `T`” exists (4.11), and *cv2* is the same cv-qualification as, or greater cv-qualification than, *cv1*. The null member pointer value (4.11) is converted to the null member pointer value of the destination type. If class `B` contains or inherits the original member, the resulting pointer to member points to the member in class `B`. Otherwise, the result of the cast is undefined.

5.2.9 Reinterpret cast [expr.reinterpret.cast]

1 The result of the expression `reinterpret_cast<T>(v)` is the result of converting the expression `v` to type `T`. If `T` is a reference type, the result is an lvalue; otherwise, the result is an rvalue. Types shall not be defined in a `reinterpret_cast`. Conversions that can be performed explicitly using `reinterpret_cast` are listed below. No other conversion can be performed explicitly using `reinterpret_cast`.

2 The `reinterpret_cast` operator shall not cast away constness; [*Note:* see 5.2.10 for the definition of ‘‘casting away constness’’.]

3 The mapping performed by `reinterpret_cast` is implementation-defined. [*Note:* it might, or might not, produce a representation different from the original value.]

4 A pointer can be explicitly converted to any integral type large enough to hold it. The mapping function is implementation-defined [*Note:* it is intended to be unsurprising to those who know the addressing structure of the underlying machine.]

5 A value of integral type can be explicitly converted to a pointer. A pointer converted to an integer of sufficient size (if any such exists on the implementation) and back to the same pointer type will have its original value; mappings between pointers and integers are otherwise implementation-defined.

6 The operand of a pointer cast can be an rvalue of type “pointer to incomplete class type”. The destination type of a pointer cast can be “pointer to incomplete class type”. In such cases, if there is any inheritance relationship between the source and destination classes, the behavior is undefined.

7 A pointer to a function can be explicitly converted to a pointer to a function of a different type. The effect of calling a function through a pointer to a function type that differs from the type used in the definition of the function is undefined. Except that converting an rvalue of type “pointer to `T1`” to the type “pointer to `T2`” (where `T1` and `T2` are function types) and back to its original type yields the original pointer value, the result of such a pointer conversion is unspecified; [*Note:* see also 4.10 for more details of pointer conversions.]

8 A pointer to an object can be explicitly converted to a pointer to an object of different type. Except that converting an rvalue of type “pointer to T1” to the type “pointer to T2” (where T1 and T2 are object types and where the alignment requirements of T2 are no stricter than those of T1) and back to its original type yields the original pointer value, the result of such a pointer conversion is unspecified;

9 The null pointer value (4.10) is converted to the null pointer value of the destination type.

10 An rvalue of type “pointer to member of X of type T1”, can be explicitly converted to an rvalue of type “pointer to member of Y of type T2”, if T1 and T2 are both function types or both data member types. The null member pointer value (4.11) is converted to the null member pointer value of the destination type. The result of this conversion is unspecified, except in the following cases:

— converting an rvalue of type “pointer to member function” to a different pointer to member function type and back to its original type yields the original pointer to member value.

— converting an rvalue of type “pointer to data member of X of type T1” to the type “pointer to data member of Y of type T2” (where the alignment requirements of T2 are no stricter than those of T1) and back to its original type yields the original pointer to member value.

11 Calling a member function through a pointer to member that represents a function type that differs from the function type specified on the member function declaration results in undefined behavior.

12 An lvalue expression of type T1 can be cast to the type “reference to T2” if an expression of type “pointer to T1” can be explicitly converted to the type “pointer to T2” using a reinterpret_cast. That is, a reference cast reinterpret_cast<T&>x has the same effect as the conversion *reinterpret_cast<T*>&x with the built-in & and * operators. The result is an lvalue that refers to the same object as the source lvalue, but with a different type. No temporary is created, no copy is made, and constructors (12.1) or conversion functions (12.3) are not called.

5.2.10 Const cast [expr.const.cast]

1 The result of the expression const_cast<T>(v) is of type “T.” Types shall not be defined in a const_cast. Conversions that can be performed explicitly using const_cast are listed below. No other conversion shall be performed explicitly using const_cast.

2 An rvalue of type “pointer to *cv1* T” can be explicitly converted to the type “pointer to *cv2* T”, where T is any object type and where *cv1* and *cv2* are cv-qualifications, using the cast const_cast<*cv2* T*>. An lvalue of type *cv1* T can be explicitly converted to an lvalue of type *cv2* T, where T is any object type and where *cv1* and *cv2* are cv-qualifications, using the cast const_cast<*cv2* T&>. The result of a pointer or reference const_cast refers to the original object.

3 An rvalue of type “pointer to member of X of type *cv1* T” can be explicitly converted to the type “pointer to member of X of type *cv2* T”, where T is a data member type and where *cv1* and *cv2* are cv-qualifiers, using the cast const_cast<*cv2* T X::*>. The result of a pointer to member const_cast will refer to the same member as the original (uncast) pointer to data member.

4 The following rules define casting away constness. In these rules T*n* and X*n* represent types. For two pointer types:

$$\mathtt{X1} \text{ is } \mathtt{T1} cv_{1,1} * \cdots cv_{1,N} * \quad \text{where T1 is not a pointer type}$$

$$\mathtt{X2} \text{ is } \mathtt{T2} cv_{2,1} * \cdots cv_{2,N} * \quad \text{where T2 is not a pointer type}$$

$$K \text{ is } min(N,M)$$

casting from X1 to X2 casts away constness if, for a non-pointer type T (e.g., int), there does not exist an implicit conversion from:

$$Tcv_{1,(N-K+1)} * cv_{1,(N-K+2)} * \cdots cv_{1,N} *$$

to

$$Tcv_{2,(N-K+1)} * cv_{2,(M-K+2)} * \cdots cv_{2,M} *$$

5 Casting from an lvalue of type `T1` to an lvalue of type `T2` using a reference cast casts away constness if a cast from an rvalue of type “pointer to `T1`” to the type “pointer to `T2`” casts away constness.

6 Casting from an rvalue of type "pointer to data member of `X` of type “`T1`” to the type “pointer to data member of `Y` of type `T2`” casts away constness if a cast from an rvalue of type “pointer to `T1`” to the type “pointer to `T2`” casts away constness.

7 [*Note:* these rules are not intended to protect constness in all cases. For instance, conversions between pointers to functions are not covered because such conversions lead to values whose use causes undefined behavior. For the same reasons, conversions between pointers to member functions, and in particular, the conversion from a pointer to a const member function to a pointer to a non-const member function, are not covered. For multi-level pointers to data members, or multi-level mixed object and member pointers, the same rules apply as for multi-level object pointers. That is, the “member of” attribute is ignored for purposes of determining whether `const` has been cast away.

8 Depending on the type of the object, a write operation through the pointer, lvalue or pointer to data member resulting from a `const_cast` that casts away constness may produce undefined behavior (7.1.5.1).]

9 A null pointer value (4.10) is converted to the null pointer value of the destination type. The null member pointer value (4.11) is converted to the null member pointer value of the destination type.

5.3 Unary expressions [expr.unary]

1 Expressions with unary operators group right-to-left.

```
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
        *  &  +  -  !  ~
```

5.3.1 Unary operators [expr.unary.op]

1 The unary `*` operator means *indirection*: the expression shall be a pointer, and the result is an lvalue referring to the object or function to which the expression points. If the type of the expression is “pointer to `T`,” the type of the result is “`T`.”

2 The result of the unary `&` operator is a pointer to its operand. The operand shall be an lvalue or a *qualified-id*. In the first case, if the type of the expression is “`T`,” the type of the result is “pointer to `T`.” In particular, the address of an object of type “*cv* `T`” is “pointer to *cv* `T`,” with the same cv-qualifiers. [*Example:* the address of an object of type “`const int`” has type “pointer to `const int`.”] For a *qualified-id*, if the member is a nonstatic member of class `C` of type `T`, the type of the result is “pointer to member of `class C` of type `T`.” [*Example:*

```
struct A { int i; };
struct B : A { };
... &B::i ... // has type "int A::*"
```

—*end example*] For a static member of type “`T`”, the type is plain “pointer to `T`.” [*Note:* a pointer to member is only formed when an explicit `&` is used and its operand is a *qualified-id* not enclosed in parentheses. [*Example:* the expression `&(qualified-id)`, where the *qualified-id* is enclosed in parentheses, does not form an expression of type “pointer to member.”] Neither does `qualified-id`, because there is no implicit conversion from the type "nonstatic member function" to the type "pointer to member function", as there is from an lvalue of function type to the type "pointer to function" (4.3). Nor is `&unqualified-id` a pointer to member, even within the scope of *unqualified-id*'s class.]

3 The address of an object of incomplete type can be taken, but if the complete type of that object has the address-of operator (`operator&()`) overloaded, then the behavior is undefined (and no diagnostic is required).

4 The address of an overloaded function (13) can be taken only in a context that uniquely determines which version of the overloaded function is referred to (see 13.4). [*Note:* since the context might determine whether the operand is a static or nonstatic member function, the context can also affect whether the expression has type "pointer to function" or "pointer to member function."]

5 The operand of the unary `+` operator shall have arithmetic, enumeration, or pointer type and the result is the value of the argument. Integral promotion is performed on integral or enumeration operands. The type of the result is the type of the promoted operand.

6 The operand of the unary `-` operator shall have arithmetic or enumeration type and the result is the negation of its operand. Integral promotion is performed on integral or enumeration operands. The negative of an unsigned quantity is computed by subtracting its value from 2^n, where n is the number of bits in the promoted operand. The type of the result is the type of the promoted operand.

7 The operand of the logical negation operator `!` is converted to `bool` (4.13); its value is `true` if the converted operand is `false` and `false` otherwise. The type of the result is `bool`.

8 The operand of `~` shall have integral or enumeration type; the result is the one's complement of its operand. Integral promotions are performed. The type of the result is the type of the promoted operand.

5.3.2 Increment and decrement [expr.pre.incr]

1 The operand of prefix `++` is modified by adding `1`, or set to `true` if it is `bool` (this use is deprecated). The operand shall be a modifiable lvalue. The type of the operand shall be an arithmetic type or a pointer to a completely-defined object type. The value is the new value of the operand; it is an lvalue. If `x` is not of type `bool`, the expression `++x` is equivalent to `x+=1`. [*Note:* see the discussions of addition (5.7) and assignment operators (5.17) for information on conversions.]

2 The operand of prefix `--` is decremented analogously to the prefix `++` operator, except that the operand shall not be of type `bool`.

5.3.3 Sizeof [expr.sizeof]

1 The `sizeof` operator yields the number of bytes in the object representation of its operand. The operand is either an expression, which is not evaluated, or a parenthesized *type-id*. The `sizeof` operator shall not be applied to an expression that has function or incomplete type, or to an enumeration type before all its enumerators have been declared, or to the parenthesized name of such types, or to an lvalue that designates a bit-field. [*Note:* `sizeof(char)` is `1`, but `sizeof(bool)` and `sizeof(wchar_t)` are implementation-defined. [45] See 1.5 for the definition of *byte* and 3.9 for the definition of *object representation.*]

2 When applied to a reference, the result is the size of the referenced object. When applied to a class, the result is the number of bytes in an object of that class including any padding required for placing such objects in an array. The size of any class or class object is greater than zero. When applied to an array, the result is the total number of bytes in the array. This implies that the size of an array of *n* elements is *n* times the size of an element.

[45] `sizeof(bool)` is not required to be `1`.

3 The `sizeof` operator can be applied to a pointer to a function, but shall not be applied directly to a function.

4 The lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) standard conversions are suppressed on the operand of `sizeof`.

5 Types shall not be defined in a `sizeof` expression.

6 The result is a constant of an implementation-defined type which is the same type as that which is named `size_t` in the standard header `<cstddef>`(18.1).

5.3.4 New [expr.new]

1 The *new-expression* attempts to create an object of the *type-id* (8.1) to which it is applied. This type shall be a complete nonabstract object type or array type (1.6, 3.9, 10.4).

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*$_{opt}$ `)`

Entities created by a *new-expression* have dynamic storage duration (3.7.3). [*Note:* the lifetime of such an entity is not necessarily restricted to the scope in which it is created.] If the entity is an object, the *new-expression* returns a pointer to the object created. If it is an array, the *new-expression* returns a pointer to the initial element of the array.

2 The *new-type* in a *new-expression* is the longest possible sequence of *new-declarator*s. This prevents ambiguities between declarator operators `&`, `*`, `[]`, and their expression counterparts. [*Example:*

```
new int*i;     // syntax error: parsed as ‘(new int*) i’
               //               not as ‘(new int)*i’
```

The `*` is the pointer declarator and not the multiplication operator.]

3 Parentheses shall not appear in a *new-type-id* used as the operand for `new`.

4 [*Example:*

```
new int(*[10])();       // error
```

is ill-formed because the binding is

```
(new int) (*[10])();    // error
```

Instead, the explicitly parenthesized version of the `new` operator can be used to create objects of compound types (3.9.2):

```
        new (int (*[10])());
```

allocates an array of `10` pointers to functions (taking no argument and returning `int`).]

5 The *type-specifier-seq* shall not contain class declarations, or enumeration declarations.

6 When the allocated object is an array (that is, the *direct-new-declarator* syntax is used or the *new-type-id* or *type-id* denotes an array type), the *new-expression* yields a pointer to the initial element (if any) of the array. [*Note:* both `new int` and `new int[10]` return an `int*` and the type of `new int[i][10]` is `int (*)[10]`.]

7 Every *constant-expression* in a *direct-new-declarator* shall be an integral constant expression (5.19) with a strictly positive value. The *expression* in a *direct-new-declarator* shall be of integral type (3.9.1) with a non-negative value. [*Example:* if `n` is a variable of type `int`, then `new float[n][5]` is well-formed (because `n` is the *expression* of a *direct-new-declarator*), but `new float[5][n]` is ill-formed (because `n` is not a *constant-expression*). If `n` is negative, the effect of `new float[n][5]` is undefined.]

8 When the value of the *expression* in a *direct-new-declarator* is zero, an array with no elements is allocated. The pointer returned by the *new-expression* is non-null and distinct from the pointer to any other object.

9 Storage for the object created by a *new-expression* is obtained from the appropriate *allocation function* (3.7.3.1). When the allocation function is called, the first argument will be amount of space requested (which shall be no larger than the size of the object being created unless that object is an array).

10 An implementation shall provide default definitions of the global allocation functions `operator new()` for non-arrays (3.7.3, 18.4.1.1) and `operator new[]()` for arrays (18.4.1.2). [*Note:* A C++ program can provide alternative definitions of these functions (17.3.3.4), and/or class-specific versions (12.5).]

11 The *new-placement* syntax can be used to supply additional arguments to an allocation function. If used, overloading resolution is done by assembling an argument list from the amount of space requested (the first argument) and the expressions in the *new-placement* part of the *new-expression* (the second and succeeding arguments).

12 [*Example:*

— `new T` results in a call of `operator new(sizeof(T))`,

— `new(2,f) T` results in a call of `operator new(sizeof(T),2,f)`,

— `new T[5]` results in a call of `operator new[](sizeof(T)*5+x)`, and

— `new(2,f) T[5]` results in a call of `operator new[](sizeof(T)*5+y,2,f)`. Here, `x` and `y` are non-negative, implementation-defined values representing array allocation overhead. They might vary from one use of `new` to another.]

13 The allocation function shall either return null or a pointer to a block of storage in which the object shall be created. [*Note:* the block of storage is assumed to be appropriately aligned and of the requested size. The address of the created object will not necessarily be the same as that of the block if the object is an array.]

14 If the type of the object created by the *new-expression* is `T`:

— If the *new-initializer* is omitted and `T` is a non-POD class type (or array thereof), then if the default constructor for `T` is accessible it is called, otherwise the program is ill-formed;

— If the *new-initializer* is omitted and `T` is a POD type (or array thereof), then the object thus created has indeterminate value;

— If the *new-initializer* is of the form `()`, default-initialization shall be performed (8.5);

— If the *new-initializer* is of the form `(` *expression-list*`)` and `T` is a class type, the appropriate constructor is called, using *expression-list* as the arguments (8.5);

— If the *new-initializer* is of the form `(` *expression-list*`)` and `T` is an arithmetic, enumeration, pointer, or pointer-to-member type and *expression-list* comprises exactly one expression, then the object is initialized to the (possibly converted) value of the expression (8.5);

— Otherwise the *new-expression* is ill-formed.

15 Access and ambiguity control are done for both the allocation function and the constructor (12.1, 12.5).

16 The allocation function can indicate failure by throwing a `bad_alloc` exception (15, 18.4.2.1). In this case no initialization is done.

17 If the constructor throws an exception and the *new-expression* does not contain a *new-placement*, then the deallocation function (3.7.3.2, 12.5) is used to free the memory in which the object was being constructed, after which the exception continues to propagate in the context of the *new-expression*.

18 If the constructor throws an exception and the *new-expression* contains a *new-placement*, a name lookup is performed on the name of operator delete in the scope of this *new-expression*. If the lookup succeeds and exactly one of the declarations found matches the declaration of that placement operator new, then the matching placement operator delete shall be called (3.7.3.2).

19 A declaration of placement operator delete matches the declaration of a placement operator new when it has the same number of parameters and all parameter types except the first are identical disregarding top-level *cv-qualifiers*.

20 If placement operator delete is called, it is passed the same arguments as were passed to placement operator new. If the implementation is allowed to make a copy of an argument as part of the placement new call, it is allowed to make a copy (of the same original value) as part of the placement delete call, or to reuse the copy made as part of the placement new call. If the copy is elided in one place, it need not be elided in the other.

21 The way the object was allocated determines how it is freed: if it is allocated by `::new`, then it is freed by `::delete`, and if it is an array, it is freed by `delete[]` or `::delete[]` as appropriate.

22 Whether the allocation function is called before evaluating the constructor arguments or after evaluating the constructor arguments but before entering the constructor is unspecified. It is also unspecified whether the arguments to a constructor are evaluated if the allocation function returns the null pointer or throws an exception.

5.3.5 Delete [expr.delete]

1 The *delete-expression* operator destroys a complete object (1.6) or array created by a *new-expression*.

delete-expression:
  `::`$_{opt}$ `delete` *cast-expression*
  `::`$_{opt}$ `delete [ ]` *cast-expression*

The first alternative is for non-array objects, and the second is for arrays. The operand shall have a pointer type. The result has type `void`.

2 In either alternative, if the value of the operand of `delete` is the null pointer the operation has no effect. Otherwise, in the first alternative (*delete object*), the value of the operand of `delete` shall be a pointer to a non-array object created by a *new-expression* without a *new-placement* specification, or a pointer to a sub-object (1.6) representing a base class of such an object (10), or an expression of class type with a conversion function to pointer type (_class.conv,fct_) which yields a pointer to such an object. If not, the behavior is undefined. In the second alternative (*delete array*), the value of the operand of `delete` shall be a pointer to an array created by a *new-expression* without a *new-placement* specification. If not, the behavior is undefined.

3 In the first alternative (*delete object*), if the static type of the operand is different from its dynamic type, the static type shall have a virtual destructor or the behavior is undefined. In the second alternative (*delete array*) if the dynamic type of the object to be deleted differs from its static type, the behavior is undefined.[46)]

[46)] This implies that an object cannot be deleted using a point of type `void*` because there are no objects of type `void`.

4 It is unspecified whether the deletion of an object changes its value. If the expression denoting the object in a *delete-expression* is a modifiable lvalue, any attempt to access its value after the deletion is undefined (3.7.3.2).

5 If the object being deleted has incomplete class type at the point of deletion and the class has a non-trivial destructor or an allocation function or a deallocation function, the behavior is undefined.

6 The *delete-expression* will invoke the destructor (if any) for the object or the elements of the array being deleted. In the case of an array, the elements will be destroyed in order of decreasing address (that is, in reverse order of construction; see 12.6.2).

7 To free the storage pointed to, the *delete-expression* will call a *deallocation function* (3.7.3.2).

8 An implementation provides default definitions of the global deallocation functions `operator delete()` for non-arrays (18.4.1.1) and `operator delete[]()` for arrays (18.4.1.2). A C++ program can provide alternative definitions of these functions (17.3.3.4), and/or class-specific versions (12.5).

9 Access and ambiguity control are done for both the deallocation function and the destructor (12.4, 12.5).

5.4 Explicit type conversion (cast notation) [expr.cast]

1 The result of the expression `(T)` *cast-expression* is of type `T`. An explicit type conversion can be expressed using functional notation (5.2.3), a type conversion operator (`dynamic_cast, static_cast, reinterpret_cast, const_cast`), or the *cast* notation.

cast-expression:
    unary-expression
    `(` *type-id* `)` *cast-expression*

2 Types shall not be defined in casts.

3 Any type conversion not mentioned below and not explicitly defined by the user (12.3) is ill-formed.

4 The conversions performed by `static_cast` (5.2.8), `reinterpret_cast` (5.2.9), `const_cast` (5.2.10), or any sequence thereof, can be performed using the cast notation of explicit type conversion. The same semantic restrictions and behaviors apply. If a given conversion can be performed using either `static_cast` or `reinterpret_cast`, the `static_cast` interpretation is used.

5 In addition to those conversions, a pointer to an object of a derived class (10) can be explicitly converted to a pointer to any of its base classes regardless of accessibility restrictions (11.2), provided the conversion is unambiguous (10.2). The resulting pointer will refer to the contained object of the base class.

5.5 Pointer-to-member operators [expr.mptr.oper]

1 The pointer-to-member operators `->*` and `.*` group left-to-right.

pm-expression:
    cast-expression
    pm-expression `.*` *cast-expression*
    pm-expression `->*` *cast-expression*

2 The binary operator `.*` binds its second operand, which shall be of type “pointer to member of `T`” to its first operand, which shall be of class `T` or of a class of which `T` is an unambiguous and accessible base class. The result is an object or a function of the type specified by the second operand.

3 The binary operator `->*` binds its second operand, which shall be of type “pointer to member of `T`” to its first operand, which shall be of type “pointer to `T`” or “pointer to a class of which `T` is an unambiguous and accessible base class.” The result is an object or a function of the type specified by the second operand.

4 The restrictions on *cv*-qualification, and the manner in which the *cv*-qualifiers of the operands are combined to produce the *cv*-qualifiers of the result, are the same as the rules for `E1.E2` given in [expr.ref].

5 If the result of `.*` or `->*` is a function, then that result can be used only as the operand for the function call operator `()`. [*Example:*

```
(ptr_to_obj->*ptr_to_mfct)(10);
```

calls the member function denoted by `ptr_to_mfct` for the object pointed to by `ptr_to_obj`.] The result of a `.*` expression is an lvalue only if its first operand is an lvalue and its second operand is a pointer to data member. The result of an `->*` expression is an lvalue only if its second operand is a pointer to data member. If the second operand is the null pointer to member value (4.11), the behavior is undefined.

5.6 Multiplicative operators [expr.mul]

1 The multiplicative operators `*`, `/`, and `%` group left-to-right.

multiplicative-expression:
  pm-expression
  multiplicative-expression `*` *pm-expression*
  multiplicative-expression `/` *pm-expression*
  multiplicative-expression `%` *pm-expression*

2 The operands of `*` and `/` shall have arithmetic type; the operands of `%` shall have integral type. The usual arithmetic conversions are performed on the operands and determine the type of the result.

3 The binary `*` operator indicates multiplication.

4 The binary `/` operator yields the quotient, and the binary `%` operator yields the remainder from the division of the first expression by the second. If the second operand of `/` or `%` is zero the behavior is undefined; otherwise `(a/b)*b + a%b` is equal to `a`. If both operands are nonnegative then the remainder is nonnegative; if not, the sign of the remainder is implementation-defined.

5.7 Additive operators [expr.add]

1 The additive operators `+` and `-` group left-to-right. The usual arithmetic conversions are performed for operands of arithmetic type.

additive-expression:
  multiplicative-expression
  additive-expression `+` *multiplicative-expression*
  additive-expression `-` *multiplicative-expression*

For addition, either both operands shall have arithmetic type, or one operand shall be a pointer to a completely defined object type and the other shall have integral type.

2 For subtraction, one of the following shall hold:

— both operands have arithmetic type;

— both operands are pointers to cv-qualified or cv-unqualified versions of the same completely defined object type; or

— the left operand is a pointer to a completely defined object type and the right operand has integral type.

3 If both operands have arithmetic type, the usual arithmetic conversions are performed on them. The result of the binary `+` operator is the sum of the operands. The result of the binary `-` operator is the difference resulting from the subtraction of the second operand from the first.

4 For the purposes of these operators, a pointer to a nonarray object behaves the same as a pointer to the first element of an array of length one with the type of the object as its element type.

5 When an expression that has integral type is added to or subtracted from a pointer, the result has the type of the pointer operand. If the pointer operand points to an element of an array object, and the array is large enough, the result points to an element offset from the original element such that the difference of the subscripts of the resulting and original array elements equals the integral expression. In other words, if the expression `P` points to the *i*-th element of an array object, the expressions `(P)+N` (equivalently, `N+(P)`) and `(P)-N` (where `N` has the value *n*) point to, respectively, the *i*+*n*-th and *i*−*n*-th elements of the array object, provided they exist. Moreover, if the expression `P` points to the last element of an array object, the expression `(P)+1` points one past the last element of the array object, and if the expression `Q` points one past the last element of an array object, the expression `(Q)-1` points to the last element of the array object. If both the pointer operand and the result point to elements of the same array object, or one past the last element of the array object, the evaluation shall not produce an overflow; otherwise, the behavior is undefined. If the result is used as an operand of the unary `*` operator, the behavior is undefined unless both the pointer operand and the result point to elements of the same array object, or the pointer operand points one past the last element of an array object and the result points to an element of the same array object.

6 When two pointers to elements of the same array object are subtracted, the result is the difference of the subscripts of the two array elements. The type of the result is an implementation-defined signed integral type; this type shall be the same type that is defined as `ptrdiff_t` in the `<cstddef>` header (18.1). As with any other arithmetic overflow, if the result does not fit in the space provided, the behavior is undefined. In other words, if the expressions `P` and `Q` point to, respectively, the *i*-th and *j*-th elements of an array object, the expression `(P)-(Q)` has the value *i*−*j* provided the value fits in an object of type `ptrdiff_t`. Moreover, if the expression `P` points either to an element of an array object or one past the last element of an array object, and the expression `Q` points to the last element of the same array object, the expression `((Q)+1)-(P)` has the same value as `((Q)-(P))+1` and as `-((P)-((Q)+1))`, and has the value zero if the expression `P` points one past the last element of the array object, even though the expression `(Q)+1` does not point to an element of the array object. Unless both pointers point to elements of the same array object, or one past the last element of the array object, the behavior is undefined.[47)]

5.8 Shift operators [expr.shift]

1 The shift operators `<<` and `>>` group left-to-right.

shift-expression:
  additive-expression
  shift-expression `<<` *additive-expression*
  shift-expression `>>` *additive-expression*

The operands shall be of integral type and integral promotions are performed. The type of the result is that of the promoted left operand. The behavior is undefined if the right operand is negative, or greater than or equal to the length in bits of the promoted left operand. The value of `E1 << E2` is `E1` (interpreted as a bit pattern) left-shifted `E2` bits; vacated bits are zero-filled. The value of `E1 >> E2` is `E1` right-shifted `E2` bit positions. If `E1` has an unsigned type or has a nonnegative value, the vacated bits shall be zero-filled. If `E1` has a negative value, the behavior of the right shift is implementation-defined.

5.9 Relational operators [expr.rel]

1 [*Note:* the relational operators group left-to-right, but this fact is not very useful; `a<b<c` means `(a<b)<c` and *not* `(a<b)&&(b<c)`. *—end note*]

47) Another way to approach pointer arithmetic is first to convert the pointer(s) to character pointer(s): In this scheme the integral expression added to or subtracted from the converted pointer is first multiplied by the size of the object originally pointed to, and the resulting pointer is converted back to the original type. For pointer subtraction, the result of the difference between the character pointers is similarly divided by the size of the object originally pointed to.

7 When viewed in this way, an implementation need only provide one extra byte (which might overlap another object in the program) just after the end of the object in order to satisfy the “one past the last element” requirements.

relational-expression:
  shift-expression
  relational-expression `<` *shift-expression*
  relational-expression `>` *shift-expression*
  relational-expression `<=` *shift-expression*
  relational-expression `>=` *shift-expression*

The operands shall have arithmetic or pointer type. The operators `<` (less than), `>` (greater than), `<=` (less than or equal to), and `>=` (greater than or equal to) all yield `false` or `true`. The type of the result is `bool`.

2 The usual arithmetic conversions are performed on arithmetic operands. Pointer conversions are performed on pointer operands to bring them to the same type, which shall be a cv-qualified or cv-unqualified version of the type of one of the operands. [*Note:* this implies that any pointer can be compared to an integral constant expression evaluating to zero and any pointer can be compared to a pointer of cv-qualified or cv-unqualified type `void*` (in the latter case the pointer is first converted to `void*`).] Pointers to objects or functions of the same type (after pointer conversions) can be compared; the result depends on the relative positions of the pointed-to objects or functions in the address space as follows:

— If two pointers of the same type point to the same object or function, or both point one past the end of the same array, or are both null, they compare equal.

— If two pointers of the same type point to different objects or functions, or only one of them is null, they compare unequal.

— If two pointers point to nonstatic data members of the same object, the pointer to the later declared member compares greater provided the two members are not separated by an *access-specifier* label (11.1) and provided their class is not a union.

— If two pointers point to nonstatic members of the same object separated by an *access-specifier* label (11.1) the result is unspecified.

— If two pointers point to data members of the same union object, they compare equal (after conversion to `void*`, if necessary). If two pointers point to elements of the same array or one beyond the end of the array, the pointer to the object with the higher subscript compares higher.

— Other pointer comparisons are implementation-defined.

3

5.10 Equality operators [expr.eq]

1

equality-expression:
  relational-expression
  equality-expression `==` *relational-expression*
  equality-expression `!=` *relational-expression*

The `==` (equal to) and the `!=` (not equal to) operators have the same semantic restrictions, conversions, and result type as the relational operators except for their lower precedence and truth-value result. [*Note:* `a<b == c<d` is `true` whenever `a<b` and `c<d` have the same truth-value.]

2 In addition, pointers to members of the same type can be compared. Pointer to member conversions (4.11) are performed. A pointer to member can be compared to an integral constant expression that evaluates to zero. If one operand is a pointer to a virtual member function and the other is not the null pointer to member value, the result is unspecified.

5.11 Bitwise AND operator [expr.bit.and]

1 *and-expression:*
  equality-expression
  and-expression & *equality-expression*

The usual arithmetic conversions are performed; the result is the bitwise AND function of the operands. The operator applies only to integral operands.

5.12 Bitwise exclusive OR operator [expr.xor]

1 *exclusive-or-expression:*
  and-expression
  exclusive-or-expression ^ *and-expression*

The usual arithmetic conversions are performed; the result is the bitwise exclusive OR function of the operands. The operator applies only to integral operands.

5.13 Bitwise inclusive OR operator [expr.or]

1 *inclusive-or-expression:*
  exclusive-or-expression
  inclusive-or-expression | *exclusive-or-expression*

The usual arithmetic conversions are performed; the result is the bitwise inclusive OR function of its operands. The operator applies only to integral operands.

5.14 Logical AND operator [expr.log.and]

1 *logical-and-expression:*
  inclusive-or-expression
  logical-and-expression && *inclusive-or-expression*

The && operator groups left-to-right. The operands are both converted to type `bool` (4.13). The result is `true` if both operands are `true` and `false` otherwise. Unlike &, && guarantees left-to-right evaluation: the second operand is not evaluated if the first operand is `false`.

2 The result is a `bool`. All side effects of the first expression except for destruction of temporaries (12.2) happen before the second expression is evaluated.

5.15 Logical OR operator [expr.log.or]

1 *logical-or-expression:*
  logical-and-expression
  logical-or-expression || *logical-and-expression*

The `||` operator groups left-to-right. The operands are both converted to `bool` (4.13). It returns `true` if either of its operands is `true`, and `false` otherwise. Unlike `|`, `||` guarantees left-to-right evaluation; moreover, the second operand is not evaluated if the first operand evaluates to `true`.

2 The result is a `bool`. All side effects of the first expression except for destruction of temporaries (12.2) happen before the second expression is evaluated.

5.16 Conditional operator [expr.cond]

1 *conditional-expression:*
  logical-or-expression
  logical-or-expression ? *expression* : *assignment-expression*

Conditional expressions group right-to-left. The first expression is converted to `bool` (4.13). It is evaluated and if it is `true`, the result of the conditional expression is the value of the second expression, otherwise that of the third expression. All side effects of the first expression except for destruction of temporaries (12.2) happen before the second or third expression is evaluated.

2 If either the second or third expression is a *throw-expression* (15.1), the result is of the type of the other.

3 If both the second and the third expressions are of arithmetic type, then if they are of the same type the result is of that type; otherwise the usual arithmetic conversions are performed to bring them to a common type. Otherwise, if both the second and the third expressions are either a pointer or an integral constant expression that evaluates to zero, pointer conversions (4.10) are performed to bring them to a common type, which shall be a cv-qualified or cv-unqualified version of the type of either the second or the third expression. Otherwise, if both the second and the third expressions are either a pointer to member or an integral constant expression that evaluates to zero, pointer to member conversions (4.11) are performed to bring them to a common type[48] which shall be a cv-qualified or cv-unqualified version of the type of either the second or the third expression. Otherwise, if both the second and the third expressions are lvalues of related class types, they are converted to a common type (which shall be a *cv*-qualified or *cv*-unqualified version of the type of either the second third expression) as if by a cast to a reference to the common type (5.2.8). Otherwise, if both the second and the third expressions are of the same class `T`, the common type is `T`. Otherwise, if both the second and the third expressions have type "*cv* `void`", the common type is "*cv* `void`." Otherwise the expression is ill formed. The result has the common type; only one of the second and third expressions is evaluated. The result is an lvalue if the second and the third operands are of the same type and both are lvalues.

5.17 Assignment operators [expr.ass]

1 There are several assignment operators, all of which group right-to-left. All require a modifiable lvalue as their left operand, and the type of an assignment expression is that of its left operand. The result of the assignment operation is the value stored in the left operand after the assignment has taken place; the result is an lvalue.

assignment-expression:
  conditional-expression
  unary-expression assignment-operator assignment-expression
  throw-expression

assignment-operator: one of
```
=  *=  /=  %=   +=  -=  >>=  <<=  &=  ^=  |=
```

2 In simple assignment (`=`), the value of the expression replaces that of the object referred to by the left operand.

3 If the left operand is not of class type, the expression is converted to the cv-unqualified type of the left operand using standard conversions (4) and/or user-defined conversions (12.3), as necessary.

4 Assignment to objects of a class (9) `X` is defined by the function `X::operator=()` (13.5.3). Unless the user defines an `X::operator=()`, the default version is used for assignment (12.8). This implies that an object of a class derived from `X` (directly or indirectly) by unambiguous public derivation (10) can be assigned to an `X.`

5 For class objects, assignment is not in general the same as initialization (8.5, 12.1, 12.6, 12.8).

6 When the left operand of an assignment operator denotes a reference to `T`, the operation assigns to the object of type `T` denoted by the reference.

7 The behavior of an expression of the form `E1` *op*`=` `E2` is equivalent to `E1=E1` *op* `E2` except that `E1` is evaluated only once. `E1` shall not have `bool` type. In `+=` and `-=`, `E1` shall either have arithmetic type or be a pointer to a possibly-qualified completely defined object type. In all other cases, `E1` shall have arithmetic type.

[48] This is one instance in which the "composite type", as described in the C Standard, is still employed in C++.

8 See 15.1 for throw expressions.

5.18 Comma operator [expr.comma]

1 The comma operator groups left-to-right.

expression:
    assignment-expression
    expression `,` *assignment-expression*

A pair of expressions separated by a comma is evaluated left-to-right and the value of the left expression is discarded. All side effects of the left expression are performed before the evaluation of the right expression. The type and value of the result are the type and value of the right operand; the result is an lvalue if its right operand is.

2 In contexts where comma is given a special meaning, [*Example:* in lists of arguments to functions (5.2.2) and lists of initializers (8.5)] the comma operator as described in this clause can appear only in parentheses. [*Example:*

```
f(a, (t=3, t+2), c);
```

has three arguments, the second of which has the value `5`.]

5.19 Constant expressions [expr.const]

1 In several places, C++ requires expressions that evaluate to an integral or enumeration constant: as array bounds (8.3.4, 5.3.4), as `case` expressions (6.4.2), as bit-field lengths (9.7), as enumerator initializers (7.2), and as member constant initializers (9.5.2).

constant-expression:
    conditional-expression

An *integral constant-expression* can involve only literals (2.9), enumerators, `const` values of integral or enumeration types initialized with constant expressions (8.5), and `sizeof` expressions. Floating literals (2.9.3) can appear only if they are cast to integral or enumeration types. Only type conversions to integral or enumeration types can be used. In particular, except in `sizeof` expressions, functions, class objects, pointers, or references shall not be used, and assignment, increment, decrement, function-call, or comma operators shall not be used.

2 Other expressions are considered *constant-expression*s only for the purpose of non-local static object initialization (3.6.2). Such constant expressions shall evaluate to one of the following:

— a null pointer value (4.10),

— a null member pointer value (4.11),

— an arithmetic constant expression,

— an address constant expression,

— an address constant expression for an object type plus or minus an integral constant expression, or

— a pointer to member constant expression.

3 An *arithmetic constant expression* shall have arithmetic or enumeration type and shall only have operands that are integer literals (2.9.1), floating literals (2.9.3), enumerators, character literals (2.9.2) and `sizeof` expressions (5.3.3). Cast operators in an arithmetic constant expression shall only convert arithmetic or enumeration types to arithmetic or enumeration types, except as part of an operand to the `sizeof` operator.

4 An *address constant expression* is a pointer to an lvalue designating an object of static storage duration or a function. The pointer shall be created explicitly, using the unary `&` operator, or implicitly using an expression of array (4.2) or function (4.3) type. The subscripting operator `[]` and the class member access `.` and `->` operators, the `&` and `*` unary operators, and pointer casts (except `dynamic_cast`s, 5.2.6) can be used

in the creation of an address constant expression, but the value of an object shall not be accessed by the use of these operators. An expression that designates the address of a member or base class of a non-POD class object (9) is not an address constant expression (12.7). Function calls shall not be used in an address constant expression, even if the function is `inline` and has a reference return type.

5 A *pointer to member constant expression* shall be created using the unary `&` operator applied to a *qualified-id* operand (5.3.1).

6 Statements [stmt.stmt]

1 Except as indicated, statements are executed in sequence.

> *statement:*
>   *labeled-statement*
>   *expression-statement*
>   *compound-statement*
>   *selection-statement*
>   *iteration-statement*
>   *jump-statement*
>   *declaration-statement*
>   *try-block*

6.1 Labeled statement [stmt.label]

1 A statement can be labeled.

> *labeled-statement:*
>   *identifier* `:` *statement*
>   `case` *constant-expression* `:` *statement*
>   `default :` *statement*

An identifier label declares the identifier. The only use of an identifier label is as the target of a `goto`. The scope of a label is the function in which it appears. Labels shall not be redeclared within a function. A label can be used in a `goto` statement before its definition. Labels have their own name space and do not interfere with other identifiers.

2 Case labels and default labels shall occur only in switch statements.

6.2 Expression statement [stmt.expr]

1 Expression statements have the form

> *expression-statement:*
>   *expression*$_{opt}$ `;`

All side effects from an expression statement are completed before the next statement is executed. An expression statement with the expression missing is called a null statement. [*Note:* Most statements are expression statements—usually assignments or function calls. A null statement is useful to carry a label just before the `}` of a compound statement and to supply a null body to an iteration statement such as `while` (6.5.1). —*end note*]

6.3 Compound statement or block [stmt.block]

1 So that several statements can be used where one is expected, the compound statement (also, and equivalently, called "block") is provided.

> *compound-statement:*
>   `{` *statement-seq*$_{opt}$ `}`

statement-seq:
  statement
  statement-seq statement

A compound statement defines a local scope (3.3). [*Note:* a declaration is a *statement* (6.7). —*end note*]

6.4 Selection statements [stmt.select]

1 Selection statements choose one of several flows of control.

selection-statement:
  `if (` *condition* `)` *statement*
  `if (` *condition* `)` *statement* `else` *statement*
  `switch (` *condition* `)` *statement*

condition:
  expression
  type-specifier-seq declarator `=` *assignment-expression*

In this clause, the term *substatement* refers to the contained *statement* or *statement*s that appear in the syntax notation. The substatement in a *selection-statement* (both substatements, in the `else` form of the `if` statement) implicitly defines a local scope (3.3). [*Example:* If the substatement in a selection-statement is a single statement and not a *compound-statement,* it is as if it was rewritten to be a compound-statement containing the original substatement.

```
if (x)
    int i;
```

can be equivalently rewritten as

```
if (x) {
    int i;
}
```

Thus after the `if` statement, `i` is no longer in scope. —*end example*]

2 The rules for *condition*s apply both to *selection-statement*s and to the `for` and `while` statements (6.5). The *declarator* shall not specify a function or an array. The *type-specifier* shall not contain `typedef` and shall not declare a new class or enumeration.

3 A name introduced by a declaration in a *condition* is in scope from its point of declaration until the end of the substatements controlled by the condition. If the name is re-declared in the outermost block of a substatement controlled by the condition, the declaration that re-declares the name is ill-formed.

4 The value of a *condition* that is an initialized declaration is the value of a temporary object of type *bool* initialized with the value of the declared variable. The value of a *condition* that is an expression is the value of the expression. The value of the condition will be referred to as simply "the condition" where the usage is unambiguous.

5 If a *condition* can be syntactically resolved as either an expression or the declaration of a local name, it is interpreted as a declaration.

6.4.1 The `if` statement [stmt.if]

1 The condition is converted to type `bool`; if that is not possible, the program is ill-formed. If it yields `true` the first substatement is executed. If the `else` part of the selection statement is present and the condition yields `false`, the second substatement is executed. In the second form of `if` statement (the one including `else`), if the first substatement is also an `if` statement then that inner `if` statement shall contain an `else` part.[49)]

[49)] In other words, the `else` is associated with the nearest un-elsed `if`.

6.4.2 The `switch` statement [stmt.switch]

1 The `switch` statement causes control to be transferred to one of several statements depending on the value of a condition.

2 The condition shall be of integral type or of a class or enumeration type for which an unambiguous conversion to integral type exists (12.3). Integral promotion is performed. Any statement within the `switch` statement can be labeled with one or more case labels as follows:

`case` *constant-expression* `:`

where the *constant-expression* (5.19) is converted to the promoted type of the switch condition. No two of the case constants in the same switch shall have the same value after conversion to the promoted type of the switch condition.

3 There shall be at most one label of the form

```
default :
```

within a `switch` statement.

4 Switch statements can be nested; a `case` or `default` label is associated with the smallest switch enclosing it.

5 When the `switch` statement is executed, its condition is evaluated and compared with each case constant. If one of the case constants is equal to the value of the condition, control is passed to the statement following the matched case label. If no case constant matches the condition, and if there is a `default` label, control passes to the statement labeled by the default label. If no case matches and if there is no `default` then none of the statements in the switch is executed.

6 `case` and `default` labels in themselves do not alter the flow of control, which continues unimpeded across such labels. To exit from a switch, see `break`, 6.6.1. [*Note:* Usually, the substatement that is the subject of a switch is compound and `case` and `default` labels appear on the top-level statements contained within the (compound) substatement, but this is not required. Declarations can appear in the substatement of a *switch-statement*.]

6.5 Iteration statements [stmt.iter]

1 Iteration statements specify looping.

iteration-statement:
  `while (` *condition* `)` *statement*
  `do` *statement* `while (` *expression* `) ;`
  `for (` *for-init-statement* *condition*$_{opt}$ `;` *expression*$_{opt}$ `)` *statement*

for-init-statement:
  expression-statement
  simple-declaration

[*Note:* Note that a *for-init-statement* ends with a semicolon. —*end note*]

2 The substatement in an *iteration-statement* implicitly defines a local scope (3.3) which is entered and exited each time through the loop.

3 If the substatement in an iteration-statement is a single statement and not a *compound-statement,* it is as if it was rewritten to be a compound-statement containing the original statement. [*Example:*

```
while (--x >= 0)
    int i;
```

can be equivalently rewritten as

```
while (--x >= 0) {
    int i;
}
```

Thus after the `while` statement, `i` is no longer in scope. —*end example*]

4 The requirements on *condition*s are the same as for *if* statements (6.4.1).

6.5.1 The `while` statement [stmt.while]

1 The condition is converted to `bool` (4.13).

2 In the `while` statement the substatement is executed repeatedly until the value of the condition becomes `false`. The test takes place before each execution of the substatement.

6.5.2 The `do` statement [stmt.do]

1 The condition is converted to `bool` (4.13).

2 In the `do` statement the substatement is executed repeatedly until the value of the condition becomes `false`. The test takes place after each execution of the statement.

6.5.3 The `for` statement [stmt.for]

1 The condition is converted to `bool` (4.13).

2 The `for` statement

`for (` *for-init-statement condition*$_{opt}$ `;` *expression*$_{opt}$ `)` *statement*

is equivalent to

```
{
        for-init-statement
        while ( condition ) {
                statement
                expression ;
        }
}
```

except that a `continue` in *statement* (not enclosed in another iteration statement) will execute *expression* before re-evaluating *condition*. [*Note:* Thus the first statement specifies initialization for the loop; the condition specifies a test, made before each iteration, such that the loop is exited when the condition becomes `false`; the expression often specifies incrementing that is done after each iteration. —*end note*]

3 Either or both of the condition and the expression can be omitted. A missing *condition* makes the implied `while` clause equivalent to `while(true)`.

4 If the *for-init-statement* is a declaration, the scope of the name(s) declared extends to the end of the *for-statement*. [*Example:*

```
int i = 42;
int a[10];

for (int i = 0; i < 10; i++)
        a[i] = i;

int j = i;        // j = 42
```

—*end example*]

6.6 Jump statements [stmt.jump]

1 Jump statements unconditionally transfer control.

jump-statement:
    `break ;`
    `continue ;`
    `return` *expression*$_{opt}$ `;`
    `goto` *identifier* `;`

2 On exit from a scope (however accomplished), destructors (12.4) are called for all constructed objects with automatic storage duration (3.7.2) (named objects or temporaries) that are declared in that scope, in the reverse order of their declaration. Transfer out of a loop, out of a block, or back past an initialized variable with automatic storage duration involves the destruction of variables with automatic storage duration that are in scope at the point transferred from but not at the point transferred to. (See 6.7 for transfers into blocks). [*Note:* However, the program can be terminated (by calling `exit()` or `abort()`(18.3), for example) without destroying class objects with automatic storage duration. —*end note*]

6.6.1 The `break` statement [stmt.break]

1 The `break` statement shall occur only in an *iteration-statement* or a `switch` statement and causes termination of the smallest enclosing *iteration-statement* or `switch` statement; control passes to the statement following the terminated statement, if any.

6.6.2 The `continue` statement [stmt.cont]

1 The `continue` statement shall occur only in an *iteration-statement* and causes control to pass to the loop-continuation portion of the smallest enclosing *iteration-statement*, that is, to the end of the loop. More precisely, in each of the statements

```
while (foo) {       do {                for (;;) {
 {                                       { {
  // ...              // ...              // ...
 }                                       } }
contin: ;           contin: ;           contin: ;
}                   } while (foo);      }
```

a `continue` not contained in an enclosed iteration statement is equivalent to `goto contin`.

6.6.3 The `return` statement [stmt.return]

1 A function returns to its caller by the `return` statement.

2 A return statement without an expression can be used only in functions that do not return a value, that is, a function with the return value type `void`, a constructor (12.1), or a destructor (12.4). A return statement with an expression can be used only in functions returning a value; the value of the expression is returned to the caller of the function. If required, the expression is converted, as in an initialization (8.5), to the return type of the function in which it appears. A return statement can involve the construction and copy of a temporary object (12.2). Flowing off the end of a function is equivalent to a `return` with no value; this results in undefined behavior in a value-returning function.

6.6.4 The `goto` statement [stmt.goto]

1 The `goto` statement unconditionally transfers control to the statement labeled by the identifier. The identifier shall be a label (6.1) located in the current function.

6.7 Declaration statement [stmt.dcl]

1 A declaration statement introduces one or more new identifiers into a block; it has the form

declaration-statement:
    block-declaration

If an identifier introduced by a declaration was previously declared in an outer block, the outer declaration is hidden for the remainder of the block, after which it resumes its force.

2 Variables with automatic storage duration (3.7.2) are initialized each time their *declaration-statement* is executed. Variables with automatic storage duration declared in the block are destroyed on exit from the block (6.6).

3 It is possible to transfer into a block, but not in a way that bypasses declarations with initialization. A program that jumps from a point where a local variable with automatic storage duration is not in scope to a point where it is in scope is ill-formed unless the variable has pointer or arithmetic type or is an aggregate (8.5.1), and is declared without an *initializer* (8.5). [*Example:*

```
void f()
{
    // ...
    goto lx;    // ill-formed: jump into scope of 'a'
    // ...
ly:
    X a = 1;
    // ...
lx:
    goto ly;    // ok, jump implies destructor
                // call for 'a' followed by construction
                // again immediately following label ly
}
```

—*end example*]

4 The zero-initialization (8.5) of all local objects with static storage duration (3.7.1) is performed before any other initialization takes place. A local object with static storage duration (3.7.1) initialized with a *constant-expression* is initialized before its block is first entered. A local object with static storage duration not initialized with a *constant-expression* is initialized the first time control passes completely through its declaration. If the initialization exits by throwing an exception, the initialization is not complete, so it will be tried again the next time the function is called.

5 The destructor for a local object with static storage duration will be executed if and only if the variable was constructed. The destructor is called either immediately before or as part of the calls of the `atexit()` functions (18.3). Exactly when is unspecified.

6.8 Ambiguity resolution [stmt.ambig]

1 There is an ambiguity in the grammar involving *expression-statement*s and *declaration*s: An *expression-statement* with a function-style explicit type conversion (5.2.3) as its leftmost subexpression can be indistinguishable from a *declaration* where the first *declarator* starts with a `(`. In those cases the *statement* is a *declaration*. [*Note:* To disambiguate, the whole *statement* might have to be examined to determine if it is an *expression-statement* or a *declaration*. This disambiguates many examples. [*Example:* assuming `T` is a *simple-type-specifier* (7.1.5),

```
T(a)->m = 7;       // expression-statement
T(a)++;            // expression-statement
T(a,5)<<c;         // expression-statement
```

```
T(*d)(int);        // declaration
T(e)[];            // declaration
T(f) = { 1, 2 };   // declaration
T(*g)(double(3));  // declaration
```

—*end example*] In the last example above, `g`, which is a pointer to `T`, is initialized to `double(3)`. This is of course ill-formed for semantic reasons, but that does not affect the syntactic analysis.

2 The remaining cases are *declaration*s. [*Example:*

```
T(a);          // declaration
T(*b)();       // declaration
T(c)=7;        // declaration
T(d),e,f=3;    // declaration
T(g)(h,2);     // declaration
```

—*end example*]

3 The disambiguation is purely syntactic; that is, the meaning of the names, beyond whether they are *type-id*s or not, is not used in the disambiguation.]

4 A slightly different ambiguity between *expression-statement*s and *declaration*s is resolved by requiring a *type-id* for function declarations within a block (6.3). [*Example:*

```
void g()
{
    int f();   // declaration
    int a;     // declaration
    f();       // expression-statement
    a;         // expression-statement
}
```

—*end example*]

7 Declarations [dcl.dcl]

1 A declaration introduces one or more names into a program and specifies how those names are to be interpreted. Declarations have the form

> *declaration-seq:*
>   *declaration*
>   *declaration-seq declaration*
>
> *declaration:*
>   *block-declaration*
>   *function-definition*
>   *template-declaration*
>   *linkage-specification*
>   *namespace-definition*
>
> *block-declaration:*
>   *simple-declaration*
>   *asm-definition*
>   *namespace-alias-definition*
>   *using-declaration*
>   *using-directive*
>
> *simple-declaration:*
>   *decl-specifier-seq*$_{opt}$ *init-declarator-list*$_{opt}$;

[*Note: asm-definition*s are described in 7.4, and *linkage-specification*s are described in 7.5. *Function-definition*s are described in 8.4 and *template-declaration*s are described in 14. *Namespace-definition*s are described in 7.3.1, *using-declaration*s are described in 7.3.3 and *using-directive*s are described in 7.3.4.] The description of the general form of declaration

> *decl-specifier-seq*$_{opt}$ *init-declarator-list*$_{opt}$;

is divided into two parts: *decl-specifier*s, the components of a *decl-specifier-seq*, are described in 7.1 and *declarator*s, the components of an *init-declarator-list*, are described in 8.

2 A declaration occurs in a scope (3.3); the scope rules are summarized in 3.4. A declaration that declares a function or defines a class, namespace, template, or function also has one or more scopes nested within it. These nested scopes, in turn, can have declarations nested within them. Unless otherwise stated, utterances in this clause about components in, of, or contained by a declaration or subcomponent thereof refer only to those components of the declaration that are *not* nested within scopes nested within the declaration.

3 In the general form of declaration, the optional *init-declarator-list* can be omitted only when declaring a class (9), enumeration (7.2) or namespace (7.3.1), that is, when the *decl-specifier-seq* contains either a *class-specifier*, an *elaborated-type-specifier* with a *class-key* (9.1), an *enum-specifier*, or a *namespace-definition*. In these cases and whenever a *class-specifier, enum-specifier*, or *namespace-definition* is present in the *decl-specifier-seq*, the identifiers in these specifiers are among the names being declared by the declaration (as *class-names*, *enum-names, enumerators*, or *namespace-name*, depending on the syntax).

4 Each *init-declarator* in the *init-declarator-list* contains exactly one *declarator-id*, which is the name declared by that *init-declarator* and hence one of the names declared by the declaration. The *type-specifiers* (7.1.5) in the *decl-specifier-seq* and the recursive *declarator* structure of the *init-declarator* describe a type

(8.3), which is then associated with the name being declared by the *init-declarator*.

5 If the *decl-specifier-seq* contains the `typedef` specifier, the declaration is called a *typedef declaration* and the name of each *init-declarator* is declared to be a *typedef-name*, synonymous with its associated type (7.1.3). If the *decl-specifier-seq* contains no `typedef` specifier, the declaration is called a *function declaration* if the type associated with the name is a function type (8.3.5) and an *object declaration* otherwise.

6 Syntactic components beyond those found in the general form of declaration are added to a function declaration to make a *function-definition*. An object declaration, however, is also a definition unless it contains the `extern` specifier and has no initializer (3.1). A definition causes the appropriate amount of storage to be reserved and any appropriate initialization (8.5) to be done.

7 Only in *function-definitions* (8.4) and in function declarations for constructors, destructors, and type conversions can the *decl-specifier-seq* be omitted.

8 The names declared by a declaration are introduced into the scope in which the declaration occurs, except that the presence of a `friend` specifier (11.4), certain uses of the *elaborated-type-specifier* (7.1.5.3), and *using-directive*s (7.3.4) alter this general behavior.

9 In a declaration in which the *declarator-id* is a *qualified-id*, names before the *qualified-id* being defined are sought in the defining scope. Names following the *qualified-id* are sought in the scope of the member's class or namespace.

7.1 Specifiers [dcl.spec]

1 The specifiers that can be used in a declaration are

decl-specifier:
  storage-class-specifier
  type-specifier
  function-specifier
  `friend`
  `typedef`

decl-specifier-seq:
  decl-specifier-seq$_{opt}$ *decl-specifier*

2 The longest sequence of *decl-specifier*s that could possibly be a type name is taken as the *decl-specifier-seq* of a *declaration*. The sequence shall be self-consistent as described below. [*Example:*

```
typedef char* Pc;
static Pc;              // error: name missing
```

Here, the declaration `static Pc` is ill-formed because no name was specified for the static variable of type `Pc`. To get a variable of type `int` called `Pc`, the *type-specifier* `int` has to be present to indicate that the *typedef-name* `Pc` is the name being (re)declared, rather than being part of the *decl-specifier* sequence. For another example,

```
void f(const Pc);       // void f(char* const)  (not const char*)
void g(const int Pc);   // void g(const int)
```

—*end example*]

3 [*Note:* since `signed`, `unsigned`, `long`, and `short` by default imply `int`, a *type-name* appearing after one of those specifiers is treated as the name being (re)declared. [*Example:*

```
void h(unsigned Pc);       // void h(unsigned int)
void k(unsigned int Pc);   // void k(unsigned int)
```

—*end example*] —*end note*]

7.1.1 Storage class specifiers [dcl.stc]

1 The storage class specifiers are

```
        storage-class-specifier:
                auto
                register
                static
                extern
                mutable
```

At most one *storage-class-specifier* shall appear in a given *decl-specifier-seq*. If a *storage-class-specifier* appears in a *decl-specifier-seq*, there can be no `typedef` specifier in the same *decl-specifier-seq* and the *init-declarator-list* of the declaration shall not be empty (except for global anonymous unions, which shall be declared `static` (9.6). The *storage-class-specifier* applies to the name declared by each *init-declarator* in the list and not to any names declared by other specifiers.

2 The `auto` or `register` specifiers can be applied only to names of objects declared in a block (6.3) or to function parameters (8.4). They specify that the named object has automatic storage duration (3.7.2). An object declared without a *storage-class-specifier* at block scope or declared as a function parameter has automatic storage duration by default. Hence, the `auto` specifier is almost always redundant and not often used; one use of `auto` is to distinguish a *declaration-statement* from an *expression-statement* (6.2) explicitly.

3 A `register` specifier has the same semantics as an `auto` specifier together with a hint to the implementation that the object so declared will be heavily used. The hint can be ignored and in most implementations it will be ignored if the address of the object is taken.

4 The `static` specifier can be applied only to names of objects and functions and to anonymous unions (9.6). There can be no `static` function declarations within a block, nor any `static` function parameters. A `static` specifier used in the declaration of an object declares the object to have static storage duration (3.7.1). A `static` specifier can be used in declarations of class members; 9.5 describes its effect. A name declared with a `static` specifier in a scope other than class scope (3.3.5) has internal linkage. For a nonmember function, an `inline` specifier is equivalent to a `static` specifier for linkage purposes (3.5) unless the inline declaration explicitly includes `extern` as part of its *decl-specifier* or matches a previous declaration of the function, in which case the function name retains the linkage of the previous declaration.

5 The `extern` specifier can be applied only to the names of objects and functions. The `extern` specifier cannot be used in the declaration of class members or function parameters. An object or function introduced by a declaration with an `extern` specifier has external linkage unless the declaration matches a visible prior declaration at namespace scope of the same object or function, in which case the object or function has the linkage specified by the prior declaration.[50)]

6 A name declared in a namespace scope without a *storage-class-specifier* has external linkage unless it has internal linkage because of a previous declaration and provided it is not declared `const`. Objects declared `const` and not explicitly declared `extern` have internal linkage.

7 The linkages implied by successive declarations for a given entity shall agree. That is, within a given scope, each declaration declaring the same object name or the same overloading of a function name shall imply the same linkage. Each function in a given set of overloaded functions can have a different linkage, however. [*Example:*

```
static char* f(); // f() has internal linkage
char* f()         // f() still has internal linkage
    { /* ... */ }
```

50) ‘‘Prior’’ declarations can be introduced in enclosing scopes. This implies that a name specified `static` at namespace scope and then specified `extern` in an inner scope still has internal linkage.

```
char* g();         // g() has external linkage
static char* g()  // error: inconsistent linkage
    { /* ... */ }

void h();
inline void h();  // external linkage

inline void l();
void l();         // internal linkage

inline void m();
extern void m();  // internal linkage

static void n();
inline void n();  // internal linkage

static int a;     // ‘a’ has internal linkage
int a;            // error: two definitions

static int b;     // ‘b’ has internal linkage
extern int b;     // ‘b’ still has internal linkage

int c;            // ‘c’ has external linkage
static int c;     // error: inconsistent linkage

extern int d;     // ‘d’ has external linkage
static int d;     // error: inconsistent linkage
```

—*end example*]

8 The name of a declared but undefined class can be used in an `extern` declaration. Such a declaration, however, cannot be used before the class has been defined. [*Example:*

```
struct S;
extern S a;
extern S f();
extern void g(S);

void h()
{
    g(a);       // error: S undefined
    f();        // error: S undefined
}
```

—*end example*] The `mutable` specifier can be applied only to names of class data members (9.2) and can not be applied to names declared `const` or `static`. [*Example:*

```
class X {
        mutable const int* p;   // ok
        mutable int* const q;   // ill-formed
};
```

—*end example*]

9 The `mutable` specifier on a class data member nullifies a `const` specifier applied to the containing class object and permits modification of the mutable class member even though the rest of the object is *const* (7.1.5.1).

7.1.2 Function specifiers [dcl.fct.spec]

1 *Function-specifiers* can be used only in function declarations.

```
function-specifier:
        inline
        virtual
        explicit
```

2 The `inline` specifier is a hint to the implementation that inline substitution of the function body is to be preferred to the usual function call implementation. The hint can be ignored. The `inline` specifier shall not appear on a block scope function declaration. For the linkage of inline functions, see 3.5 and 7.1.1. A function (8.3.5, 9.4, 11.4) defined within the class definition is inline.

3 An inline function shall be defined in every translation unit in which it is used (3.2), and shall have exactly the same definition in every case (see one definition rule, 3.2). If a function with external linkage is declared inline in one translation unit, it shall be declared inline in all translation units in which it appears. A call to an inline function shall not precede its definition.[51] [*Example:*

```
class X {
public:
    int f();
    inline int g();
};

void k(X* p)
{
    int i = p->f();
    int j = p->g();  // A call appears before X::g is defined
                     // ill-formed
    // ...
}

inline int X::f()    // Declares X::f as an inline function
                     // A call appears before X::f is defined
                     // ill-formed
{
    // ...
}

inline int X::g()
{
    // ...
}
```

—*end example*]

4 The `virtual` specifier shall be used only in declarations of nonstatic class member functions within a class declaration; see 10.3.

5 The `explicit` specifier shall be used only in declarations of constructors within a class declaration; see 12.3.1.

[51] Many function calls are implicit, particularly calls to constructors, destructors, conversions, and `operator new`. Although such calls are implicit, that does not affect the requirement that the function definitions precede their calls.

7.1.3 The `typedef` specifier [dcl.typedef]

1 Declarations containing the *decl-specifier* `typedef` declare identifiers that can be used later for naming fundamental (3.9.1) or compound (3.9.2) types. The `typedef` specifier shall not be used in a *function-definition* (8.4), and it shall not be combined in a *decl-specifier-seq* with any other kind of specifier except a *type-specifier*.

typedef-name:
    identifier

A name declared with the `typedef` specifier becomes a *typedef-name*. Within the scope of its declaration, a *typedef-name* is syntactically equivalent to a keyword and names the type associated with the identifier in the way described in 8. A *typedef-name* is thus a synonym for another type. A *typedef-name* does not introduce a new type the way a class declaration (9.1) or enum declaration does. [*Example:* after

```
typedef int MILES, *KLICKSP;
```

the constructions

```
MILES distance;
extern KLICKSP metricp;
```

are all correct declarations; the type of `distance` is `int`; that of `metricp` is "pointer to `int`."]

2 In a given scope, a `typedef` specifier can be used to redefine the name of any type declared in that scope to refer to the type to which it already refers. [*Example:*

```
typedef struct s { /* ... */ } s;
typedef int I;
typedef int I;
typedef I I;
```

—*end example*]

3 In a given scope, a `typedef` specifier shall not be used to redefine the name of any type declared in that scope to refer to a different type. [*Example:*

```
class complex { /* ... */ };
typedef int complex;    // error: redefinition
```

—*end example*] Similarly, in a given scope, a class or enumeration shall not be declared with the same name as a *typedef-name* that is declared in that scope and refers to a type other than the class or enumeration itself. [*Example:*

```
typedef int complex;
class complex { /* ... */ };  // error: redefinition
```

—*end example*]

4 A *typedef-name* that names a class is a *class-name* (9.1). The *typedef-name* shall not be used after a `class`, `struct`, or `union` prefix and not in the names for constructors and destructors within the class declaration itself. [*Example:*

```
struct S {
    S();
    ~S();
};

typedef struct S T;

S a = T();      // ok
struct T * p;   // error
```

—*end example*]

5 An unnamed class defined in a declaration with a `typedef` specifier gets a dummy name. For linkage purposes only (3.5), the first *typedef-name* declared by the declaration is used to denote the class type in place of the dummy name. [*Example:*

```
typedef struct { } S, R; // 'S' is the class name for linkage purposes
```

—*end example*] The *typedef-name* is still only a synonym for the dummy name and shall not be used where a true class name is required. [*Note:* such a class cannot have user-declared constructors or destructors because they cannot be named by the user. [*Example:*

```
typedef struct {
    S();    // error: requires a return type since S is
            // an ordinary member function, not a constructor
} S;
```

—*end example*] —*end note*] If an unnamed class is defined in a `typedef` declaration but the declaration does not declare a class type, the name of the class for linkage purposes is a dummy name. [*Example:*

```
typedef struct { }* ps; // 'ps' is not the class linkage name
```

—*end example*]

6 A *typedef-name* that names an enumeration is an *enum-name* (7.2). The *typedef-name* shall not be used after an `enum` prefix.

7.1.4 The `friend` specifier [dcl.friend]

1 The `friend` specifier is used to specify access to class members; see 11.4.

7.1.5 Type specifiers [dcl.type]

1 The type-specifiers are

type-specifier:
  simple-type-specifier
  class-specifier
  enum-specifier
  elaborated-type-specifier
  cv-qualifier

As a general rule, at most one *type-specifier* is allowed in the complete *decl-specifier-seq* of a *declaration*. The only exceptions to this rule are the following:

2

— `const` or `volatile` can be combined with any other *type-specifier*. However, redundant cv-qualifiers are prohibited except when introduced through the use of typedefs (7.1.3) or template type arguments (14.8), in which case the redundant cv-qualifiers are ignored.

— `signed` or `unsigned` can be combined with `char`, `long`, `short`, or `int`.

— `short` or `long` can be combined with `int`.

— `long` can be combined with `double`.

3 At least one *type-specifier* is required in a typedef declaration. At least one *type-specifier* is required in a function declaration unless it declares a constructor, destructor or type conversion operator.[52)]

4 *class-specifier*s and *enum-specifier*s are discussed in 9 and 7.2, respectively. The remaining *type-specifier*s are discussed in the rest of this section.

[52)] There is no special provision for a *decl-specifier-seq* that lacks a *type-specifier*. The “implicit int” rule of C is no longer supported.

7.1.5.1 The *cv-qualifiers* [dcl.type.cv]

1 There are two *cv-qualifiers*, `const` and `volatile`. [*Note:* Subclause 3.9.3 describes how cv-qualifiers affect object and function types.]

2 Unless explicitly declared `extern`, a `const` object does not have external linkage and shall be initialized (8.5, 12.1); for a `const` object of type `T`, if `T` is a class with a user-declared default constructor, the constructor for `T` is called, otherwise, if the `const` object is not initialized with an explicit *initializer*, the program is ill-formed. An integral or enumeration `const` object initialized by an integral or enumeration constant expression can be used in integral or enumeration constant expressions (5.19).

3 CV-qualifiers are supported by the type system so that they cannot be subverted without casting (5.2.10). A pointer or reference to a cv-qualified type need not actually point or refer to a cv-qualified object, but it is treated as if it does; a const-qualified access path cannot be used to modify an object even if the object referenced is a non-const object and can be modified through some other access path.

4 Except that any class member declared `mutable` (7.1.1) can be modified, any attempt to modify a `const` object during its lifetime (3.8) results in undefined behavior.

5 [*Example:*

```
const int ci = 3;  // cv-qualified (initialized as required)
ci = 4;            // ill-formed: attempt to modify const

int i = 2;         // not cv-qualified
const int* cip;    // pointer to const int
cip = &i;          // okay: cv-qualified access path to unqualified
*cip = 4;          // ill-formed: attempt to modify through ptr to const

int* ip;
ip = const_cast<int*> cip; // cast needed to convert const int* to int*
*ip = 4;           // defined: *ip points to i, a non-const object

const int* ciq = new const int (3); // initialized as required
int* iq = const_cast<int*> ciq;     // cast required
iq = 4;            // undefined: modifies a const object
```

6 For another example

```
class X {
    public:
        mutable int i;
        int j;
};
class Y { public: X x; }

const Y y;
y.x.i++;        // well-formed: mutable member can be modified
y.x.j++;        // ill-formed: const-qualified member modified
Y* p = const_cast<Y*>(&y);      // cast away const-ness of y
p->x.i = 99;    // well-formed: mutable member can be modified
p->x.j = 99;    // undefined: modifies a const member
```

—*end example*]

7 [*Note:* `volatile` is a hint to the processor to avoid aggressive optimization involving the object because the value of the object might be changed by means undetectable by a processor. See 1.8 for detailed semantics. In general, the semantics of `volatile` are intended to be the same in C++ as they are in C.]

7.1.5.2 Simple type specifiers [dcl.type.simple]

1 The simple type specifiers are

simple-type-specifier:
  `::`$_{opt}$ *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

The *simple-type-specifier*s specify either a previously-declared user-defined type or one of the fundamental types (3.9.1). Table 7 summarizes the valid combinations of *simple-type-specifier*s and the types they specify.

Table 7—*simple-type-specifier*s and the types they specify

Specifier(s)	Type
type-name	the type named
`char`	“`char`”
`unsigned char`	“`unsigned char`”
`signed char`	“`signed char`”
`bool`	“`bool`”
`unsigned`	“`unsigned int`”
`unsigned int`	“`unsigned int`”
`signed`	“`int`”
`signed int`	“`int`”
`int`	“`int`”
`unsigned short int`	“`unsigned short int`”
`unsigned short`	“`unsigned short int`”
`unsigned long int`	“`unsigned long int`”
`unsigned long`	“`unsigned long int`”
`signed long int`	“`long int`”
`signed long`	“`long int`”
`long int`	“`long int`”
`long`	“`long int`”
`signed short int`	“`short int`”
`signed short`	“`short int`”
`short int`	“`short int`”
`short`	“`short int`”
`wchar_t`	“`wchar_t`”
`float`	“`float`”
`double`	“`double`”
`long double`	“`long double`”
`void`	“`void`”

When multiple *simple-type-specifiers* are allowed, they can be freely intermixed with other *decl-specifiers* in any order. It is implementation-defined whether bit-fields and objects of `char` type are represented as signed or unsigned quantities. The `signed` specifier forces `char` objects and bit-fields to be signed; it is redundant with other integral types.

7.1.5.3 Elaborated type specifiers [dcl.type.elab]

1 Generally speaking, the *elaborated-type-specifier* is used to refer to a previously declared *class-name* or *enum-name* even though the name can be hidden by an intervening object, function, or enumerator declaration (3.3), but in some cases it also can be used to declare a *class-name*.

elaborated-type-specifier:
  class-key `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *identifier*
  `enum` *::*$_{opt}$ *nested-name-specifier*$_{opt}$ *identifier*

class-key:
  `class`
  `struct`
  `union`

2 If an *elaborated-type-specifier* is the sole constituent of a declaration, the declaration is ill-formed unless it has one of the following forms:

— *class-key identifier* `;`

in which case the *elaborated-type-specifier* declares the *identifier* to be a class-name in the scope that contains the declaration (9.1);

3 — `friend` *class-key identifier* `;`

in which case, if the *identifier* in the *elaborated-type-specifier* has not been previously declared, the *elaborated-type-specifier* declares the *identifier* to be a class-name in the smallest enclosing non-class, non-function prototype scope that contains the declaration; otherwise the *identifier* is resolved as when the *elaborated-type-specifier* is not the sole constituent of a declaration;

4 — `friend` *class-key ::identifier* `;`
`friend` *class-key nested-name-specifier identifier* `;`

in which case the *identifier* is resolved as when the *elaborated-type-specifier* is not the sole constituent of a declaration.

5 If the *elaborated-type-specifier* is not the sole constituent of the declaration, the *identifier* following the *class-key* or `enum` keyword is resolved as described in 3.4 according to its qualifications, if any, but ignoring any objects, functions, or enumerators that have been declared. If the *identifier* resolves to a *class-name* or *enum-name*, the *elaborated-type-specifier* introduces it into the declaration the same way a *simple-type-specifier* introduces its *type-name*. If the *identifier* resolves to a *typedef-name*, the *elaborated-type-specifier* is ill-formed. If the resolution is unsuccessful, the *elaborated-type-specifier* is ill-formed unless it is of the simple form *class-key identifier*. In this case, the *identifier* is declared in the smallest non-class, non-function prototype scope that contains the declaration.

6 The *class-key* or `enum` keyword present in the *elaborated-type-specifier* shall agree in kind with the declaration to which the name in the *elaborated-type-specifier* refers. This rule also applies to the form of *elaborated-type-specifier* that declares a *class-name* or `friend` class since it can be construed as referring to the definition of the class. Thus, in any *elaborated-type-specifier*, the `enum` keyword shall be used to refer to an enumeration (7.2), the `union` *class-key* shall be used to refer to a union (9), and either the `class` or `struct` *class-key* shall be used to refer to a structure (9) or to a class declared using the `class` *class-key*. [*Example:*

```
struct Node {
        struct Node* Next;      // ok: Refers to Node at global scope
        struct Data* Data;      // ok: Declares type Data
                                // at global scope and member Data
};

struct Data {
        struct Node* Node;      // ok: Refers to Node at global scope
        friend struct ::Glob;   // error: Glob is not declared
                                // cannot introduce a qualified type
        friend struct Glob;     // ok: Declares Glob in global scope
        /* ... */
};

struct Base {
        struct Data;                    // ok: Declares nested Data
        struct ::Data*     thatData;    // ok: Refers to ::Data
        struct Base::Data* thisData;    // ok: Refers to nested Data

        friend class ::Data;            // ok: global Data is a friend
        friend class Data;              // ok: nested Data is a friend
        struct Data { /* ... */ };      // Defines nested Data

        struct Data;                    // ok: Redeclares nested Data
};
```

```
        struct Data;            // ok: Redeclares Data at global scope

        struct ::Data;          // error: cannot introduce a qualified type
        struct Base::Data;      // error: cannot introduce a qualified type
        struct Base::Datum;     // error: Datum undefined

        struct Base::Data* pBase;       // ok: refers to nested Data
```

—*end example*]

7.2 Enumeration declarations [dcl.enum]

1 An enumeration is a distinct type (3.9.1) with named constants. Its name becomes an *enum-name*, within its scope.

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

The identifiers in an *enumerator-list* are declared as constants, and can appear wherever constants are required. If no *enumerator-definition*s with `=` appear, then the values of the corresponding constants begin at zero and increase by one as the *enumerator-list* is read from left to right. An *enumerator-definition* with `=` gives the associated *enumerator* the value indicated by the *constant-expression*; subsequent *enumerator*s without initializers continue the progression from the assigned value. The *constant-expression* shall be of integral or enumeration type.

2 [*Example:*

```
        enum { a, b, c=0 };
        enum { d, e, f=e+2 };
```

defines `a`, `c`, and `d` to be zero, `b` and `e` to be `1`, and `f` to be `3`.]

3 The point of declaration for an enumerator is immediately after its *enumerator-definition*. [*Example:*

```
        const int x = 12;
        { enum { x = x }; }
```

Here, the enumerator `x` is initialized with the value of the constant `x`, namely 12.]

4 Each enumeration defines a type that is different from all other types. The type of an enumerator is its enumeration.

5 The *underlying type* of an enumeration is an integral type, not gratuitously larger than `int`,[53] that can represent all enumerator values defined in the enumeration. If the *enumerator-list* is empty, the underlying type is as if the enumeration had a single enumerator with value 0. The value of `sizeof()` applied to an enumeration type, an object of enumeration type, or an enumerator, is the value of `sizeof()` applied to

[53] The type should be larger than `int` only if the value of an enumerator won't all fit in an `int` or `unsigned int`.

the underlying type.

6 For an enumeration where $e_{\min}$ is the smallest enumerator and $e_{\max}$ is the largest, the values of the enumeration are the values of the underlying type in the range $b_{\min}$ to $b_{\max}$, where $b_{\min}$ and $b_{\max}$ are, respectively, the smallest and largest values of the smallest bit-field that can store $e_{\min}$ and $e_{\max}$. On a two's-complement machine, $b_{\max}$ is the smallest value greater than or equal to $\max(abs(e_{\min})-1, abs(e_{\max}))$ of the form $2^M - 1$; $b_{\min}$ is zero if $e_{\min}$ is non-negative and $-(b_{\max}+1)$ otherwise. It is possible to define an enumeration that has values not defined by any of its enumerators.

7 Two enumeration types are layout-compatible if they have the same sets of enumerator values.

8 The value of an enumerator or an object of an enumeration type is converted to an integer by integral promotion (4.5). [*Example:*

```
enum color { red, yellow, green=20, blue };
color col = red;
color* cp = &col;
if (*cp == blue) // ...
```

makes `color` a type describing various colors, and then declares `col` as an object of that type, and `cp` as a pointer to an object of that type. The possible values of an object of type `color` are `red`, `yellow`, `green`, `blue`; these values can be converted to the integral values `0`, `1`, `20`, and `21`. Since enumerations are distinct types, objects of type `color` can be assigned only values of type `color`.

```
color c = 1;     // error: type mismatch,
                 // no conversion from int to color

int i = yellow;  // ok: yellow converted to integral value 1
                 // integral promotion
```

See also C.3.]

9 An expression of arithmetic or enumeration type or of type `wchar_t` can be converted to an enumeration type explicitly. The value is unchanged if it is in the range of enumeration values of the enumeration type; otherwise the resulting enumeration value is unspecified.

10 The enum-name and each enumerator declared by an enum-specifier is declared in the scope that immediately contains the enum-specifier. These names obey the scope rules defined for all names in (3.3) and (3.4). An enumerator declared in class scope can be referred to using the class member access operators (`::`, `.` (dot) and `->` (arrow)), see 5.2.4. [*Example:*

```
class X {
public:
    enum direction { left='l', right='r' };
    int f(int i)
        { return i==left ? 0 : i==right ? 1 : 2; }
};

void g(X* p)
{
    direction d;        // error: 'direction' not in scope
    int i;
    i = p->f(left);     // error: 'left' not in scope
    i = p->f(X::right); // ok
    i = p->f(p->left);  // ok
    // ...
}
```

—*end example*]

7.3 Namespaces [basic.namespace]

1 A namespace is an optionally-named declarative region. The name of a namespace can be used to access entities declared in that namespace; that is, the members of the namespace. Unlike other declarative regions, the definition of a namespace can be split over several parts of one or more translation units.

2 A name declared outside all named namespaces, blocks (6.3) and classes (9) has global namespace scope (3.3.4).

7.3.1 Namespace definition [namespace.def]

1 The grammar for a *namespace-definition* is

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$_{opt}$

2 The *identifier* in an *original-namespace-definition* shall not have been previously defined in the declarative region in which the *original-namespace-definition* appears. The *identifier* in an *original-namespace-definition* is the name of the namespace. Subsequently in that declarative region, it is treated as an *original-namespace-name*.

3 The *original-namespace-name* in an *extension-namespace-definition* shall have previously been defined in an *original-namespace-definition* in the same declarative region.

4 Every *namespace-definition* shall appear in the global scope or in a namespace scope (3.3.4).

7.3.1.1 Explicit qualification [namespace.qual]

1 A name in a class or namespace can be accessed using qualification according to the grammar:

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

2 The *namespace-name*s in a *nested-name-specifier* shall have been previously defined by a *named-namespace-definition* or a *namespace-alias-definition*.

3 The search for the initial qualifier preceding any `::` operator locates only the names of types or namespaces. The search for a name after a `::` locates only named members of a namespace or class. In particular, *using-directive*s (7.3.4) are ignored, as is any enclosing declarative region.

7.3.1.2 Unnamed namespaces [namespace.unnamed]

1 An *unnamed-namespace-definition* behaves as if it were replaced by

```
namespace unique { namespace-body }
using namespace unique;
```

where, for each translation unit, all occurrences of ***unique*** in that translation unit are replaced by an identifier that differs from all other identifiers in the entire program.[54] [*Example:*

```
namespace { int i; }       // unique::i
void f() { i++; }          // unique::i++

namespace A {
        namespace {
                int i;     // A::unique::i
                int j;     // A::unique::j
        }
        void g() { i++; }  // A::unique::i++
}

using namespace A;
void h() {
        i++;               // error: unique::i or A::unique::i
        A::i++;            // error: A::i undefined
        j++;               // A::unique::j
}
```

—*end example*]

7.3.1.3 Namespace scope [namespace.scope]

1 The declarative region of a *namespace-definition* is its *namespace-body*. The potential scope denoted by an *original-namespace-name* is the concatenation of the declarative regions established by each of the *namespace-definition*s in the same declarative region with that *original-namespace-name*. Entities declared in a *namespace-body* are said to be *member*s of the namespace, and names introduced by these declarations

[54] Although entities in an unnamed namespace might have external linkage, they are effectively qualified by a name unique to their translation unit and therefore can never be seen from any other translation unit.

into the declarative region of the namespace are said to be *member names* of the namespace. [*Example:*

```
namespace N {
        int i;
        int g(int a) { return a; }
        int k();
        void q();
}

namespace { int l=1; }

namespace N {
        int g(char a)         // overloads N::g(int)
        {
                return l+a;   // l is from unnamed namespace
        }

        int i;                // error: duplicate definition

        int k();              // ok: duplicate function declaration

        int k()               // ok: definition of N::k()
        {
                return g(i);  // calls N::g(int)
        }

        int q();              // error: different return type
}
```

—*end example*]

2 Because a *namespace-definition* contains *declaration*s in its *namespace-body* and a *namespace-definition* is itself a *declaration*, it follows that *namespace-definition*s can be nested. [*Example:*

```
namespace Outer {
        int i;
        namespace Inner {
                void f() { i++; } // Outer::i
                int i;
                void g() { i++; } // Inner::i
        }
}
```

—*end example*]

3 The use of the `static` keyword is deprecated when declaring objects in a namespace scope (see _future.directions_); the *unnamed-namespace* provides a superior alternative.

7.3.1.4 Namespace member definitions [namespace.memdef]

1 Members of a namespace can be defined within that namespace. [*Example:*

```
namespace X {
        void f() { /* ... */ }
}
```

—*end example*]

2 Members of a named namespace can also be defined outside that namespace by explicit qualification (7.3.1.1) of the name being defined, provided that the entity being defined was already declared in the namespace and the definition appears after the point of declaration in a namespace that encloses the declaration's namespace. [*Example:*

```
namespace Q {
        namespace V {
                void f();
        }
        void V::f() { /* ... */ }  // fine
        void V::g() { /* ... */ }  // error: g() is not yet a member of V
        namespace V {
                void g();
        }
}

namespace R {
        void Q::V::g() { /* ... */ } // error: R doesn't enclose Q
}
```

—*end example*]

3 Every name first declared in a namespace is a member of that namespace. A `friend` function first declared within a class is a member of the innermost enclosing namespace. [*Example:*

```
// Assume f and g have not yet been defined.
namespace A {
        class X {
                friend void f(X);  // declaration of f
                class Y {
                        friend void g();
                };
        };

        void f(X) { /* ... */}     // definition of f declared above
        X x;
        void g() { f(x); }         // f and g are members of A
}

using A::x;

void h()
{
        A::f(x);
        A::X::f(x);    // error: f is not a member of A::X
        A::X::Y::g();  // error: g is not a member of A::X::Y
}
```

—*end example*] The scope of class names first introduced in *elaborated-type-specifiers* is described in (7.1.5.3).

4 When an entity declared with the `extern` specifier is not found to refer to some other declaration, then that entity is a member of the innermost enclosing namespace. However such a declaration does not introduce the member name in its namespace scope. [*Example:*

```
namespace X {
        void p()
        {
                q();              // error: q not yet declared
                extern void q();  // q is a member of namespace X
        }

        void middle()
        {
                q();              // error: q not yet declared
        }
        void q() { /* ... */ }    // definition of X::q
}

void q() { /* ... */ }            // some other, unrelated q
```

—*end example*]

7.3.2 Namespace or class alias [namespace.alias]

1 A *namespace-alias-definition* declares an alternate name for a namespace according to the following grammar:

> *namespace-alias:*
>   *identifier*
>
> *namespace-alias-definition:*
>   `namespace` *identifier* `=` *qualified-namespace-specifier ;*
>
> *qualified-namespace-specifier:*
>   *::*$_{opt}$ *nested-name-specifier*$_{opt}$ *class-or-namespace-name*

2 The *identifier* in a *namespace-alias-definition* is a synonym for the name of the namespace denoted by the *qualified-namespace-specifier* and becomes a *namespace-alias*.

3 In a declarative region, a *namespace-alias-definition* can be used to redefine a *namespace-alias* declared in that declarative region to refer to the namespace to which it already refers. [*Example:* the following declarations are well-formed:

```
namespace Company_with_very_long_name { /* ... */ }
namespace CWVLN = Company_with_very_long_name;
namespace CWVLN = Company_with_very_long_name;  // ok: duplicate
namespace CWVLN = CWVLN;
```

—*end example*]

4 A *namespace-name* or *namespace-alias* shall not be declared as the name of any other entity in the same declarative region. A *namespace-name* defined at global scope shall not be declared as the name of any other entity in any global scope of the program. No diagnostic is required for a violation of this rule by declarations in different translation units.

7.3.3 The `using` declaration [namespace.udecl]

1 A *using-declaration* introduces a name into the declarative region in which the *using-declaration* appears. That name is a synonym for the name of some entity declared elsewhere.

> *using-declaration:*
>   `using ::`$_{opt}$ *nested-name-specifier unqualified-id* `;`
>   `using ::` *unqualified-id* `;`

2 The member names specified in a *using-declaration* are declared in the declarative region in which the *using-declaration* appears.

3 Every *using-declaration* is a *declaration* and a *member-declaration* and so can be used in a class definition. [*Example:*

```
struct B {
        void f(char);
        void g(char);
};
```

```
        struct D : B {
                using B::f;
                void f(int) { f('c'); } // calls B::f(char)
                void g(int) { g('c'); } // recursively calls D::g(int)
        };
```

—*end example*]

4 A *using-declaration* used as a *member-declaration* shall refer to a member of a base class of the class being defined. [*Example:*

```
        class C {
                int g();
        };

        class D2 : public B {
                using B::f;  // ok: B is a base of D
                using C::g;  // error: C isn't a base of D2
        };
```

—*end example*]

5 A *using-declaration* for a member shall be a *member-declaration.* [*Example:*

```
        struct X {
                int i;
                static int s;
        };

        void f()
        {
                using X::i;  // error: X::i is a class member
                             // and this is not a member declaration.
                using X::s;  // error: X::s is a class member
                             // and this is not a member declaration.
        }
```

—*end example*]

6 Members declared by a *using-declaration* can be referred to by explicit qualification just like other member names (7.3.1.1). In a *using-declaration*, a prefix `::` refers to the global namespace (as ever). [*Example:*

```
        void f();

        namespace A {
                void g();
        }

        namespace X {
                using ::f;   // global f
                using A::g;  // A's g
        }

        void h()
        {
                X::f();      // calls ::f
                X::g();      // calls A::g
        }
```

—*end example*]

7 A *using-declaration* is a *declaration* and can therefore be used repeatedly where (and only where) multiple declarations are allowed. [*Example:*

```
namespace A {
        int i;
}

namespace A1 {
        using A::i;
        using A::i; // ok: double declaration
}

void f()
{
        using A::i;
        using A::i; // error: double declaration
}

class B {
        int i;
};

class X : public B {
        using B::i;
        using B::i;  // error: double member declaration
};
```

—*end example*]

8 The entity declared by a *using-declaration* shall be known in the context using it according to its definition at the point of the *using-declaration*. Definitions added to the namespace after the *using-declaration* are not considered when a use of the name is made. [*Example:*

```
namespace A {
        void f(int);
}

using A::f;              // f is a synonym for A::f;
                         // that is, for A::f(int).
namespace A {
        void f(char);
}

void foo()
{
        f('a');          // calls f(int),
}                        // even though f(char) exists.

void bar()
{
        using A::f;      // f is a synonym for A::f;
                         // that is, for A::f(int) and A::f(char).
        f('a');          // calls f(char)
}
```

—*end example*]

9 A name defined by a *using-declaration* is an alias for its original declarations so that the *using-declaration* does not affect the type, linkage or other attributes of the members referred to.

10 If the set of local declarations and *using-declaration*s for a single name are given in a declarative region, they shall all refer to the same entity, or all refer to functions. [*Example:*

```
namespace B {
        int i;
        void f(int);
        void f(double);
}

void g()
{
        int i;
        using B::i;     // error: i declared twice
        void f(char);
        using B::f;     // fine: each f is a function
}
```

—*end example*]

11 If a local function declaration has the same name and type as a function introduced by a *using-declaration*, the program is ill-formed. [*Example:*

```
namespace C {
        void f(int);
        void f(double);
        void f(char);
}

void h()
{
        using B::f;   // B::f(int) and B::f(double)
        using C::f;   // C::f(int), C::f(double), and C::f(char)
        f('h');       // calls C::f(char)
        f(1);         // error: ambiguous: B::f(int) or C::f(int) ?
        void f(int);  // error: f(int) conflicts with C::f(int)
}
```

—*end example*]

12 When a *using-declaration* brings names from a base class into a derived class scope, member functions in the derived class override and/or hide virtual member functions with the same name and argument types in a base class (rather than conflicting). [*Example:*

```
struct B {
        virtual void f(int);
        virtual void f(char);
        void g(int);
        void h(int);
};

struct D : B {
        using B::f;
        void f(int);   // ok: D::f(int) overrides B::f(int);

        using B::g;
        void g(char);  // ok

        using B::h;
        void h(int);   // ok: D::h(int) hides B::h(int)
};
```

```
void k(D* p)
{
        p->f(1);    // calls D::f(int)
        p->f('a');  // calls B::f(char)
        p->g(1);    // calls B::g(int)
        p->g('a');  // calls D::g(char)
}
```

—*end example*]

13 For the purpose of overload resolution, the functions which are introduced by a *using-declaration* into a derived class will be treated as though they were members of the derived class. In particular, the implicit `this` parameter shall be treated as if it were a pointer to the derived class rather than to the base class. This has no effect on the type of the function, and in all other respects the function remains a member of the base class.

14 All instances of the name mentioned in a *using-declaration* shall be accessible. In particular, if a derived class uses a *using-declaration* to access a member of a base class, the member name shall be accessible. If the name is that of an overloaded member function, then all functions named shall be accessible.

15 The alias created by the *using-declaration* has the usual accessibility for a *member-declaration*. [*Example:*

```
class A {
private:
        void f(char);
public:
        void f(int);
protected:
        void g();
};

class B : public A {
        using A::f; // error: A::f(char) is inaccessible
public:
        using A::g; // B::g is a public synonym for A::g
};
```

—*end example*]

16 [*Note:* Use of *access-declaration*s (11.3) is deprecated; member *using-declaration*s provide a better alternative.]

7.3.4 Using directive [namespace.udir]

1 *using-directive:*
  `using namespace ::`$_{opt}$ *nested-name-specifier*$_{opt}$ *namespace-name ;*

2 A *using-directive* specifies that the names in the namespace with the given *namespace-name*, including those specified by any *using-directive*s in that namespace, can be used in the scope in which the *using-directive* appears after the using directive, exactly as if the names from the namespace had been declared outside the namespace at the points where the namespace was defined. Furthermore, if the *using-directive* specifies a *nested-name-specifier*:

— if the *using-directive* appears in a namespace `A` and the namespace nominated by the *using-directive* is a nested namespace of `A`, the names from the nested namespace appear as if they were declared in namespace `A` at the point were the nested namespace was defined in `A`; otherwise,

— for a *using-directive* with a *nested-name-specifier* of the form `T1::...::Tn::` and a *namespace-name* `N`, the names from the nested namespace `N` appear as if they were declared outside of `T1::...::Tn::N` at the point where the nested namespace was defined.

A *using-directive* does not add any members to the declarative region in which it appears. If a namespace is extended by an *extended-namespace-definition* after a *using-directive* is given, the additional members of the extended namespace can be used after the *extended-namespace-definition.*

3 The *using-directive* is transitive: if a namespace contains a *using-directive* that nominates a second namespace that itself contains *using-directive*s, the effect is as if the *using-directive*s from the second namespace also appeared in the first. In particular, a name in a namespace does not hide names in a second namespace which is the subject of a *using-directive* in the first namespace. [*Example:*

```
namespace M {
        int i;
}

namespace N {
        int i;
        using namespace M;
}

void f()
{
        N::i = 7; // well-formed: M::i is not a member of N
        using namespace N;
        i = 7;    // error: both M::i and N::i are accessible
}
```

—*end example*]

4 During overload resolution, all functions from the transitive search are considered for argument matching. An ambiguity exists if the best match finds two functions with the same signature, even if one might seem to ‘‘hide’’ the other in the *using-directive* lattice. [*Example:*

```
namespace D {
        int d1;
        void f(char);
}
using namespace D;

int d1;            // ok: no conflict with D::d1

namespace E {
        int e;
        void f(int);
}

namespace D {       // namespace extension
        int d2;
        using namespace E;
        void f(int);
}

void f()
{
        d1++;      // error: ambiguous ::d1 or D::d1?
        ::d1++;    // ok
        D::d1++;   // ok
        d2++;      // ok: D::d2
        e++;       // ok: E::e
        f(1);      // error: ambiguous: D::f(int) or E::f(int)?
        f('a');    // ok: D::f(char)
}
```

—*end example*]

7.4 The `asm` declaration [dcl.asm]

1 An `asm` declaration has the form

asm-definition:
  `asm (` *string-literal* `) ;`

The meaning of an `asm` declaration is implementation-defined. [*Note:* Typically it is used to pass information through the processor to an assembler. —*end note*]

7.5 Linkage specifications [dcl.link]

1 Linkage (3.5) between C++ and non-C++ code fragments can be achieved using a *linkage-specification*:

linkage-specification:
  `extern` *string-literal* `{` *declaration-seq*$_{opt}$ `}`
  `extern` *string-literal* *declaration*

declaration-seq:
  declaration
  declaration-seq declaration

The *string-literal* indicates the required linkage. The meaning of the *string-literal* is implementation-defined. Every implementation shall provide for linkage to functions written in the C programming language, `"C"`, and linkage to C++ functions, `"C++"`. Default linkage is `"C++"`. [*Example:*

```
complex sqrt(complex);    // C++ linkage by default
extern "C" {
    double sqrt(double);  // C linkage
}
```

—*end example*]

2 Linkage specifications nest. A linkage specification does not establish a scope. A *linkage-specification* can occur only in namespace scope (3.3). A *linkage-specification* for a class applies to nonmember functions and objects declared within it. A *linkage-specification* for a function also applies to functions and objects declared within it. A linkage declaration with a string that is unknown to the implementation is ill-formed.

3 If a function or object has more than one *linkage-specification*, they shall agree; that is, they shall specify the same *string-literal*. Except for functions with C++ linkage, a function declaration without a linkage specification shall not precede the first linkage specification for that function. A function can be declared without a linkage specification after an explicit linkage specification has been seen; the linkage explicitly specified in the earlier declaration is not affected by such a function declaration.

4 At most one of a set of overloaded functions (13) with a particular name can have C linkage.

5 Linkage can be specified for objects. [*Example:*

```
extern "C" {
    // ...
    _iobuf _iob[_NFILE];
    // ...
    int _flsbuf(unsigned,_iobuf*);
    // ...
}
```

—*end example*] Functions and objects can be declared `static` or `inline` within the `{}` of a linkage specification. The linkage directive is ignored for a function or object with internal linkage (3.5). A function first declared in a linkage specification behaves as a function with external linkage. [*Example:*

```
extern "C" double f();
static double f();     // error
```

is ill-formed (7.1.1).] An object defined within an

```
        extern "C" { /* ... */ }
```

construct is still defined (and not just declared).

6 The linkage of a pointer to function affects only the pointer. When the pointer is dereferenced, the function to which it refers is considered to be a C++ function. There is no way to specify that the function to which a function pointer refers is written in another language

7 Linkage from C++ to objects defined in other languages and to objects defined in C++ from other languages is implementation-defined and language-dependent. Only where the object layout strategies of two language implementations are similar enough can such linkage be achieved. Taking the address of a function whose linkage is other than C++ or C produces undefined behavior.

8 When the name of a programming language is used to name a style of linkage in the *string-literal* in a *linkage-specification*, it is recommended that the spelling be taken from the document defining that language, [*Example:* `Ada` (not `ADA`) and `FORTRAN` (not `Fortran`).]

8 Declarators [dcl.decl]

1 A declarator declares a single object, function, or type, within a declaration. The *init-declarator-list* appearing in a declaration is a comma-separated sequence of declarators, each of which can have an initializer.

> *init-declarator-list:*
>   *init-declarator*
>   *init-declarator-list* `,` *init-declarator*
>
> *init-declarator:*
>   *declarator initializer*$_{opt}$

2 The two components of a *declaration* are the specifiers (*decl-specifier-seq*; 7.1) and the declarators (*init-declarator-list*). The specifiers indicate the fundamental type, storage class, or other properties of the objects and functions being declared. The declarators specify the names of these objects and functions and (optionally) modify the type with operators such as `*` (pointer to) and `()` (function returning). Initial values can also be specified in a declarator; initializers are discussed in 8.5 and 12.6.

3 Each *init-declarator* in a declaration is analyzed separately as if it was in a declaration by itself.[55)]

4 Declarators have the syntax

> *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* `)`

---

55) A declaration with several declarators is usually equivalent to the corresponding sequence of declarations each with a single declarator. That is

```
        T  D1, D2, ... Dn;
```

is usually equvalent to

```
        T  D1; T D2; ... T Dn;
```

where `T` is a *decl-specifier-seq* and each `Di` is a *init-declarator.* The exception occurs when one declarator modifies the name environment used by a following declarator, as in

```
        struct S { ... };
        S   S, T;  // declare two instances of struct S
```

which is not equivalent to

```
        struct S { ... };
        S   S;
        S   T;   // error
```

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*

A *class-name* has special meaning in a declaration of the class of that name and when qualified by that name using the scope resolution operator `::` (5.1, 12.1, 12.4).

8.1 Type names [dcl.name]

1 To specify type conversions explicitly, and as an argument of `sizeof`, `new`, or `typeid`, the name of a type shall be specified. This can be done with a *type-id*, which is syntactically a declaration for an object or function of that type that omits the name of the object or function.

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* `)`

It is possible to identify uniquely the location in the *abstract-declarator* where the identifier would appear if the construction were a declarator in a declaration. The named type is then the same as the type of the hypothetical identifier. [*Example:*

```
int                    // int i
int *                  // int *pi
int *[3]               // int *p[3]
int (*)[3]             // int (*p3i)[3]
int *()                // int *f()
int (*)(double)        // int (*pf)(double)
```

name respectively the types "integer," "pointer to integer," "array of 3 pointers to integers," "pointer to array of 3 integers," "function having no parameters and returning pointer to integer," and "pointer to function of `double` returning an integer."]

2 A type can also be named (often more easily) by using a *typedef* (7.1.3).

8.2 Ambiguity resolution [dcl.ambig.res]

1 The ambiguity arising from the similarity between a function-style cast and a declaration mentioned in 6.8 can also occur in the context of a declaration. In that context, it surfaces as a choice between a function declaration with a redundant set of parentheses around a parameter name and an object declaration with a function-style cast as the initializer. Just as for statements, the resolution is to consider any construct that could possibly be a declaration a declaration. A declaration can be explicitly disambiguated by a nonfunction-style cast or a = to indicate initialization. [*Example:*

```
struct S {
    S(int);
};

void foo(double a)
{
    S x(int(a));        // function declaration
    S x(int());         // function declaration
    S y((int)a);        // object declaration
    S z = int(a);       // object declaration
}
```

—*end example*]

2 The ambiguity arising from the similarity between a function-style cast and a *type-id* can occur in many different contexts. The ambiguity surfaces as a choice between a function-style cast expression and a declaration of a type. The resolution is that any construct that could possibly be a *type-id* in its syntactic context shall be considered a *type-id*.

3 [*Example:*

```
#include <cstddef>
char *p;
void *operator new(size_t, int);
void foo(int x)  {
        new (int(*p)) int;      // new-placement expression
        new (int(*[x]));        // new type-id
}
```

4 For another example,

```
template <class T>
struct S {
T *p;
};
S<int()> x;             // type-id
S<int(1)> y;            // expression (ill-formed)
```

5 For another example,

```
void foo()
{
        sizeof(int(1)); // expression
        sizeof(int());  // type-id (ill-formed)
}
```

6 For another example,

```
void foo()
{
        (int(1));       // expression
        (int())1;       // type-id (ill-formed)
}
```

—*end example*]

8.3 Meaning of declarators [dcl.meaning]

1 A list of declarators appears after an optional (7) *decl-specifier-seq* (7.1). Each declarator contains exactly one *declarator-id*; it names the identifier that is declared. A *declarator-id* shall be a simple *identifier*, except for the following cases: the declaration of some special functions (12.3, 12.4, 13.5), the definition of a member function (9.4), the definition of a static data member (9.5), the declaration of a friend function that is a member of another class (11.4). An `auto`, `static`, `extern`, `register`, `friend`, `inline`, `virtual`, or `typedef` specifier applies directly to each *declarator-id* in a *init-declarator-list*; the type specified for each *declarator-id* depends on both the *decl-specifier-seq* and its *declarator*.

2 Thus, a declaration of a particular identifier has the form

```
T D
```

where `T` is a *decl-specifier-seq* and `D` is a declarator. The following subsections give an inductive procedure for determining the type specified for the contained *declarator-id* by such a declaration.

3 First, the *decl-specifier-seq* determines a type. In a declaration

```
T D
```

the *decl-specifier-seq* `T` determines the type "`T`." [*Example:* in the declaration

```
int unsigned i;
```

the type specifiers `int unsigned` determine the type "`unsigned int`" (7.1.5.2).]

4 In a declaration `T D` where `D` is an unadorned identifier the type of this identifier is "`T`."

5 In a declaration `T D` where `D` has the form

```
( D1 )
```

the type of the contained *declarator-id* is the same as that of the contained *declarator-id* in the declaration

```
T D1
```

Parentheses do not alter the type of the embedded *declarator-id*, but they can alter the binding of complex declarators.

8.3.1 Pointers [dcl.ptr]

1 In a declaration `T D` where `D` has the form

`*` *cv-qualifier-seq*$_{opt}$ `D1`

and the type of the identifier in the declaration `T D1` is "*derived-declarator-type-list* `T`," then the type of the identifier of `D` is "*derived-declarator-type-list cv-qualifier-seq* pointer to `T`." The *cv-qualifier*s apply to the pointer and not to the object pointed to.

2 [*Example:* the declarations

```
const int ci = 10, *pc = &ci, *const cpc = pc, **ppc;
int i, *p, *const cp = &i;
```

declare `ci`, a constant integer; `pc`, a pointer to a constant integer; `cpc`, a constant pointer to a constant integer, `ppc`, a pointer to a pointer to a constant integer; `i`, an integer; `p`, a pointer to integer; and `cp`, a constant pointer to integer. The value of `ci`, `cpc`, and `cp` cannot be changed after initialization. The value of `pc` can be changed, and so can the object pointed to by `cp`. Examples of some correct operations are

```
i = ci;
*cp = ci;
pc++;
pc = cpc;
pc = p;
ppc = &pc;
```

Examples of ill-formed operations are

```
ci = 1;         // error
ci++;           // error
*pc = 2;        // error
cp = &ci;       // error
cpc++;          // error
p = pc;         // error
ppc = &p;       // error
```

Each is unacceptable because it would either change the value of an object declared `const` or allow it to be changed through a cv-unqualified pointer later, for example:

```
*ppc = &ci;  // okay, but would make p point to ci ...
             // ... because of previous error
*p = 5;      // clobber ci
```

—*end example*]

3 `volatile` specifiers are handled similarly.

4 See also 5.17 and 8.5.

5 There can be no pointers to references (8.3.2) or pointers to bit-fields (9.7).

8.3.2 References [dcl.ref]

1 In a declaration `T D` where `D` has the form

```
& D1
```

and the type of the identifier in the declaration `T D1` is "*derived-declarator-type-list* `T`," then the type of the identifier of `D` is "*derived-declarator-type-list* reference to `T`." At all times during the determination of a type, any type of the form "*cv-qualifier-seq* reference to `T`" is adjusted to be "reference to `T`". [*Example:* in

```
typedef int& A;
const A aref = 3;
```

the type of `aref` is "reference to `int`", not "`const` reference to `int`".] A declarator that specifies the type "reference to *cv* void" is ill-formed.

2 [*Example:*

```
void f(double& a) { a += 3.14; }
// ...
    double d = 0;
    f(d);
```

declares `a` to be a reference parameter of `f` so the call `f(d)` will add `3.14` to `d`.

```
int v[20];
// ...
int& g(int i) { return v[i]; }
// ...
g(3) = 7;
```

declares the function `g()` to return a reference to an integer so `g(3)=7` will assign `7` to the fourth element of the array `v`. For another example,

```
struct link {
    link* next;
};

link* first;
```

```
void h(link*& p)  // ‘p’ is a reference to pointer
{
    p->next = first;
    first = p;
    p = 0;
}

void k()
{
        link* q = new link;
        h(q);
}
```

declares `p` to be a reference to a pointer to `link` so `h(q)` will leave `q` with the value zero. See also 8.5.3.]

3 It is unspecified whether or not a reference requires storage (3.7).

4 There shall be no references to references, no references to bit-fields (9.7), no arrays of references, and no pointers to references. The declaration of a reference shall contain an *initializer* (8.5.3) except when the declaration contains an explicit `extern` specifier (7.1.1), is a class member (9.2) declaration within a class declaration, or is the declaration of a parameter or a return type (8.3.5); see 3.1. A reference shall be initialized to refer to a valid object or function. In particular, null references are prohibited; no diagnostic is required.

8.3.3 Pointers to members [dcl.mptr]

1 In a declaration `T D` where `D` has the form

`::`$_{opt}$ *nested-name-specifier* `*` *cv-qualifier-seq*$_{opt}$ `D1`

and the *nested-name-specifier* names a class, and the type of the identifier in the declaration `T D1` is "*derived-declarator-type-list* `T`," then the type of the identifier of `D` is "*derived-declarator-type-list cv-qualifier-seq* pointer to member of *class nested-name-specifier of type* `T`."

2 [*Example:*

```
class X {
public:
    void f(int);
    int a;
};
class Y;

int X::* pmi = &X::a;
void (X::* pmf)(int) = &X::f;
double X::* pmd;
char Y::* pmc;
```

declares `pmi`, `pmf`, `pmd` and `pmc` to be a pointer to a member of `X` of type `int`, a pointer to a member of `X` of type `void(int)`, a pointer to a member of `X` of type `double` and a pointer to a member of `Y` of type `char` respectively. The declaration of `pmd` is well-formed even though `X` has no members of type `double`. Similarly, the declaration of `pmc` is well-formed even though `Y` is an incomplete type. `pmi` and `pmf` can be used like this:

```
X obj;
//...
obj.*pmi = 7;   // assign 7 to an integer
                // member of obj
(obj.*pmf)(7);  // call a function member of obj
                // with the argument 7
```

—*end example*]

3 A pointer to member shall not point to a static member of a class (9.5), a member with reference type, or "*cv* `void`." [*Note:* There is no "reference-to-member" type in C++. See also 5.5 and 5.3.]

8.3.4 Arrays [dcl.array]

1 In a declaration `T D` where `D` has the form

`D1 [`*constant-expression$_{opt}$*`]`

and the type of the identifier in the declaration `T D1` is "*derived-declarator-type-list* `T`," then the type of the identifier of `D` is an array type. `T` shall not be a reference type, an incomplete type, a function type or an abstract class type. If the *constant-expression* (5.19) is present, its value shall be greater than zero. The constant expression specifies the *bound* of (number of elements in) the array. If the value of the constant expression is `N`, the array has `N` elements numbered `0` to `N-1`, and the type of the identifier of `D` is "*derived-declarator-type-list* array of `N T`." If the constant expression is omitted, the type of the identifier of `D` is "*derived-declarator-type-list* array of unknown bound of `T`," an incomplete object type. The type "*derived-declarator-type-list* array of `N T`" is a different type from the type "*derived-declarator-type-list* array of unknown bound of `T`," see 3.9. At all times during the determination of a type, any type of the form "*cv-qualifier-seq* array of `N T`" is adjusted to "array of `N` *cv-qualifier-seq* ,`T`" and similarly for "array of unknown bound of .`T`" [*Example:*

```
typedef int A[5], AA[2][3];
const A x;      // type is ‘‘array of 5 const int’’
const AA y;     // type is ‘‘array of 2 array of 3 const int’’
```

—*end example*]

2 An array can be constructed from one of the fundamental types[56] (except `void`), from a pointer, from a pointer to member, from a class, or from another array.

3 When several "array of" specifications are adjacent, a multidimensional array is created; the constant expressions that specify the bounds of the arrays can be omitted only for the first member of the sequence. [*Note:* this elision is useful for function parameters of array types, and when the array is external and the definition, which allocates storage, is given elsewhere.] The first *constant-expression* can also be omitted when the declarator is followed by an *initializer* (8.5). In this case the bound is calculated from the number of initial elements (say, `N`) supplied (8.5.1), and the type of the identifier of `D` is "array of `N T`."

4 [*Example:*

```
float fa[17], *afp[17];
```

declares an array of `float` numbers and an array of pointers to `float` numbers. For another example,

```
static int x3d[3][5][7];
```

declares a static three-dimensional array of integers, with rank 3×5×7. In complete detail, `x3d` is an array of three items; each item is an array of five arrays; each of the latter arrays is an array of seven integers. Any of the expressions `x3d`, `x3d[i]`, `x3d[i][j]`, `x3d[i][j][k]` can reasonably appear in an expression.]

5 [*Note:* conversions affecting lvalues of array type are described in 4.2. Objects of array types cannot be modified, see 3.10.]

6 Except where it has been declared for a class (13.5.5), the subscript operator `[]` is interpreted in such a way that `E1[E2]` is identical to `*((E1)+(E2))`. Because of the conversion rules that apply to `+`, if `E1` is an array and `E2` an integer, then `E1[E2]` refers to the `E2`-th member of `E1`. Therefore, despite its asymmetric appearance, subscripting is a commutative operation.

56) The enumeration types are included in the fundamental types.

7 A consistent rule is followed for multidimensional arrays. If `E` is an *n*-dimensional array of rank $i \times j \times \cdots \times k$, then `E` appearing in an expression is converted to a pointer to an $(n-1)$-dimensional array with rank $j \times \cdots \times k$. If the `*` operator, either explicitly or implicitly as a result of subscripting, is applied to this pointer, the result is the pointed-to $(n-1)$-dimensional array, which itself is immediately converted into a pointer.

8 [*Example:* consider

```
int x[3][5];
```

Here `x` is a 3×5 array of integers. When `x` appears in an expression, it is converted to a pointer to (the first of three) five-membered arrays of integers. In the expression `x[i]`, which is equivalent to `*(x+i)`, `x` is first converted to a pointer as described; then `x+i` is converted to the type of `x`, which involves multiplying `i` by the length of the object to which the pointer points, namely five integer objects. The results are added and indirection applied to yield an array (of five integers), which in turn is converted to a pointer to the first of the integers. If there is another subscript the same argument applies again; this time the result is an integer.]

9 [*Note:* it follows from all this that arrays in C++ are stored row-wise (last subscript varies fastest) and that the first subscript in the declaration helps determine the amount of storage consumed by an array but plays no other part in subscript calculations.]

8.3.5 Functions [dcl.fct]

1 In a declaration `T D` where `D` has the form

> `D1 (` *parameter-declaration-clause* `)` *cv-qualifier-seq*$_{opt}$ *exception-specification*$_{opt}$

and the type of the contained *declarator-id* in the declaration `T D1` is "*derived-declarator-type-list* `T`," the type of the *declarator-id* in `D` is "*derived-declarator-type-list cv-qualifier-seq*$_{opt}$ function with parameters of type *parameter-declaration-clause* and returning `T`"; a type of this form is a *function type*[57].

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*

2 The *parameter-declaration-clause* determines the arguments that can be specified, and their processing, when the function is called. If the *parameter-declaration-clause* terminates with an ellipsis, the number of arguments shall be equal to or greater than the number of parameters specified; if it is empty, the function takes no arguments. The parameter list `(void)` is equivalent to the empty parameter list. Except for this special case, `void` shall not be a parameter type (though types derived from `void`, such as `void*`, can). Where syntactically correct, "`, ...`" is synonymous with "`...`". [*Note:* the standard header `<cstdarg>` contains a mechanism for accessing arguments passed using the ellipsis (see 5.2.2 and 18.7).]

3 A single name can be used for several different functions in a single scope; this is function overloading (13). All declarations for a function with a given parameter list shall agree exactly both in the type of the value returned and in the number and type of parameters; the presence or absence of the ellipsis is

57) As indicated by the syntax, cv-qualifiers are a significant component in function return types.

considered part of the function type. The type of each parameter is determined from its own *decl-specifier-seq* and *declarator*. After determining the type of each parameter, any parameter of type “array of `T`” or “function returning `T`” is adjusted to be “pointer to `T`” or “pointer to function returning `T`,” respectively. After producing the list of parameter types, several transformations take place upon the types. Any *cv-qualifier* modifying a parameter type is deleted; e.g., the type `void(const int)` becomes `void(int)`. Such *cv-qualifier*s affect only the definition of the parameter within the body of the function. If the *storage-class-specifier* `register` modifies a parameter type, the specifier is deleted; e.g., `register char*` becomes `char*`. Such *storage-class-qualifier*s affect only the definition of the parameter within the body of the function. The resulting list of transformed parameter types is the function’s *parameter type list*. The return type and the parameter type list, but not the default arguments (8.3.6) or exception specification (15.4), are part of the function type. If the type of a parameter includes a type of the form “pointer to array of unknown bound of `T`” or “reference to array of unknown bound of `T`,” the program is ill-formed.[58] A *cv-qualifier-seq* can only be part of a declaration or definition of a nonstatic member function, and of a pointer to a member function; see 9.4.2. It is part of the function type.

4 Functions shall not return arrays or functions, although they can return pointers and references to such things. There shall be no arrays of functions, although there can be arrays of pointers to functions.

5 Types shall not be defined in return or parameter types.

6 [*Note:* the *parameter-declaration-clause* is used to check and convert arguments in calls and to check pointer-to-function, reference-to-function, and pointer-to-member-function assignments and initializations.]

7 An identifier can optionally be provided as a parameter name; if present in a function definition (8.4), it names a parameter (sometimes called “formal argument”). [*Note:* in particular, parameter names are also optional in function definitions and names used for a parameter in different declarations and the definition of a function need not be the same. If an identifier is present in a function declaration, it cannot be used since it goes out of scope at the end of the function declarator (3.3);]

8 [*Note:* The *exception-specification* is described in 15.4 .]

9 [*Example:* the declaration

```
int i,
    *pi,
    f(),
    *fpi(int),
    (*pif)(const char*, const char*);
    (*fpif(int))(int);
```

declares an integer `i`, a pointer `pi` to an integer, a function `f` taking no arguments and returning an integer, a function `fpi` taking an integer argument and returning a pointer to an integer, a pointer `pif` to a function which takes two pointers to constant characters and returns an integer, a function `fpif` taking an integer argument and returning a pointer to a function that takes an integer argument and returns an integer. It is especially useful to compare `fpi` and `pif`. The binding of `*fpi(int)` is `*(fpi(int))`, so the declaration suggests, and the same construction in an expression requires, the calling of a function `fpi`, and then using indirection through the (pointer) result to yield an integer. In the declarator `(*pif)(const char*, const char*)`, the extra parentheses are necessary to indicate that indirection through a pointer to a function yields a function, which is then called.

10 Typedefs are sometimes convenient when the return type of a function is complex. For another example, the function `fpif` above could have been declared

[58] This excludes parameters of type “*ptr-arr-seq* `T2`” where `T2` is “pointer to array of unknown bound of `T`” and where *ptr-arr-seq* means any sequence of “pointer to” and “array of” derived declarator types. This exclusion applies to the parameters of the function, and if a parameter is a pointer to function or pointer to member function then to its parameters also, etc.

```
typedef int  IFUNC(int);
IFUNC*  fpif(int);
```

11 The declaration

```
int fseek(FILE*, long, int);
```

declares a function taking three arguments of the specified types, and returning `int` (7.1.5). The declaration

```
int printf(const char*, ...);
```

declares a function that can be called with varying numbers and types of arguments.

```
printf("hello world");
printf("a=%d b=%d", a, b);
```

However, the first argument must be of a type that can be converted to a `const char*`.

12 —*end example*]

8.3.6 Default arguments [dcl.fct.default]

1 If an expression is specified in a parameter declaration this expression is used as a default argument. Default arguments will be used in calls where trailing arguments are missing.

2 [*Example:* the declaration

```
void point(int = 3, int = 4);
```

declares a function that can be called with zero, one, or two arguments of type `int`. It can be called in any of these ways:

```
point(1,2);  point(1);  point();
```

The last two calls are equivalent to `point(1,4)` and `point(3,4)`, respectively.]

3 A default argument expression shall be specified only in the *parameter-declaration-clause* of a function declaration or in a *template-parameter* (14.7). If it is specified in a *parameter-declaration-clause*, it shall not occur within a *declarator* or *abstract-declarator* of a *parameter-declaration*.[59)]

4 For non-template functions, default arguments can be added in later declarations of a function in the same scope. Declarations in different scopes have completely distinct sets of default arguments. That is, declarations in inner scopes do not acquire default arguments from declarations in outer scopes, and vice versa. In a given function declaration, all parameters subsequent to a parameter with a default argument shall have default arguments supplied in this or previous declarations. A default argument shall not be redefined by a later declaration (not even to the same value). [*Example:*

```
void f(int, int);
void f(int, int = 7);
void h()
{
    f(3);                       // ok, calls f(3, 7)
    void f(int = 1, int);       // error: does not use default
                                // from surrounding scope
}
```

59) This means that default arguments cannot appear, for example, in declarations of pointers to functions, references to functions, or `typedef` declarations.

```
void m()
{
    void f(int, int);           // has no defaults
    f(4);                       // error: wrong number of arguments
    void f(int, int = 5);       // ok
    f(4);                       // ok, calls f(4, 5);
    void f(int, int = 5);       // error: cannot redefine, even to
                                // same value
}
void n()
{
    f(6);                       // ok, calls f(6, 7)
}
```

—*end example*] Declarations of a given nonmember function in different translation units need not specify the same default arguments. Declarations of a given member function in different translation units, however, shall specify the same default arguments (the accumulated sets of default arguments at the end of the translation units shall be the same).

5 Default argument expressions have their names bound and their types checked at the point of declaration. [*Example:* in the following code, `g` will be called with the value `f(1)`:

```
int a = 1;
int f(int);
int g(int x = f(a)); // default argument: f(::a)

void h() {
    a = 2;
    {
        int a = 3;
        g();        // g(f(::a))
    }
}
```

—*end example*]

6 In member function declarations, names in default argument expressions are looked up in the scope of the class like names in member function bodies (9.3). The default arguments in an out-of-line function definition are added to the set of default arguments provided by the member function declaration in the class definition. [*Example:*

```
class C {
        void f(int i = 3);
        void g(int i, int j = 99);
};

void C::f(int i = 3) // error: default argument already
{ }                  // specified in class scope
void C::g(int i = 88, int j) // in this translation unit,
{ }                          // C::g can be called with no argument
```

—*end example*]

7 Local variables shall not be used in default argument expressions. [*Example:*

```
void f()
{
    int i;
    extern void g(int x = i);   // error
    // ...
}
```

—*end example*]

8 The keyword `this` shall not be used in a default argument of a member function. [*Example:*

```
class A {
    void f(A* p = this) { }     // error
};
```

—*end example*]

9 Default arguments are evaluated at each point of call before entry into a function. The order of evaluation of function arguments is implementation-defined. Consequently, parameters of a function shall not be used in default argument expressions, even if they are not evaluated. Parameters of a function declared before a default argument expression are in scope and can hide namespace and class member names. [*Example:*

```
int a;
int f(int a, int b = a);    // error: parameter ‘a’
                            // used as default argument
typedef int I;
int g(float I, int b = I(2)); // error: parameter ‘I’ found
int h(int a, int b = sizeof(a));  // error, parameter ‘a’ used
                                  // in default argument
```

—*end example*] Similarly, a nonstatic member shall not be used in a default argument expression, even if it is not evaluated, unless it appears as the id-expression of a class member access expression (5.2.4) or unless it is used to form a pointer to member (5.3.1). [*Example:* the declaration of `X::mem1()` in the following example is ill-formed because no object is supplied for the nonstatic member `X::a` used as an initializer.

```
int b;
class X {
    int a;
    int mem1(int i = a); // error: nonstatic member ‘a’
                         // used as default argument
    int mem2(int i = b); // ok;  use X::b
    static b;
};
```

The declaration of `X::mem2()` is meaningful, however, since no object is needed to access the static member `X::b`. Classes, objects, and members are described in 9.] A default argument is not part of the type of a function. [*Example:*

```
int f(int = 0);

void h()
{
    int j = f(1);
    int k = f();      // fine, means f(0)
}

int (*p1)(int) = &f;
int (*p2)() = &f;     // error: type mismatch
```

—*end example*] When a declaration of a function is introduced by way of a `using` declaration (7.3.3), any default argument information associated with the declaration is imported as well.

10 A virtual function call (10.3) uses the default arguments in the declaration of the virtual function determined by the static type of the pointer or reference denoting the object. An overriding function in a derived class does not acquire default arguments from the function it overrides. [*Example:*

```
struct A {
    virtual void f(int a = 7);
};
struct B : public A {
    void f(int a);
};
void m()
{
    B* pb = new B;
    A* pa = pb;
    pa->f();          // ok, calls pa->A::f(7)
    pb->f();          // error: wrong number of arguments for B::f()
}
```

—*end example*]

8.4 Function definitions [dcl.fct.def]

1 Function definitions have the form

function-definition:
  decl-specifier-seq$_{opt}$ *declarator* *ctor-initializer*$_{opt}$ *function-body*
  decl-specifier-seq$_{opt}$ *declarator* *function-try-block*

function-body:
  compound-statement

The *declarator* in a *function-definition* shall have the form

`D1 (` *parameter-declaration-clause* `)` *cv-qualifier-seq*$_{opt}$ *exception-specification*$_{opt}$

as described in 8.3.5. A function shall be defined only in namespace or class scope.

2 The parameters are in the scope of the outermost block of the *function-body*.

3 [*Example:* a simple example of a complete function definition is

```
int max(int a, int b, int c)
{
    int m = (a > b) ? a : b;
    return (m > c) ? m : c;
}
```

Here `int` is the *decl-specifier-seq*; `max(int a, int b, int c)` is the *declarator*; `{ /* ... */ }` is the *function-body*.]

4 A *ctor-initializer* is used only in a constructor; see 12.1 and 12.6.

5 A *cv-qualifier-seq* can be part of a non-static member function declaration, non-static member function definition, or pointer to member function only; see 9.4.2. It is part of the function type.

6 [*Note:* unused parameters need not be named. For example,

```
void print(int a, int)
{
    printf("a = %d\n",a);
}
```

—*end note*]

8.5 Initializers [dcl.init]

1 A declarator can specify an initial value for the identifier being declared. The identifier designates an object or reference being initialized. The process of initialization described in the remainder of this subclause (8.5) applies also to initializations specified by other syntactic contexts, such as the initialization of function parameters with argument expressions (5.2.2) or the initialization of return values (6.6.3).

```
initializer:
        = initializer-clause
        ( expression-list )

initializer-clause:
        assignment-expression
        { initializer-list ,opt }
        { }

initializer-list:
        initializer-clause
        initializer-list , initializer-clause
```

2 Automatic, register, static, and external variables of namespace scope can be initialized by arbitrary expressions involving constants and previously declared variables and functions. [*Example:*

```
int f(int);
int a = 2;
int b = f(a);
int c(b);
```

—*end example*]

3 [*Note:* default argument expressions are more restricted; see 8.3.6.

4 The order of initialization of static objects is described in 3.6 and 6.7.]

5 To *zero-initialize* storage for an object of type `T` means:

— if `T` is a scalar or pointer-to-member type, the storage is set to the value of `0` (zero) converted to `T`;

— if `T` is a non-union class type, the storage for each nonstatic data member and each base-class subobject is zero-initialized;

— if `T` is a union type, the storage for its first nonstatic data member is zero-initialized;

— if `T` is an array type, the storage for each element is zero-initialized;

— if `T` is a reference type, no initialization is performed.

To *default-initialize* an object of type `T` means:

— if `T` is a non-POD class type, the default constructor for `T` is called (and the initialization is ill-formed if `T` has no accessible default constructor);

— if `T` is an array type, each element is default-initialized;

— otherwise, the storage for the object is zero-initialized.

Default-initialization uses the direct-initialization semantics described below.

6 The memory occupied by any object of static storage duration shall be zero-initialized. Furthermore, if no initializer is explicitly specified in the declaration of the object and the object is of non-POD class type (or array thereof), then default initialization shall be performed. If no *initializer* is specified for an object with automatic or dynamic storage duration, the object and its subobjects, if any, have an indeterminate initial value.[60)]

[60)] This does not apply to aggregate objects with automatic storage duration initialized with an incomplete brace-enclosed *initializer-list*; see 8.5.1.

7 An initializer for a static member is in the scope of the member's class. [*Example:*

```
int a;

struct X {
    static int a;
    static int b;
};

int X::a = 1;
int X::b = a;   // X::b = X::a
```

—*end example*]

8 The form of initialization (using parentheses or `=`) is generally insignificant, but does matter when the entity being initialized has a class type; see below. A parenthesized initializer can be a list of expressions only when the entity being initialized has a class type.

9 [*Note:* since `()` is not permitted by the syntax for *initializer*,

```
X a();
```

is not the declaration of an object of class `X`, but the declaration of a function taking no argument and returning an `X`. The form `()` is permitted in certain other initialization contexts (5.3.4, 5.2.3, 12.6.2).]

10 The initialization that occurs in argument passing, function return, and brace-enclosed initializer lists (8.5.1) is called *copy-initialization* and is equivalent to the form

```
T x = a;
```

The initialization that occurs in `new` expressions (5.3.4), `static_cast` expressions (5.2.8), functional notation type conversions (5.2.3), and base and member initializers (12.6.2) is called *direct-initialization* and is equivalent to the form

```
T x(a);
```

11 The semantics of initializers are as follows. The *destination type* is the type of the object or reference being initialized and the *source type* is the type of the initializer expression. The source type is not defined when the initializer is brace-enclosed or when it is a parenthesized list of expressions.

- — If the destination type is a reference type, see 8.5.3.
- — If the destination type is an array of characters or an array of `wchar_t`, and the initializer is a string literal, see 8.5.2.
- — Otherwise, if the destination type is an array, see 8.5.1.
- — If the destination type is a (possibly cv-qualified) class type:
  - — If the class is an aggregate (8.5.1), and the initializer is a brace-enclosed list, see 8.5.1.
  - — If the initialization is direct-initialization, or if it is copy-initialization where the cv-unqualified version of the source type is the same class as, or a derived class of, the class of the destination, constructors are considered. The applicable constructors are enumerated (13.3.1.4), and the best one is chosen through overload resolution (13.3). The constructor so selected is called to initialize the object, with the initializer expression(s) as its argument(s). If no constructor applies, or the overload resolution is ambiguous, the initialization is ill-formed.
  - — Otherwise (i.e., for the remaining copy-initialization cases), a temporary of the destination type is created. User-defined conversions that can convert from the source type to the destination type are enumerated (13.3.1.3), and the best one is chosen through overload resolution (13.3). The user-defined conversion so selected is called to convert the initializer expression into the temporary. If the conversion cannot be done or is ambiguous, the initialization is ill-formed. The object being initialized is then direct-initialized from the temporary according to the rules above.[61] In certain cases,

[61] Because the type of the temporary is the same as the type of the object being initialized, this direct-initialization, if well-formed, will use a copy constructor (12.8) to copy the temporary.

an implementation is permitted to eliminate the temporary by initializing the object directly; see 12.2.

— Otherwise, if the source type is a (possibly cv-qualified) class type, conversion functions are considered. The applicable conversion functions are enumerated (13.3.1.3), and the best one is chosen through overload resolution (13.3). The user-defined conversion so selected is called to convert the initializer expression into the object being initialized. If the conversion cannot be done or is ambiguous, the initialization is ill-formed.

— Otherwise, the initial value of the object being initialized is the (possibly converted) value of the initializer expression. Standard conversions (clause 4) will be used, if necessary, to convert the initializer expression to the cv-unqualified version of the destination type; no user-defined conversions are considered. If the conversion cannot be done, the initialization is ill-formed. [*Note:* an expression of type "*cv1* T" can initialize an object of type "*cv2* T" independently of the cv-qualifiers *cv1* and *cv2*.

```
int a;
const int b = a;
int c = b;
```

—*end note*]

12 If T is a scalar type, then a declaration of the form

```
T x = { a };
```

is equivalent to

```
T x = a;
```

8.5.1 Aggregates [dcl.init.aggr]

1 An *aggregate* is an array or a class (9) with no user-declared constructors (12.1), no private or protected non-static data members (11), no non-static members of reference type, no non-static `const` members, no base classes (10), and no virtual functions (10.3).[62)]

2 When an aggregate is initialized the *initializer* can be an *initializer-clause* consisting of a brace-enclosed, comma-separated list of *initializer*s for the members of the aggregate, written in increasing subscript or member order. If the aggregate contains subaggregates, this rule applies recursively to the members of the subaggregate. [*Example:*

```
struct A {
        int x;
        struct B {
                int i;
                int j;
        } b;
} a = { 1, { 2, 3 } };
```

initializes `a.x` with 1, `a.b.i` with 2, `a.b.j` with 3.]

3 An aggregate that is a class can also be initialized with a single expression not enclosed in braces, as described in 8.5.

4 An array of unknown size initialized with a brace-enclosed *initializer-list* containing `n` *initializer*s, where `n` shall be greater than zero, is defined as having `n` elements (8.3.4). [*Example:*

```
int x[] = { 1, 3, 5 };
```

declares and initializes `x` as a one-dimensional array that has three elements since no size was specified and there are three initializers.] An empty initializer list `{}` shall not be used as the initializer for an array of unknown bound.[62)]

62) The syntax provides for empty *initializer-list*s, but nonetheless C++ does not have zero length arrays.

5 Static data members are not considered members of the class for purposes of aggregate initialization. [*Example:*

```
struct A {
        int i;
        static int s;
        int j;
} a = { 1, 2 };
```

Here, the second initializer 2 initializes `a.j` and not the static data member `A::s`.]

6 An *initializer-list* is ill-formed if the number of *initializer*s exceeds the number of members or elements to initialize. [*Example:*

```
char cv[4] = { 'a', 's', 'd', 'f', 0 };  // error
```

is ill-formed.]

7 If there are fewer *initializer*s in the list than there are members in the aggregate, then each member not explicitly initialized shall be initialized with a value of the form `T()` (5.2.3), where `T` represents the type of the uninitialized member. [*Example:*

```
struct S { int a; char* b; int c; };
S ss = { 1, "asdf" };
```

initializes `ss.a` with 1, `ss.b` with `"asdf"`, and `ss.c` with the value of an expression of the form `int()`, that is, `0`.]

8 An *initializer* for an aggregate member that is an empty class shall have the form of an empty *initializer-list* `{}.` [*Example:*

```
struct S { };
struct A {
        S s;
        int i;
} a = { { } , 3 };
```

—*end example*] An empty initializer-list can be used to initialize any aggregate. If the aggregate is not an empty class, then each member of the aggregate shall be initialized with a value of the form `T()` (5.2.3), where `T` represents the type of the uninitialized member.

9 When initializing a multi-dimensional array, the *initializer*s initialize the elements with the last (rightmost) index of the array varying the fastest (8.3.4). [*Example:*

```
float y[4][3] = {
    { 1 }, { 2 }, { 3 }, { 4 }
};
```

initializes the first column of `y` (regarded as a two-dimensional array) and leaves the rest zero.]

10 Braces can be elided in an *initializer-list* as follows. If the *initializer-list* begins with a left brace, then the succeeding comma-separated list of *initializer*s initializes the members of a subaggregate; it is erroneous for there to be more initializers than members. If, however, the *initializer-list* for a subaggregate does not begin with a left brace, then only enough *initializer*s from the list are taken to initialize the members of the subaggregate; any remaining *initializer*s are left to initialize the next member of the aggregate of which the current subaggregate is a member. [*Example:*

```
float y[4][3] = {
    { 1, 3, 5 },
    { 2, 4, 6 },
    { 3, 5, 7 },
};
```

is a completely-braced initialization: 1, 3, and 5 initialize the first row of the array `y[0]`, namely `y[0][0]`, `y[0][1]`, and `y[0][2]`. Likewise the next two lines initialize `y[1]` and `y[2]`. The initializer ends early and therefore `y[3]`'s elements are initialized as if explicitly initialized with an expression of the form `float()`, that is, are initialized with `0.0`. In the following example, braces in the *initializer-list* are elided; however the *initializer-list* has the same effect as the completely-braced *initializer-list* of the above example,

```
float y[4][3] = {
    1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

The initializer for `y` begins with a left brace, but the one for `y[0]` does not, therefore three elements from the list are used. Likewise the next three are taken successively for `y[1]` and `y[2]`. —*end example*]

11 All type conversions (13.3.1.3) are considered when initializing the aggregate member with an initializer from an *initializer-list*. If the *initializer* can initialize a member, the member is initialized. Otherwise, if the member is itself a non-empty subaggregate, brace elision is assumed and the *initializer* is considered for the initialization of the first member of the subaggregate. [*Example:*

```
struct A {
    int i;
    operator int();
};
struct B {
        A a1, a2;
        int z;
};
A a;
B b = { 4, a, a };
```

Braces are elided around the *initializer* for `b.a1.i`. `b.a1.i` is initialized with 4, `b.a2` is initialized with `a`, `b.z` is initialized with whatever `a.operator int()` returns.]

12 [*Note:* An aggregate array or an aggregate class may contain members of a class type with a user-declared constructor (12.1). Initialization of these aggregate objects is described in 12.6.1.]

13 When an aggregate is initialized with a brace-enclosed *initializer-list*, if some members are initialized with constant expressions and other members are initialized with non-constant expressions, it is unspecified whether the initialization of members with constant expressions takes place during the static phase or during the dynamic phase of initialization (3.6.2).

14 The initializer for a union with no user-declared constructor is either a single expression of the same type, or a brace-enclosed initializer for the first member of the union. [*Example:*

```
union u { int a; char* b; };

u a = { 1 };
u b = a;
u c = 1;              // error
u d = { 0, "asdf" };  // error
u e = { "asdf" };     // error
```

—*end example*]

8.5.2 Character arrays [dcl.init.string]

1 A `char` array (whether plain `char`, `signed`, or `unsigned`) can be initialized by a string; a `wchar_t` array can be initialized by a wide string literal; successive characters of the string initialize the members of the array. [*Example:*

```
char msg[] = "Syntax error on line %s\n";
```

shows a character array whose members are initialized with a string. Note that because `'\n'` is a single character and because a trailing `'\0'` is appended, `sizeof(msg)` is `25`.]

2 There shall not be more initializers than there are array elements. [*Example:*

```
char cv[4] = "asdf";  // error
```

is ill-formed since there is no space for the implied trailing `'\0'`.]

8.5.3 References [dcl.init.ref]

1 A variable declared to be a `T&`, that is "reference to type `T`" (8.3.2), shall be initialized by an object, or function, of type `T` or by an object that can be converted into a `T`. [*Example:*

```
int g(int);
void f()
{
    int i;
    int& r = i;  // 'r' refers to 'i'
    r = 1;       // the value of 'i' becomes 1
    int* p = &r; // 'p' points to 'i'
    int& rr = r; // 'rr' refers to what 'r' refers to,
                 // that is, to 'i'
    int (&rg)(int) = g; // 'rg' refers to the function 'g'
    rg(i);              // calls function 'g'
    int a[3];
    int (&ra)[3] = a;   // 'ra' refers to the array 'a'
    ra[1] = i;          // modifies 'a[1]'
}
```

—*end example*]

2 A reference cannot be changed to refer to another object after initialization. Note that initialization of a reference is treated very differently from assignment to it. Argument passing (5.2.2) and function value return (6.6.3) are initializations.

3 The initializer can be omitted for a reference only in a parameter declaration (8.3.5), in the declaration of a function return type, in the declaration of a class member within its class declaration (9.2), and where the `extern` specifier is explicitly used. [*Example:*

```
int& r1;         // error: initializer missing
extern int& r2;  // ok
```

—*end example*]

4 Given types "*cv1* `T1`" and "*cv2* `T2`," "*cv1* `T1`" is *reference-related* to "*cv2* `T2`" if `T1` is the same type as `T2`, or `T1` is a base class of `T2`. "*cv1* `T1`" is *reference-compatible* with "*cv2* `T2`" if `T1` is reference-related to `T2` and *cv1* is the same cv-qualification as, or greater cv-qualification than, *cv2*. For purposes of overload resolution, cases for which *cv1* is greater cv-qualification than *cv2* are identified as *reference-compatible with added qualification* (see 13.3.3.2). In all cases where the reference-related or reference-compatible relationship of two types is used to establish the validity of a reference binding, and `T1` is a base class of `T2`, a program that necessitates such a binding is ill-formed if `T1` is an inaccessible (11) or ambiguous (10.2) base class of `T2`.

5 A reference to type "*cv1* `T1`" is initialized by an expression of type "*cv2* `T2`" as follows:

— If the initializer expression is an lvalue (but not an lvalue for a bit-field), and

6

  — "*cv1* `T1`" is reference-compatible with "*cv2* `T2`," or

  — the initializer expression can be implicitly converted to an lvalue of type "cv3 `T1`," where *cv3* is the same cv-qualification as, or lesser cv-qualification than, *cv1*, [63] then

[63] This requires a conversion function (12.3.2) returning a reference type, and therefore applies only when `T2` is a class type.

7 the reference is bound directly to the initializer expression lvalue. [*Note:* the usual lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) standard conversions are not needed, and therefore are suppressed, when such direct bindings to lvalues are done.] [*Example:*

```
double d = 2.0;
double& rd = d;         // rd refers to ‘d’
const double& rcd = d;  // rcd refers to ‘d’

struct A { };
struct B : public A { } b;
A& ra = b;              // ra refers to A sub-object in ‘b’
const A& rca = b;       // rca refers to A sub-object in ‘b’
```

—*end example*]

8 — Otherwise, the reference shall be to a non-volatile const type (i.e., *cv1* shall be `const`). [*Example:*

```
double& rd2 = 2.0;      // error: not an lvalue and reference
                        //   not const
int  i = 2;
double& rd3 = i;        // error: type mismatch and reference
                        //   not const
```

—*end example*]

— If the initializer expression is an rvalue, with `T2` a class type, and “*cv1* `T1`” is reference-compatible with “*cv2* `T2`,” the reference is bound in one of the following ways (the choice is implementation-defined):

— The reference is bound directly to the object represented by the rvalue (see 3.10) or to a sub-object within that object.

— A temporary of type “*cv1* `T2`” [sic] is created, and a copy constructor is called to copy the entire rvalue object into the temporary. The reference is bound to the temporary or to a sub-object within the temporary.[64)]

9 The appropriate copy constructor must be callable whether or not the copy is actually done. [*Example:*

```
struct A { };
struct B : public A { } b;
extern B f();
const A& rca = f();     // Either bound directly or
                        //   the entire B object is copied and
                        //   the reference is bound to the
                        //   A sub-object of the copy
```

—*end example*]

10 — Otherwise, a temporary of type “*cv1* `T1`” is created and initialized from the initializer expression using the rules for a non-reference initialization (8.5). The reference is then bound to the temporary. If `T1` is reference-related to `T2`, *cv1* must be the same cv-qualification as, or greater cv-qualification than, *cv2*; otherwise, the program is ill-formed. [*Example:*

```
const double& rcd2 = 2; // rcd2 refers to temporary
                        // with value ‘2.0’
const volatile int cvi = 1;
const int& r = cvi;     // error: type qualifiers dropped
```

—*end example*]

64) Clearly, if the reference initialization being processed is one for the first argument of a copy constructor call, an implementation must eventually choose the direct-binding alternative to avoid infinite recursion.

11 [*Note:* 12.2 describes the lifetime of temporaries bound to references.]

9 Classes [class]

1 A class is a type. Its name becomes a *class-name* (9.1) within its scope.

> *class-name:*
>   *identifier*
>   *template-id*

*Class-specifier*s and *elaborated-type-specifier*s (7.1.5.3) are used to make *class-name*s. An object of a class consists of a (possibly empty) sequence of members and base class objects.

> *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`

2 A *class-name* is inserted into the scope in which it is declared and into the scope of the class itself. The name of a class can be used as a *class-name* even within the *base-clause* and *member-specification* of the *class-specifier* itself. For purposes of access checking, the inserted class name is treated as if it were a public member name. A *class-specifier* is commonly referred to as a class definition. A class is considered defined after the closing brace of its *class-specifier* has been seen even though its member functions are in general not yet defined.

3 A class with an empty sequence of members and base class objects is an *empty* class. Objects of an empty class have a nonzero size. [*Note:* Class objects can be assigned, passed as arguments to functions, and returned by functions (except objects of classes for which copying has been restricted; see 12.8). Other plausible operators, such as equality comparison, can be defined by the user; see 13.5.]

4 A *structure* is a class declared with the *class-key* `struct`; its members and base classes (10) are public by default (11). A *union* is a class declared with the *class-key* `union`; its members are public by default and it holds only one member at a time (9.6). [*Note:* Aggregates of class type are described in 8.5.1.] A *POD-struct*[65)] is an aggregate class that has no members of type reference, pointer to member, non-POD-struct or non-POD-union. Similarly, a *POD-union* is an aggregate union that has no members of type reference, pointer to member, non-POD-struct or non-POD-union.

9.1 Class names [class.name]

1 A class definition introduces a new type. [*Example:*

65) The acronym POD stands for "plain ol' data."

```
struct X { int a; };
struct Y { int a; };
X a1;
Y a2;
int a3;
```

declares three variables of three different types. This implies that

```
a1 = a2;        // error: Y assigned to X
a1 = a3;        // error: int assigned to X
```

are type mismatches, and that

```
int f(X);
int f(Y);
```

declare an overloaded (13) function `f()` and not simply a single function `f()` twice. For the same reason,

```
struct S { int a; };
struct S { int a; };  // error, double definition
```

is ill-formed because it defines `S` twice.]

2 A class definition introduces the class name into the scope where it is defined and hides any class, object, function, or other declaration of that name in an enclosing scope (3.3). If a class name is declared in a scope where an object, function, or enumerator of the same name is also declared, then when both declarations are in scope, the class can be referred to only using an *elaborated-type-specifier* (7.1.5.3). [*Example:*

```
struct stat {
    // ...
};

stat gstat;             // use plain ‘stat’ to
                        // define variable

int stat(struct stat*); // redefine ‘stat’ as function

void f()
{
    struct stat* ps;    // ‘struct’ prefix needed
                        // to name struct stat
    // ...
    stat(ps);           // call stat()
    // ...
}
```

—*end example*] A *declaration* consisting solely of *class-key identifier ;* is either a redeclaration of the name in the current scope or a forward declaration of the identifier as a class name. It introduces the class name into the current scope. [*Example:*

```
struct s { int a; };

void g()
{
    struct s;               // hide global struct ‘s’
    s* p;                   // refer to local struct ‘s’
    struct s { char* p; };  // declare local struct ‘s’
    struct s;               // receclaration, has no effect
}
```

—*end example*] [*Note:* Such declarations allow definition of classes that refer to each other. [*Example:*

```
class Vector;

class Matrix {
    // ...
    friend Vector operator*(Matrix&, Vector&);
};

class Vector {
    // ...
    friend Vector operator*(Matrix&, Vector&);
};
```

Declaration of `friend`s is described in 11.4, operator functions in 13.5.]]

3 An *elaborated-type-specifier* (7.1.5.3) can also be used in the declarations of objects and functions. It differs from a class declaration in that if a class of the elaborated name is in scope the elaborated name will refer to it. [*Example:*

```
struct s { int a; };

void g(int s)
{
    struct s* p = new struct s;    // global 's'
    p->a = s;                      // local 's'
}
```

—*end example*]

4 [*Note:* A name declaration takes effect immediately after the *identifier* is seen. For example,

```
class A * A;
```

first specifies `A` to be the name of a class and then redefines it as the name of a pointer to an object of that class. This means that the elaborated form `class A` must be used to refer to the class. Such artistry with names can be confusing and is best avoided.]

5 A *typedef-name* (7.1.3) that names a class is a *class-name*, but shall not be used in an *elaborated-type-specifier*; see also 7.1.3.

9.2 Class members [class.mem]

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*

1 The *member-specification* in a class definition declares the full set of members of the class; no member can be added elsewhere. Members of a class are data members, member functions (9.4), nested types, and member constants. Data members and member functions are static or nonstatic; see 9.5. Nested types are classes (9.1, 9.8) and enumerations (7.2) defined in the class, and arbitrary types declared as members by use of a typedef declaration (7.1.3). The enumerators of an enumeration (7.2) defined in the class are member constants of the class. Except when used to declare friends (11.4) or to adjust the access to a member of a base class (11.3), *member-declaration*s declare members of the class, and each such *member-declaration* shall declare at least one member name of the class. A member shall not be declared twice in the *member-specification*, except that a nested class can be declared and then later defined.

2 [*Note:* a single name can denote several function members provided their types are sufficiently different (13).]

3 A *member-declarator* can contain a *constant-initializer* only if it declares a `static` member (9.5) of integral or enumeration type, see 9.5.2.

4 A member can be initialized using a constructor; see 12.1.

5 A member shall not be `auto`, `extern`, or `register`.

6 The *decl-specifier-seq* can be omitted in constructor, destructor, and conversion function declarations only. The *member-declarator-list* can be omitted only after a *class-specifier*, an *enum-specifier*, or a *decl-specifier-seq* of the form `friend` *elaborated-type-specifier*. A *pure-specifier* shall be used only in the declaration of a virtual function (10.3).

7 Non-`static` (9.5) members that are class objects shall be objects of previously defined classes. In particular, a class `cl` shall not contain an object of class `cl`, but it can contain a pointer or reference to an object of class `cl`. When an array is used as the type of a nonstatic member all dimensions shall be specified.

8 Except when used to form a pointer to member (5.3.1), when used in the body of a nonstatic member function of its class or of a class derived from its class (9.4.1), or when used in a *mem-initializer* for a constructor for its class or for a class derived from its class (12.6.2), a nonstatic nontype member of a class shall only be referred to with the class member access syntax (5.2.4).

9 [*Example:* A simple example of a class definition is

```
struct tnode {
    char tword[20];
    int count;
    tnode *left;
    tnode *right;
};
```

which contains an array of twenty characters, an integer, and two pointers to similar structures. Once this definition has been given, the declaration

```
tnode s, *sp;
```

declares `s` to be a `tnode` and `sp` to be a pointer to a `tnode`. With these declarations, `sp->count` refers to the `count` member of the structure to which `sp` points; `s.left` refers to the `left` subtree pointer of the structure `s`; and `s.right->tword[0]` refers to the initial character of the `tword` member of the `right` subtree of `s`.]

10 The type of a nonstatic data member is data member type, not object type; the type of a nonstatic member function is member function type, not function type; see 5.3.1 and 9.4. [*Example:* the type of the *qualified-id* expression `tnode::count` is data member type and the type of `&tnode::count` is pointer to data member (that is, `int (tnode::*)`; see 5.3.1).] [*Note:* the type of `static` members is described in 9.5.]

11 Nonstatic data members of a (non-union) class declared without an intervening *access-specifier* are allocated so that later members have higher addresses within a class object. The order of allocation of nonstatic data members separated by an *access-specifier* is implementation-defined (11.1). Implementation alignment requirements might cause two adjacent members not to be allocated immediately after each other; so might requirements for space for managing virtual functions (10.3) and virtual base classes (10.1); see also 5.4. [*Note:* a constructor (12.1) is a function member (9.4) that is declared using the same name as its class.]

12 A static data member, enumerator, member of an anonymous union, or nested type shall not have the same name as its class.

13 Two POD-struct (9) types are layout-compatible if they have the same number of members, and corresponding members (in order) have layout-compatible types (3.9).

14 Two POD-union (9) types are layout-compatible if they have the same number of members, and corresponding members (in any order) have layout-compatible types (3.9).

15 If a POD-union contains two or more POD-structs that share a common initial sequence, and if the POD-union object currently contains one of these POD-structs, it is permitted to inspect the common initial part of any of them. Two POD-structs share a common initial sequence if corresponding members have layout-compatible types (and, for bit-fields, the same widths) for a sequence of one or more initial members.

16 A pointer to a POD-struct object, suitably converted, points to its initial member (or if that member is a bit-field, then to the unit in which it resides) and vice versa. [*Note:* There might therefore be unnamed padding within a POD-struct object, but not at its beginning, as necessary to achieve appropriate alignment.]

9.3 Scope rules for classes [class.scope0]

1 The following rules describe the scope of names declared in classes.

1) The scope of a name declared in a class consists not only of the declarative region (3.3.5) following the name's declarator, but also of all function bodies, default arguments, and constructor initializers in that class (including such things in nested classes).

2) A name `N` used in a class `S` shall refer to the same declaration when re-evaluated in its context and in the completed scope of S.

3) If reordering member declarations in a class yields an alternate valid program under (1) and (2), the program's behavior is undefined.

4) A declaration in a nested declarative region hides a declaration whose declarative region contains the nested declarative region.

5) A declaration within a member function hides a declaration whose scope extends to or past the end of the member function's class.

6) The scope of a declaration that extends to or past the end of a class definition also extends to the regions defined by its member definitions, even if defined lexically outside the class (this includes static data member initializations, nested class definitions and member function definitions (that is, the *parameter-declaration-clause* including default arguments (8.3.6), the member function body and, for constructor functions (12.1), the ctor-initializer (12.6.2)). [*Example:*

```
typedef int  c;
enum { i = 1 };
```

```
class X {
    char  v[i];  // error: 'i' refers to ::i
                 // but when reevaluated is X::i
    int  f() { return sizeof(c); }  // okay: X::c
    char  c;
    enum { i = 2 };
};

typedef char*  T;
struct Y {
    T  a;    // error: 'T' refers to ::T
             // but when reevaluated is Y::T
    typedef long  T;
    T  b;
};

struct Z {
    int  f(const R);  // error: 'R' is parameter name
                      // but swapping the two declarations
                      // changes it to a type
    typedef int  R;
};
```

—*end example*]

9.4 Member functions [class.mfct]

1 Functions declared in the definition of a class, excluding those declared with a `friend` specifier (11.4), are called member functions of that class. A member function may be declared `static` in which case it is a *static* member function of its class (9.5); otherwise it is a *nonstatic* member function of its class (9.4.1, 9.4.2).

2 A member function may be defined (8.4) in its class definition, in which case it is an *inline* member function, or it may be defined outside of its class definition if it has already been declared but not defined in its class definition. This *out-of-line* definition shall appear in a namespace scope enclosing the definition of the member function's class. Except for the out-of-line definition of a member function, and except for the out-of-line declaration of an explicit specialization of a template member function (14.5), a member function shall not be redeclared.

3 An `inline` member function (whether static or nonstatic) may also be defined outside of its class definition provided either its declaration in the class definition or its definition outside of the class definition declares the function as `inline` (7.1.2). [*Note:* Member functions of a class in namespace scope have external linkage. Member functions of a local class (9.9) have no linkage. See 3.5.]

4 There shall be at most one definition of a non-inline member function in a program; no diagnostic is required. There may be more than one `inline` member function definition in a program. See 3.2 and 7.1.2.

5 If the definition of a member function is lexically outside its class definition, the member function name shall be qualified by its class name using the `::` operator. A member function definition (that is, the *parameter-declaration-clause* including the default arguments (8.3.6), the member function body and, for a constructor function (12.1), the ctor-initializer (12.6.2)) is in the scope of the member function's class (9.3). [*Example:*

```
struct X {
        typedef int T;
        static T count;
        void f(T);
};
void X::f(T t = count) { }
```

The member function `f` of class `X` is defined in global scope; the notation `X::f` specifies that the function `f` is a member of class `X` and in the scope of class `X`. In the function definition, the parameter type `T` refers to the typedef member `T` declared in class `X` and the default argument `count` refers to the static data member `count` declared in class `X`.]

6 A `static` local variable in a member function always refers to the same object, whether or not the member function is `inline`.

7 Member functions may be mentioned in `friend` declarations after their class has been defined.

8 Member functions of a local class shall be defined inline in their class definition, if they are defined at all.

9.4.1 Nonstatic member functions [class.mfct.nonstatic]

1 A *nonstatic* member function may be called for an object of its class type, or for an object of a class derived (10) from its class type, using the class member access syntax (5.2.4, 13.3.1.1). A nonstatic member function may also be called directly using the function call syntax (5.2.2, 13.3.1.1)

— from within the body of a member function of its class or of a class derived from its class, or

— from a *mem-initializer* (12.6.2) for a constructor for its class or for a class derived from its class.

If a nonstatic member function of a class `X` is called for an object that is not of type `X`, or of a type derived from `X`, the behavior is undefined.

2 When an *id-expression* (5.1) that is not part of a class member access syntax (5.2.4) and not used to form a pointer to member (5.3.1) is used in the body of a nonstatic member function of class `X` or used in the *mem-initializer* for a constructor of class `X`, if name lookup (3.4) resolves the name in the *id-expression* to a nonstatic nontype member of class `X` or of a base class of `X`, the *id-expression* is transformed into a class member access expression (5.2.4) using `(*this)` (9.4.2) as the *postfix-expression* to the left of the `.` operator. The member name then refers to the member of the object for which the function is called. Similarly during name lookup, when an *unqualified-id* (5.1) used in the definition of a member function for class `X` resolves to a `static` member, an enumerator or a nested type of class `X` or of a base class of `X`, the *unqualified-id* is transformed into a *qualified-id* (5.1) in which the *nested-name-specifier* names the class of the member function. [*Example:*

```
struct tnode {
        char tword[20];
        int count;
        tnode *left;
        tnode *right;
        void set(char*, tnode* l, tnode* r);
};

void tnode::set(char* w, tnode* l, tnode* r)
{
        count = strlen(w)+1;
        if (sizeof(tword)<=count)
                error("tnode string too long");
        strcpy(tword,w);
        left = l;
        right = r;
}
```

```
void f(tnode n1, tnode n2)
{
        n1.set("abc",&n2,0);
        n2.set("def",0,0);
}
```

In the body of the member function `tnode::set`, the member names `tword`, `count`, `left`, and `right` refer to members of the object for which the function is called. Thus, in the call `n1.set("abc",&n2,0)`, `tword` refers to `n1.tword`, and in the call `n2.set("def",0,0)`, it refers to `n2.tword`. The functions `strlen`, `error`, and `strcpy` are not members of the class `tnode` and should be declared elsewhere.[66)]

3 The type of a nonstatic member function involves its class name; thus the type of the *qualified-id* expression `tnode::set` is member function type and the type of `&tnode::set` is pointer to member function (that is, `void (tnode::*)(char*,tnote*,tnode*)`, see 5.3.1).]

4 A nonstatic member function may be declared `const`, `volatile`, or `const volatile`. These *cv-qualifiers* affect the type of the `this` pointer (9.4.2). They also affect the type of the member function; a member function declared `const` is a *const* member function, a member function declared `volatile` is a *volatile* member function and a member function declared `const volatile` is a *const volatile* member function. [*Example:*

```
struct X {
        void g() const;
        void h() const volatile;
};
```

`X::g` is a `const` member function and `X::h` is a `const volatile` member function.]

5 A nonstatic member function may be declared *virtual* (10.3) or *pure virtual* (10.4).

9.4.2 The `this` pointer [class.this]

1 In the body of a nonstatic (9.4) member function, the keyword `this` is a non-lvalue expression whose value is the address of the object for which the function is called. The type of `this` in a member function of a class `X` is `X*`. If the member function is declared `const`, the type of `this` is `const X*`, if the member function is declared `volatile`, the type of `this` is `volatile X*`, and if the member function is declared `const volatile`, the type of `this` is `const volatile X*`.

2 In a `const` member function, the object for which the function is called is accessed through a `const` access path; therefore, a `const` member function shall not modify the object and its non-static data members. [*Example:*

```
struct s {
    int a;
    int f() const;
    int g() { return a++; }
    int h() const { return a++; } // error
};

int s::f() const { return a; }
```

The `a++` in the body of `s::h` is ill-formed because it tries to modify (a part of) the object for which `s::h()` is called. This is not allowed in a `const` member function where `this` is a pointer to `const`, that is, `*this` is a `const`.]

3 Similarly, `volatile` semantics (7.1.5.1) apply in `volatile` member functions when accessing the object and its non-static data members.

66) See, for example, `<cstring>` (21.2).

4 A *cv-qualified* member function can be called on an object-expression (5.2.4) only if the object-expression is as cv-qualified or less-cv-qualified than the member function. [*Example:*

```
void k(s& x, const s& y)
{
    x.f();
    x.g();
    y.f();
    y.g();      // error
}
```

The call `y.g()` is ill-formed because `y` is `const` and `s::g()` is a non-`const` member function, that is, `s::g()` is less-qualified than the object-expression y.]

5 Constructors (12.1) and destructors (12.4) shall not be declared `const`, `volatile` or `const volatile`. [*Note:* However, these functions can be invoked to create and destroy objects with cv-qualified types, see (12.1) and (12.4).]

9.5 Static members [class.static]

1 A data or function member of a class may be declared `static` in a class definition, in which case it is a *static member* of the class.

2 A `static` member `s` of class `X` may be referred to using the *qualified-id* expression `X::s`; it is not necessary to use the class member access syntax (5.2.4) to refer to a `static` member. A `static` member may be referred to using the class member access syntax, in which case the *object-expression* is always evaluated. [*Example:*

```
class process {
public:
        static void reschedule();
};
process& g();

void f()
{
        process::reschedule(); // ok: no object necessary
        g().reschedule();      // g() is called
}
```

—*end example*] A `static` member may be referred to directly in the scope of its class or in the scope of a class derived (10) from its class; in this case, the `static` member is referred to as if a *qualified-id* expression was used in which the *nested-name-specifier* names the class scope from which the static member is referred. [*Example:*

```
int g();
class X {
public:
        static int i;
        static int g();
};
int X::i = g(); // equivalent to X::g();
```

—*end example*]

3 The definition of a `static` member function or the *initializer* expression for a `static` data member may directly use the names of the `static` members, enumerators, and nested types of its class or of a base class of its class; during name lookup (3.4), when an *unqualified-id* (5.1) used in one of these contexts resolves to the declaration for one of these members, the *unqualified-id* is transformed into a *qualified-id* expression in which the *nested-name-specifier* names the class scope from which the the member is referred. The definition of a `static` member shall not use directly the names of the nonstatic members of its class or of a base class of its class (including as operands of the `sizeof` operator). The definition of a

`static` member may only refer to these members to form pointer to members (5.3.1) or with the class member access syntax (5.2.4).

4 Static members obey the usual class member access rules (11).

5 The type of a `static` member does not involve its class name. [*Example:* Thus, in the example above, the type of the *qualified-id* expression `X::g` is a function type and the type of `&X::g` is pointer to function type (that is, `void(*)()`, see 5.3.1).]

9.5.1 Static member functions [class.static.mfct]

1 [*Note:* the rules described in 9.4 apply to `static` member functions.]

2 [*Note:* a `static` member function does not have a `this` pointer (9.4.2).] A `static` member function shall not be `virtual`. There shall not be a `static` and a nonstatic member function with the same name and the same parameter types (13.1). A `static` member function shall not be declared `const`, `volatile`, or `const volatile`.

9.5.2 Static data members [class.static.data]

1 A `static` data member is not part of the subobjects of a class. There is only one copy of a `static` data member shared by all the objects of the class.

2 The declaration of a `static` data member in its class definition is not a definition and may be of an incomplete type other than cv-qualified `void`. A definition shall be provided for the `static` data member in a namespace scope enclosing the member's class definition. In the definition at namespace scope, the name of the `static` data member shall be qualified by its class name using the `::` operator. The *initializer* expression in the definition of a `static` data member is in the scope of its class (9.3). [*Example:*

```
class process {
        static process* run_chain;
        static process* running;
};

process* process::running = get_main();
process* process::run_chain = running;
```

The `static` data member `run_chain` of class `process` is defined in global scope; the notation `process::run_chain` specifies that the member `run_chain` is a member of class `process` and in the scope of class `process`. In the `static` data member definition, the *initializer* expression refers to the `static` data member `running` of class `process`.]

3 [*Note:* once the `static` data member has been defined, it exists even if no objects of its class have been created. [*Example:* in the example above, `run_chain` and `running` exist even if no objects of class `process` are created by the program.]]

4 If a `static` data member is of `const` integral or `const` enumeration type, its declaration in the class definition can specify a *constant-initializer* which shall be an integral constant expression (5.19). In that case, the member can appear in integral constant expressions within its scope. The member shall still be defined in a namespace scope and the definition of the member in namespace scope shall not contain an *initializer*.

5 There shall be exactly one definition of a `static` data member in a program; no diagnostic is required; see 3.2.

6 `Static` data members of a class in namespace scope have external linkage (3.5). A local class shall not have `static` data members.

7 `Static` data members are initialized and destroyed exactly like non-local objects (3.6.2, 3.6.3).

8 A `static` data member shall not be `mutable` (7.1.1).

9.6 Unions [class.union]

1 A union can be thought of as a class whose member objects all begin at offset zero and whose size is sufficient to contain any of its member objects. At most one of the member objects can be stored in a union at any time. A union can have member functions (including constructors and destructors), but not virtual (10.3) functions. A union shall not have base classes. A union shall not be used as a base class. An object of a class with a non-trivial default constructor (12.1), a non-trivial copy constructor (12.8), a non-trivial destructor (12.4), or a non-trivial copy assignment operator (13.5.3, 12.8) cannot be a member of a union, nor can array of such objects. A union can have no `static` data members.

2 A union of the form

`union {` *member-specification* `} ;`

is called an anonymous union; it defines an unnamed object (and not a type). The names of the members of an anonymous union shall be distinct from other names in the scope in which the union is declared; they are used directly in that scope without the usual member access syntax (5.2.4). [*Example:*

```
void f()
{
    union { int a; char* p; };
    a = 1;
    // ...
    p = "Jennifer";
    // ...
}
```

Here `a` and `p` are used like ordinary (nonmember) variables, but since they are union members they have the same address.]

3 Anonymous unions declared at namespace scope shall be declared `static`. All other anonymous unions shall not be declared `static`. An anonymous union shall not have `private` or `protected` members (11). An anonymous union shall not have function members.

4 A union for which objects or pointers are declared is not an anonymous union. [*Example:*

```
union { int aa; char* p; } obj, *ptr = &obj;
aa = 1;        // error
ptr->aa = 1;  // ok
```

The assignment to plain `aa` is ill formed since the member name is not visible outside the union, and even if it were visible, it is not associated with any particular object.] [*Note:* Initialization of unions with no user-declared constructors is described in (8.5.1).]

9.7 Bit-fields [class.bit]

1 A *member-declarator* of the form

identifier$_{opt}$ `:` *constant-expression*

specifies a bit-field; its length is set off from the bit-field name by a colon. Allocation of bit-fields within a class object is implementation-defined. Fields are packed into some addressable allocation unit. Fields straddle allocation units on some machines and not on others. Alignment of bit-fields is implementation-defined. Fields are assigned right-to-left on some machines, left-to-right on others.

2 An unnamed bit-field is useful for padding to conform to externally-imposed layouts. Unnamed fields are not members and cannot be initialized. As a special case, an unnamed bit-field with a width of zero specifies alignment of the next bit-field at an allocation unit boundary.

3 A bit-field shall not be a static member. A bit-field shall have integral or enumeration type (3.9.1). It is implementation-defined whether a plain (neither explicitly signed nor unsigned) `int` field is signed or unsigned. The address-of operator & shall not be applied to a bit-field, so there are no pointers to bit-fields.

Nor are there references to bit-fields.

9.8 Nested class declarations [class.nest]

1 A class can be defined within another class. A class defined within another is called a *nested* class. The name of a nested class is local to its enclosing class. The nested class is in the scope of its enclosing class. Except by using explicit pointers, references, and object names, declarations in a nested class can use only type names, static members, and enumerators from the enclosing class. [*Example:*

```
int x;
int y;

class enclose {
public:
    int x;
    static int s;

    class inner {

        void f(int i)
        {
            x = i;   // error: assign to enclose::x
            s = i;   // ok: assign to enclose::s
            ::x = i; // ok: assign to global x
            y = i;       // ok: assign to global y
        }

        void g(enclose* p, int i)
        {
            p->x = i;   // ok: assign to enclose::x
        }

    };
};

inner* p = 0;   // error ‘inner’ not in scope
```

—*end example*]

2 Member functions of a nested class have no special access to members of an enclosing class; they obey the usual access rules (11). Member functions of an enclosing class have no special access to members of a nested class; they obey the usual access rules. [*Example:*

```
class E {
    int x;

    class I {
        int y;
        void f(E* p, int i)
        {
            p->x = i;   // error: E::x is private
        }
    };

    int g(I* p)
    {
        return p->y;    // error: I::y is private
    }
};
```

—*end example*]

3 Member functions and static data members of a nested class can be defined in a namespace scope enclosing the definition of their class. [*Example:*

```
class enclose {
public:
    class inner {
        static int x;
        void f(int i);
    };
};

int enclose::inner::x = 1;

void enclose::inner::f(int i) { /* ... */ }
```

—*end example*] If class `X` is defined in a namespace scope a nested class `Y` may be declared in class `X` and later defined in the definition of class `X` or be later defined in a namespace scope enclosing the definition of class `X`. [*Example:*

```
class E {
    class I1;      // forward declaration of nested class
    class I2;
    class I1 {};  // definition of nested class
};
class E::I2 {};   // definition of nested class
```

—*end example*]

4 Like a member function, a friend function (11.4) defined within a nested class is in the lexical scope of that class; it obeys the same rules for name binding as a static member function of that class (9.5) and has no special access rights to members of an enclosing class.

9.9 Local class declarations [class.local]

1 A class can be defined within a function definition; such a class is called a *local* class. The name of a local class is local to its enclosing scope. The local class is in the scope of the enclosing scope. Declarations in a local class can use only type names, static variables, `extern` variables and functions, and enumerators from the enclosing scope. [*Example:*

```
int x;
void f()
{
    static int s ;
    int x;
    extern int g();

    struct local {
        int g() { return x; }    // error: ‘x’ is auto
        int h() { return s; }    // ok
        int k() { return ::x; }  // ok
        int l() { return g(); }  // ok
    };
    // ...
}

local* p = 0;   // error: ‘local’ not in scope
```

—*end example*]

2 An enclosing function has no special access to members of the local class; it obeys the usual access rules (11). Member functions of a local class shall be defined within their class definition, if they are defined at all.

3 If class `X` is a local class a nested class `Y` may be declared in class `X` and later defined in the definition of class `X` or be later defined in the same scope as the definition of class `X`. A local class shall not have static data members.

9.10 Nested type names [class.nested.type]

1 Type names obey exactly the same scope rules as other names. In particular, type names defined within a class definition cannot be used outside their class without qualification. [*Example:*

```
class X {
public:
    typedef int I;
    class Y { /* ... */ };
    I a;
};

I b;     // error
Y c;     // error
X::Y d;  // ok
X::I e;  // ok
```

—*end example*]

10 Derived classes [class.derived]

1 A list of base classes can be specified in a class definition using the notation:

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`

The *class-name* in a *base-specifier* shall denote a previously defined class (9), which is called a *direct base class* for the class being declared. The *base-specifier* is evaluated as a type.[67] A class `B` is a base class of a class `D` if it is a direct base class of `D` or a direct base class of one of `D`'s base classes. A class is an *indirect* base class of another if it is a base class but not a direct base class. A class is said to be (directly or indirectly) *derived* from its (direct or indirect) base classes. [*Note:* for the meaning of *access-specifier* see 11.] Unless redefined in the derived class, members of a base class can be referred to in expressions as if they were members of the derived class. The base class members are said to be *inherited* by the derived class. [*Note:* the scope resolution operator `::` (5.1) can be used to refer to a base member explicitly. This allows access to a name that has been redefined in the derived class. A derived class can itself serve as a base class subject to access control; see 11.2. A pointer to a derived class can be implicitly converted to a pointer to an accessible unambiguous base class (4.10). An lvalue of a derived class type can be bound to a reference to an accessible unambiguous base class (8.5.3).]

2 The *base-specifier-list* specifies the type of the *base class subobjects* contained in an object of the derived class type. [*Example:*

```
class Base {
public:
    int a, b, c;
};

class Derived : public Base {
public:
    int b;
};
```

[67] If the name of the base is also being used to name a data member in the class, the lookup of the *base-specifier* finds the class type, not the data member.

```
class Derived2 : public Derived {
public:
    int c;
};
```

Here, an object of class `Derived2` will have a sub-object of class `Derived` which in turn will have a sub-object of class `Base`.]

3 The order in which the base class subobjects are allocated in the complete object is unspecified. [*Note:* a derived class and its base class sub-objects can be represented by a directed acyclic graph (DAG) where an arrow means "directly derived from." A DAG of sub-objects is often referred to as a "sub-object lattice."

$$\begin{tikzcd}
\texttt{Base} \\
\texttt{Derived} \arrow[u] \\
\texttt{Derived2} \arrow[u]
\end{tikzcd}$$

The arrows need not have a physical representation in memory.]

4 [*Note:* initialization of objects representing base classes can be specified in constructors; see 12.6.2.]

5 [*Note:* A base class subobject might have a layout (3.7) different from the layout of a complete object of the same type. A base class subobject might have a polymorphic behavior (12.7) different from the polymorphic behavior of a complete object of the same type.]

10.1 Multiple base classes [class.mi]

1 A class can be derived from any number of base classes. [*Note:* The use of more than one direct base class is often called multiple inheritance.] [*Example:*

```
class A { /* ... */ };
class B { /* ... */ };
class C { /* ... */ };
class D : public A, public B, public C { /* ... */ };
```

—*end example*]

2 The order of derivation is not significant except as specified by the semantics of initialization by constructor (12.6.2), cleanup (12.4), and storage layout (5.4, 9.2, 11.1).

3 A class shall not be specified as a direct base class of a derived class more than once but it can be an indirect base class more than once. [*Example:*

```
class B { /* ... */ };
class D : public B, public B { /* ... */ };  // ill-formed

class L { public: int next;  /* ... */ };
class A : public L { /* ... */ };
class B : public L { /* ... */ };
class C : public A, public B { void f(); /* ... */ };   // well-formed
```

—*end example*]

4 A base class specifier that does not contain the keyword `virtual`, specifies a *nonvirtual* base class. A base class specifier that contains the keyword `virtual`, specifies a *virtual* base class. For each distinct occurrence of a nonvirtual base class in the class lattice of the most derived class, the complete object shall contain a corresponding distinct base class subobject of that type. For each distinct base class that is specified virtual, the complete object shall contain a single base class subobject of that type. [*Example:* for an object of class type `C`, each distinct occurrence of a (non-virtual) base class `L` in the class lattice of `C` corresponds one-to-one with a distinct `L` subobject within the object of type `C`. Given the class `C` defined above, an object of class `C` will have two sub-objects of class `L` as shown below.

$$\begin{tikzcd}
L & & L \\
A \arrow[u] & & B \arrow[u] \\
& C \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

In such lattices, explicit qualification can be used to specify which subobject is meant. The body of function `C::f` could refer to the member `next` of each `L` subobject:

```
void C::f() { A::next = B::next; }   // well-formed
```

Without the `A::` or `B::` qualifiers, the definition of `C::f` above would be ill-formed because of ambiguity (10.2).

5 For another example,

```
class V { /* ... */ };
class A : virtual public V { /* ... */ };
class B : virtual public V { /* ... */ };
class C : public A, public B { /* ... */ };
```

for an object `c` of class type `C`, a single subobject of type `V` is shared by every base subobject of `c` that is declared to have a `virtual` base class of type `V`. Given the class `C` defined above, an object of class `C` will have one subobject of class `V`, as shown below.

$$\begin{tikzcd}
& V & \\
A \arrow[ur] & & B \arrow[ul] \\
& C \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

6 A class can have both virtual and nonvirtual base classes of a given type.

```
class B { /* ... */ };
class X : virtual public B { /* ... */ };
class Y : virtual public B { /* ... */ };
class Z : public B { /* ... */ };
class AA : public X, public Y, public Z { /* ... */ };
```

For an object of class `AA`, all `virtual` occurrences of base class `B` in the class lattice of `AA` correspond to a single `B` subobject within the object of type `AA`, and every other occurrence of a (non-virtual) base class `B` in the class lattice of `AA` corresponds one-to-one with a distinct `B` subobject within the object of type `AA`. Given the class `AA` defined above, class `AA` has two sub-objects of class `B`: `Z`'s `B` and the virtual `B` shared by `X` and `Y`, as shown below.

$$\begin{tikzcd}
& B & & B \\
X \arrow[ur] & & Y \arrow[ul] & Z \arrow[u] \\
& AA \arrow[ul] \arrow[ur] \arrow[urr] & &
\end{tikzcd}$$

—*end example*]

10.2 Member name lookup [class.member.lookup]

1 Member name lookup determines the meaning of a name (*id-expression*) in a class scope (9.3). Name lookup can result in an *ambiguity*, in which case the program is ill-formed. For an *id-expression*, name lookup begins in the class scope of `this`; for a *qualified-id*, name lookup begins in the scope of the *nested-name-specifier*. Name lookup takes place before access control (3.4, 11).

2 The following steps define the result of name lookup in a class scope. First, we consider every declaration for the name in the class and in each of its base class sub-objects. A member name `f` in one sub-object `B` *hides* a member name `f` in a sub-object `A` if `A` is a base class sub-object of `B`. We eliminate from consideration any declarations that are so hidden. If the resulting set of declarations are not all from sub-objects of the same type, or the set has a nonstatic member and includes members from distinct sub-objects, there is an ambiguity and the program is ill-formed. Otherwise that set is the result of the lookup.

3 [*Example:*

```
class A {
public:
    int a;
    int (*b)();
    int f();
    int f(int);
    int g();
};

class B {
    int a;
    int b();
public:
    int f();
    int g;
    int h();
    int h(int);
};

class C : public A, public B {};

void g(C* pc)
{
    pc->a = 1;  // error: ambiguous: A::a or B::a
    pc->b();    // error: ambiguous: A::b or B::b
    pc->f();    // error: ambiguous: A::f or B::f
    pc->f(1);   // error: ambiguous: A::f or B::f
    pc->g();    // error: ambiguous: A::g or B::g
    pc->g = 1;  // error: ambiguous: A::g or B::g
    pc->h();    // ok
    pc->h(1);   // ok
}
```

—*end example*]

4 If the name of an overloaded function is unambiguously found, overloading resolution (13.3) also takes place before access control. Ambiguities can often be resolved by qualifying a name with its class name. [*Example:*

```
class A {
public:
    int f();
};

class B {
public:
    int f();
};

class C : public A, public B {
    int f() { return A::f() + B::f(); }
};
```

—*end example*]

5 A static member, a nested type or an enumerator defined in a base class `T` can unambiguously be found even if an object has more than one base class subobject of type `T`. Two base class subobjects share the nonstatic member subobjects of their common virtual base classes. [*Example:*

```
class V { public: int v; };
class A {
public:
    int a;
    static int   s;
    enum { e };
};
class B : public A, public virtual V {};
class C : public A, public virtual V {};

class D : public B, public C { };

void f(D* pd)
{
    pd->v++;          // ok: only one ‘v’ (virtual)
    pd->s++;          // ok: only one ‘s’ (static)
    int i = pd->e;    // ok: only one ‘e’ (enumerator)
    pd->a++;          // error, ambiguous: two ‘a’s in ‘D’
}
```

—*end example*]

6 When virtual base classes are used, a hidden declaration can be reached along a path through the sub-object lattice that does not pass through the hiding declaration. This is not an ambiguity. The identical use with nonvirtual base classes is an ambiguity; in that case there is no unique instance of the name that hides all the others. [*Example:*

```
class V { public: int f();  int x; };
class W { public: int g();  int y; };
class B : public virtual V, public W
{
public:
    int f();  int x;
    int g();  int y;
};
class C : public virtual V, public W { };

class D : public B, public C { void glorp(); };
```

$$\begin{tikzcd}
W & & V & & W \\
 & B \arrow[ul] \arrow[ur] & & C \arrow[ul] \arrow[ur] & \\
 & & D \arrow[ul] \arrow[ur] & &
\end{tikzcd}$$

The names defined in `V` and the left hand instance of `W` are hidden by those in `B`, but the names defined in the right hand instance of `W` are not hidden at all.

```
void D::glorp()
{
    x++;        // ok: B::x hides V::x
    f();        // ok: B::f() hides V::f()
    y++;        // error: B::y and C’s W::y
    g();        // error: B::g() and C’s W::g()
}
```

—*end example*]

7 An explicit or implicit conversion from a pointer to or an lvalue of a derived class to a pointer or reference to one of its base classes shall unambiguously refer to a unique object representing the base class. [*Example:*

```
class V { };
class A { };
class B : public A, public virtual V { };
class C : public A, public virtual V { };
class D : public B, public C { };

void g()
{
    D d;
    B* pb = &d;
    A* pa = &d;  // error, ambiguous: C’s A or B’s A ?
    V* pv = &d;  // fine: only one V sub-object
}
```

—*end example*]

10.3 Virtual functions [class.virtual]

1 Virtual functions support dynamic binding and object-oriented programming. A class that declares or inherits a virtual function is called a *polymorphic class*.

2 If a virtual member function `vf` is declared in a class `Base` and in a class `Derived`, derived directly or indirectly from `Base`, a member function `vf` with the same name and same parameter list as `Base::vf` is declared, then `Derived::vf` is also virtual (whether or not it is so declared) and it *overrides*[68) `Base::vf`. For convenience we say that any virtual function overrides itself. Then in any well-formed class, for each virtual function declared in that class or any of its direct or indirect base classes there is a unique *final overrider* that overrides that function and every other overrider of that function. The rules for member lookup (10.2) are used to determine the final overrider for a virtual function in the scope of a derived class.

3 [*Note:* a virtual member function does not have to be visible to be overridden, for example,

```
struct B {
        virtual void f();
};
struct D : B {
        void f(int);
};
struct D2 : D {
        void f();
};
```

the function `f(int)` in class `D` hides the virtual function `f()` in its base class `B`; `D::f(int)` is not a virtual function. However, `f()` declared in class `D2` has the same name and the same parameter list as `B::f()`, and therefore is a virtual function that overrides the function `B::f()` even though `B::f()` is not visible in class `D2`.]

4 Even if destructors are not inherited, a destructor in a derived class overrides a base class destructor declared virtual; see 12.4 and 12.5.

68) A function with the same name but a different parameter list (13) as a virtual function is not necessarily virtual and does not override. The use of the `virtual` specifier in the declaration of an overriding function is legal but redundant (has empty semantics). Access control (11) is not considered in determining overriding.

5 A program is ill-formed if the return type of any overriding function differs from the return type of the overridden function unless the return type of the latter is pointer or reference (possibly cv-qualified) to a class `B`, and the return type of the former is pointer or reference (respectively) to a class `D` such that `B` is an unambiguous direct or indirect base class of `D`, accessible in the class of the overriding function, and the cv-qualification in the return type of the overriding function is less than or equal to the cv-qualification in the return type of the overridden function. In that case when the overriding function is called as the final overrider of the overridden function, its result is converted to the type returned by the (statically chosen) overridden function (5.2.2). [*Example:*

```
class B {};
class D : private B { friend class Derived; };
struct Base {
    virtual void vf1();
    virtual void vf2();
    virtual void vf3();
    virtual B*   vf4();
    void f();
};

struct No_good : public Base {
    D*  vf4();        // error: B (base class of D) inaccessible
};

struct Derived : public Base {
    void vf1();       // virtual and overrides Base::vf1()
    void vf2(int);    // not virtual, hides Base::vf2()
    char vf3();       // error: invalid difference in return type only
    D*  vf4();        // okay: returns pointer to derived class
    void f();
};

void g()
{
    Derived d;
    Base* bp = &d;     // standard conversion:
                       // Derived* to Base*
    bp->vf1();         // calls Derived::vf1()
    bp->vf2();         // calls Base::vf2()
    bp->f();           // calls Base::f() (not virtual)
    B*  p = bp->vf4(); // calls Derived::pf() and converts the
                       //  result to B*
    Derived*  dp = &d;
    D*  q = dp->vf4(); // calls Derived::pf() and does not
                       // convert the result to B*
    dp->vf2();         // ill-formed: argument mismatch
}
```

—end example]

6 [*Note:* the interpretation of the call of a virtual function depends on the type of the object for which it is called (the dynamic type), whereas the interpretation of a call of a nonvirtual member function depends only on the type of the pointer or reference denoting that object (the static type) (5.2.2).]

7 [*Note:* the `virtual` specifier implies membership, so a virtual function cannot be a nonmember (7.1.2) function. Nor can a virtual function be a static member, since a virtual function call relies on a specific object for determining which function to invoke. A virtual function declared in one class can be declared a `friend` in another class.]

8 A virtual function declared in a class shall be defined, or declared pure (10.4) in that class, or both; but no diagnostic is required (3.2).

9 [*Example:* here are some uses of virtual functions with multiple base classes:

```
struct A {
    virtual void f();
};

struct B1 : A {   // note non-virtual derivation
    void f();
};

struct B2 : A {
    void f();
};

struct D : B1, B2 {  // D has two separate A sub-objects
};

void foo()
{
    D   d;
    // A*  ap = &d; // would be ill-formed: ambiguous
    B1*  b1p = &d;
    A*   ap = b1p;
    D*   dp = &d;
    ap->f();  // calls D::B1::f
    dp->f();  // ill-formed: ambiguous
}
```

In class D above there are two occurrences of class A and hence two occurrences of the virtual member function A::f. The final overrider of B1::A::f is B1::f and the final overrider of B2::A::f is B2::f.

10 The following example shows a function that does not have a unique final overrider:

```
struct A {
    virtual void f();
};

struct VB1 : virtual A {   // note virtual derivation
    void f();
};

struct VB2 : virtual A {
    void f();
};

struct Error : VB1, VB2 {  // ill-formed
};

struct Okay : VB1, VB2 {
    void f();
};
```

Both VB1::f and VB2::f override A::f but there is no overrider of both of them in class Error. This example is therefore ill-formed. Class Okay is well formed, however, because Okay::f is a final overrider.

11 The following example uses the well-formed classes from above.

```
struct VB1a : virtual A {  // does not declare f
};
```

```
struct Da : VB1a, VB2 {
};

void foe()
{
    VB1a*  vb1ap = new Da;
    vb1ap->f();  // calls VB2:f
}
```

—*end example*]

12 Explicit qualification with the scope operator (5.1) suppresses the virtual call mechanism. [*Example:*

```
class B { public: virtual void f(); };
class D : public B { public: void f(); };

void D::f() { /* ... */ B::f(); }
```

Here, the function call in `D::f` really does call `B::f` and not `D::f`.]

10.4 Abstract classes [class.abstract]

1 The abstract class mechanism supports the notion of a general concept, such as a `shape`, of which only more concrete variants, such as `circle` and `square`, can actually be used. An abstract class can also be used to define an interface for which derived classes provide a variety of implementations.

2 An *abstract class* is a class that can be used only as a base class of some other class; no objects of an abstract class can be created except as sub-objects of a class derived from it. A class is abstract if it has at least one *pure virtual function*. [*Note:* such a function might be inherited: see below.] A virtual function is specified *pure* by using a *pure-specifier* (9.2) in the function declaration in the class declaration. A pure virtual function need be defined only if explicitly called with the *qualified-id* syntax (5.1). [*Example:*

```
class point { /* ... */ };
class shape {           // abstract class
    point center;
    // ...
public:
    point where() { return center; }
    void move(point p) { center=p; draw(); }
    virtual void rotate(int) = 0;  // pure virtual
    virtual void draw() = 0;       // pure virtual
    // ...
};
```

—*end example*] An abstract class shall not be used as a parameter type, as a function return type, or as the type of an explicit conversion. Pointers and references to an abstract class can be declared. [*Example:*

```
shape x;           // error: object of abstract class
shape* p;          // ok
shape f();         // error
void g(shape);     // error
shape& h(shape&);  // ok
```

—*end example*]

3 A class is abstract if it contains or inherits at least one pure virtual function for which the final overrider is pure virtual. [*Example:*

```
class ab_circle : public shape {
    int radius;
public:
    void rotate(int) {}
    // ab_circle::draw() is a pure virtual
};
```

Since `shape::draw()` is a pure virtual function `ab_circle::draw()` is a pure virtual by default. The alternative declaration,

```
class circle : public shape {
    int radius;
public:
    void rotate(int) {}
    void draw(); // a definition is required somewhere
};
```

would make class `circle` nonabstract and a definition of `circle::draw()` must be provided.]

4 [*Note:* an abstract class can be derived from a class that is not abstract, and a pure virtual function may override a virtual function which is not pure.]

5 Member functions can be called from a constructor (or destructor) of an abstract class; the effect of making a virtual call (10.3) to a pure virtual function directly or indirectly for the object being created (or destroyed) from such a constructor (or destrctor) is undefined.

11 Member access control [class.access]

1 A member of a class can be

— `private`; that is, its name can be used only by member functions, static data members, and friends of the class in which it is declared.

— `protected`; that is, its name can be used only by member functions, static data members, and friends of the class in which it is declared and by member functions, static data members, and friends of classes derived from this class (see 11.5).

— `public`; that is, its name can be used anywhere without access restriction.

2 Members of a class defined with the keyword `class` are `private` by default. Members of a class defined with the keywords `struct` or `union` are `public` by default. [*Example:*

```
class X {
    int a;  // X::a is private by default
};

struct S {
    int a;  // S::a is public by default
};
```

—*end example*]

3 Access control is applied uniformly to all names.

4 It should be noted that it is *access* to members and base classes that is controlled, not their *visibility*. Names of members are still visible, and implicit conversions to base classes are still considered, when those members and base classes are inaccessible. The interpretation of a given construct is established without regard to access control. If the interpretation established makes use of inaccessible member names or base classes, the construct is ill-formed.

5 All access controls in this clause affect the ability to access a class member from a particular scope. In particular, access controls apply as usual to members accessed as part of a function return type, even though it is not possible to determine the access privileges of that use without first parsing the rest of the function. [*Example:*

```
class A {
    typedef int I;      // private member
    I f();
    friend I g(I);
    static I x;
};

A::I A::f() { return 0; }
A::I g(A::I);
A::I g(A::I p) { return 0; }
A::I A::x = 0;
```

Here, all the uses of A::I are well-formed because `A::f` and `A::x` are members of class `A` and `g` is a friend of class `A`. This implies, for example, that access checking on the first use of `A::I` must be deferred until it is determined that this use of `A::I` is as the return type of a member of class `A`. —*end example*]

6 It is necessary to name a class member to define it outside of the definition of its class. For this reason, no access checking is performed on the components of the *qualified-id* used to name the member in the declarator of such a definition. [*Example:*

```
class D {
    class E {
        static int m;
    };
};
int D::E::m = 1;   // Okay, no access error on private 'E'
```

—*end example*]

11.1 Access specifiers [class.access.spec]

1 Member declarations can be labeled by an *access-specifier* (10):

access-specifier : *member-specification*$_{opt}$

An *access-specifier* specifies the access rules for members following it until the end of the class or until another *access-specifier* is encountered. [*Example:*

```
class X {
    int a;  // X::a is private by default: 'class' used
public:
    int b;  // X::b is public
    int c;  // X::c is public
};
```

—*end example*] Any number of access specifiers is allowed and no particular order is required. [*Example:*

```
struct S {
    int a;  // S::a is public by default: 'struct' used
protected:
    int b;  // S::b is protected
private:
    int c;  // S::c is private
public:
    int d;  // S::d is public
};
```

—*end example*]

2 The order of allocation of data members with separate *access-specifier* labels is implementation-defined (9.2).

11.2 Access specifiers for base classes [class.access.base]

1 If a class is declared to be a base class (10) for another class using the `public` access specifier, the `public` members of the base class are accessible as `public` members of the derived class and `protected` members of the base class are accessible as `protected` members of the derived class. If a class is declared to be a base class for another class using the `protected` access specifier, the `public` and `protected` members of the base class are accessible as `protected` members of the derived class. If a class is declared to be a base class for another class using the `private` access specifier, the `public` and `protected` members of the base class are accessible as `private` members of the derived class[69].

[69] As specified previously in 11, private members of a base class remain inaccessible even to derived classes unless `friend` declarations within the base class declaration are used to grant access explicitly.

2 In the absence of an *access-specifier* for a base class, `public` is assumed when the derived class is declared `struct` and `private` is assumed when the class is declared `class`. [*Example:*

```
class B { /* ... */ };
class D1 : private B { /* ... */ };
class D2 : public B { /* ... */ };
class D3 : B { /* ... */ };    // ‘B’ private by default
struct D4 : public B { /* ... */ };
struct D5 : private B { /* ... */ };
struct D6 : B { /* ... */ };   // ‘B’ public by default
class D7 : protected B { /* ... */ };
struct D8 : protected B { /* ... */ };
```

Here `B` is a public base of `D2`, `D4`, and `D6`, a private base of `D1`, `D3`, and `D5`, and a protected base of `D7` and `D8`. —*end example*]

3 [*Note:* Because of the rules on pointer conversion (4.10), a static member of a private base class might be inaccessible as an inherited name, but accessible directly. For example,

```
class B {
public:
        int mi;         // nonstatic member
        static int si;  // static member
};
class D : private B {
};
class DD : public D {
        void f();
};

void DD::f() {
        mi = 3;        // error: mi is private in D
        si = 3;        // error: si is private in D
        B  b;
        b.mi = 3;      // okay (b.mi is different from this->mi)
        b.si = 3;      // okay (b.si is different from this->si)
        B::si = 3;     // okay
        B* bp1 = this; // error: B is a private base class
        B* bp2 = (B*)this;  // okay with cast
        bp2->mi = 3;   // okay: access through a pointer to B.
}
```

—*end note*]

4 A base class is said to be accessible if an invented public member of the base class is accessible. If a base class is accessible, one can implicitly convert a pointer to a derived class to a pointer to that base class (4.10, 4.11). [*Note:* It follows that members and friends of a class `X` can implicitly convert an `X*` to a pointer to a private or protected immediate base class of `X`.]

11.3 Access declarations [class.access.dcl]

1 The access of a member of a base class can be changed in the derived class by mentioning its *qualified-id* in the derived class declaration. Such mention is called an *access declaration*. The base class member is given, in the derived class, the access in effect in the derived class declaration at the point of the access declaration. The effect of an access declaration *qualified-id* `;` is defined to be equivalent to the declaration `using` *qualified-id* `;`.[70)]

[70)] Access declarations are deprecated; member *using-declarations* (7.3.3) provide a better means of doing the same things. In earlier versions of the C++ language, access declarations were more limited; they were generalized and made equivalent to using-declarations in the interest of simplicity. Programmers are encouraged to use `using`, rather than the new capabilities of access declarations, in new code.

2 [*Example:*

```
class A {
public:
    int z;
    int z1;
};

class B : public A {
    int a;
public:
    int b, c;
    int bf();
protected:
    int x;
    int y;
};

class D : private B {
    int d;
public:
    B::c;  // adjust access to ‘B::c’
    B::z;  // adjust access to ‘A::z’
    A::z1; // adjust access to ‘A::z1’
    int e;
    int df();
protected:
    B::x;  // adjust access to ‘B::x’
    int g;
};

class X : public D {
    int xf();
};

int ef(D&);
int ff(X&);
```

The external function `ef` can use only the names `c`, `z`, `z1`, `e`, and `df`. Being a member of `D`, the function `df` can use the names `b`, `c`, `z`, `z1`, `bf`, `x`, `y`, `d`, `e`, `df`, and `g`, but not `a`. Being a member of `B`, the function `bf` can use the members `a`, `b`, `c`, `z`, `z1`, `bf`, `x`, and `y`. The function `xf` can use the public and protected names from `D`, that is, `c`, `z`, `z1`, `e`, and `df` (public), and `x`, and `g` (protected). Thus the external function `ff` has access only to `c`, `z`, `z1`, `e`, and `df`. If `D` were a protected or private base class of `X`, `xf` would have the same privileges as before, but `ff` would have no access at all.]

11.4 Friends [class.friend]

1 A friend of a class is a function that is not a member of the class but is permitted to use the private and protected member names from the class. The name of a friend is not in the scope of the class, and the friend is not called with the member access operators (5.2.4) unless it is a member of another class. [*Example:* the following example illustrates the differences between members and friends:

```
class X {
    int a;
    friend void friend_set(X*, int);
public:
    void member_set(int);
};

void friend_set(X* p, int i) { p->a = i; }
void X::member_set(int i) { a = i; }
```

```
        void f()
        {
            X obj;
            friend_set(&obj,10);
            obj.member_set(10);
        }
```

—*end example*]

2 When a `friend` declaration refers to an overloaded name or operator, only the function specified by the parameter types becomes a friend. A member function of a class `X` can be a friend of a class `Y`. [*Example:*

```
        class Y {
            friend char* X::foo(int);
            // ...
        };
```

—*end example*] Declaring a class to be a friend implies that private and protected names from the class granting friendship can be used in the class receiving it. [*Example:*

```
        class X {
            enum { a=100 };
            friend class Y;
        };

        class Y {
            int v[X::a];  // ok, Y is a friend of X
        };

        class Z {
            int v[X::a];  // error: X::a is private
        };
```

—*end example*] Access to private and protected names is also granted to member functions of the friend class (as if the functions were each friends) and to the static data member definitions of the friend class.

3 A function declared as a `friend` and not previously declared, is introduced in the smallest enclosing non-class, non-function prototype scope that contains the `friend` declaration. [*Note:* For a class mentioned as a `friend` and not previously declared, see 7.1.5.3.]

4 A function first declared in a friend declaration has external linkage (3.5). Otherwise, it retains its previous linkage (7.1.1). No *storage-class-specifier* shall appear in the *decl-specifier-seq* of a friend declaration.

5 A function of namespace scope can be defined in a `friend` declaration of a non-local class (9.9). The function is then `inline`. A `friend` function defined in a class is in the (lexical) scope of the class in which it is defined. A friend function defined outside the class is not (3.4).

6 Friend declarations are not affected by *access-specifiers* (9.2).

7 Friendship is neither inherited nor transitive. [*Example:*

```
        class A {
            friend class B;
            int a;
        };

        class B {
            friend class C;
        };
```

```
class C  {
    void f(A* p)
    {
        p->a++;  // error: C is not a friend of A
                 // despite being a friend of a friend
    }
};

class D : public B  {
    void f(A* p)
    {
        p->a++;  // error: D is not a friend of A
                 // despite being derived from a friend
    }
};
```

—*end example*]

11.5 Protected member access [class.protected]

1 A friend or a member function of a derived class can access a protected static member, type or enumerator constant of a base class; if the access is through a *qualified-id*, the *nested-name-specifier* must name the derived class (or any class derived from that class).

2 A friend or a member function of a derived class can access a protected nonstatic member of a base class. Except when forming a pointer to member (5.3.1), the access must be through a pointer to, reference to, or object of the derived class itself (or any class derived from that class) (5.2.4). If the nonstatic protected member thus accessed is also qualified, the qualification is ignored for the purpose of this access checking. If the access is to form a pointer to member, the *nested-name-specifier* shall name the derived class (or any class derived from that class). [*Example:*

```
class B {
protected:
    int i;
    static int j;
};

class D1 : public B {
};

class D2 : public B {
    friend void fr(B*,D1*,D2*);
    void mem(B*,D1*);
};

void fr(B* pb, D1* p1, D2* p2)
{
    pb->i = 1;  // illegal
    p1->i = 2;  // illegal
    p2->i = 3;  // ok (access through a D2)
    p2->B::i = 4;  // ok (access through a D2, qualification ignored)
    int B::* pmi_B = &B::i;    // illegal
    int B::* pmi_B = &D2::i;   // ok (type of &D2::i is "int B::*")
    B::j = 5;   // illegal
    D2::j =6;   // ok (access through a D2)
}
```

```
void D2::mem(B* pb, D1* p1)
{
    pb->i = 1;  // illegal
    p1->i = 2;  // illegal
    i = 3;      // ok (access through ‘this’)
    B::i = 4;   // ok (access through ‘this’, qualification ignored)
    j = 5;      // ok (static member accessed by derived class function)
    B::j = 6;   // illegal
}

void g(B* pb, D1* p1, D2* p2)
{
    pb->i = 1;  // illegal
    p1->i = 2;  // illegal
    p2->i = 3;  // illegal
}
```

—*end example*]

11.6 Access to virtual functions [class.access.virt]

1 The access rules (11) for a virtual function are determined by its declaration and are not affected by the rules for a function that later overrides it. [*Example:*

```
class B {
public:
    virtual int f();
};

class D : public B {
private:
    int f();
};

void f()
{
    D d;
    B* pb = &d;
    D* pd = &d;

    pb->f();  // ok: B::f() is public,
              // D::f() is invoked
    pd->f();  // error: D::f() is private
}
```

—*end example*] Access is checked at the call point using the type of the expression used to denote the object for which the member function is called (`B*` in the example above). The access of the member function in the class in which it was defined (`D` in the example above) is in general not known.

11.7 Multiple access [class.paths]

1 If a name can be reached by several paths through a multiple inheritance graph, the access is that of the path that gives most access. [*Example:*

```
class W { public: void f(); };
class A : private virtual W { };
class B : public virtual W { };
class C : public A, public B {
    void f() { W::f(); }  // ok
};
```

Since `W::f()` is available to `C::f()` along the public path through `B`, access is allowed. —*end* *example*]

12 Special member functions [special]

1 [*Note:* the special member functions affect the way objects of class type are created, copied, and destroyed, and how values can be converted to values of other types. Often such special member functions are called implicitly. The processor will implicitly declare these member functions for a class type when the programmer does not explicitly declare them.]

2 These member functions obey the usual access rules (11). [*Example:* declaring a constructor `protected` ensures that only derived classes and friends can create objects using it.]

12.1 Constructors [class.ctor]

1 Constructors do not have names. A special declarator syntax using the constructor's class name followed by a parameter list is used to declare the constructor in its class definition. [*Example:*

```
class C {
public:
        C(); // declares the constructor
};
```

—*end example*] A constructor is used to initialize objects of its class type. Because constructors do not have names, they are never found during name lookup; however an explicit type conversion using the functional notation (5.2.3) will cause a constructor to be called to initialize an object. [*Note:* for initialization of objects of class type see 12.6.]

2 A constructor can be invoked for a `const`, `volatile` or `const volatile` object.[71)] A constructor shall not be declared `const`, `volatile`, or `const volatile` (9.4.2). A constructor shall not be `virtual` (10.3) or `static` (9.5).

3 Constructors are not inherited (10).

4 A *default* constructor for a class `X` is a constructor of class `X` that can be called without an argument. If there is no *user-declared* constructor for class `X`, a default constructor is implicitly declared. An *implicitly-declared* default constructor is a `public` member of its class. A constructor is *trivial* if it is an implicitly-declared default constructor and if:

— its class has no virtual functions (10.3) and no virtual base classes (10.1), and

— all the direct base classes of its class have trivial constructors, and

— for all the nonstatic data members of its class that are of class type (or array thereof), each such class has a trivial constructor.

5 Otherwise, the constructor is *non-trivial*.

6 An implicitly-declared default constructor for a class is *implicitly defined* when it is used to create an object of its class type (3.7). A program is ill-formed if the class for which a default constructor is implicitly defined has:

— a nonstatic data member of `const` type, or

[71)] Volatile semantics might or might not be used.

— a nonstatic data member of reference type, or

— a nonstatic data member of class type (or array thereof) with an inaccessible default constructor, or

— a base class with an inaccessible default constructor.

Before the implicitly-declared default constructor for a class is implicitly defined, all the implicitly-declared default constructors for its base classes and its nonstatic data members shall have been implicitly defined.

7 [*Note:* subclause 12.6.2 describes the order in which constructors for base classes and non-static data members are called and describes how arguments can be specified for the calls to these constructors.]

8 A *copy constructor* for a class `X` is a constructor with a first parameter of type `X&` or of type `const X&`. [*Note:* see 12.8 for more information on copy constructors.]

9 A union member shall not be of a class type (or array thereof) that has a non-trivial constructor.

10 No return type (not even `void`) shall be specified for a constructor. A `return` statement in the body of a constructor shall not specify a return value. The address of a constructor shall not be taken.

11 A constructor can be used explicitly to create new objects of its type, using the syntax

class-name (*expression-list*$_{opt}$)

[*Example:*

```
complex zz = complex(1,2.3);
cprint( complex(7.8,1.2) );
```

—*end example*] An object created in this way is unnamed. [*Note:* subclause 12.2 describes the lifetime of temporary objects.]

12 [*Note:* some language constructs have special semantics when used during construction; see 12.6.2 and 12.7.]

12.2 Temporary objects [class.temporary]

1 In some circumstances it might be necessary or convenient for the processor to generate a temporary object. Precisely when such temporaries are introduced is implementation-defined. Even when the creation of the temporary object is avoided, all the semantic restrictions must be respected as if the temporary object was created. [*Example:* even if the copy constructor is not called, all the semantic restrictions, such as accessibility, shall be satisfied.]

2 [*Example:*

```
class X {
    // ...
public:
    // ...
    X(int);
    X(const X&);
    ~X();
};

X f(X);

void g()
{
    X a(1);
    X b = f(X(2));
    a = f(a);
}
```

Here, an implementation might use a temporary in which to construct `X(2)` before passing it to `f()` using

X's copy-constructor; alternatively, X(2) might be constructed in the space used to hold the argument. Also, a temporary might be used to hold the result of f(X(2)) before copying it to b using X's copy-constructor; alternatively, f()'s result might be constructed in b. On the other hand, the expression a=f(a) requires a temporary for either the argument a or the result of f(a) to avoid undesired aliasing of a.]

3 When a processor introduces a temporary object of a class that has a non-trivial constructor (12.1), it shall ensure that a constructor is called for the temporary object. Similarly, the destructor shall be called for a temporary with a non-trivial destructor (12.4). Temporary objects are destroyed as the last step in evaluating the full-expression (1.8) that (lexically) contains the point where they were created. This is true even if that evaluation ends in throwing an exception.

4 There are two contexts in which temporaries are destroyed at a different point than the end of the full-expression. The first context is when an expression appears as an initializer for a declarator defining an object. In that context, the temporary that holds the result of the expression shall persist until the object's initialization is complete. The object is initialized from a copy of the temporary; during this copying, an implementation can call the copy constructor many times; the temporary is destroyed as soon as it has been copied.

5 The second context is when a temporary is bound to a reference. The temporary bound to the reference or the temporary containing the sub-object that is bound to the reference persists for the lifetime of the reference initialized or until the end of the scope in which the temporary is created, which ever comes first. A temporary holding the result of an initializer expression for a declarator that declares a reference persists until the end of the scope in which the reference declaration occurs. A temporary bound to a reference in a constructor's ctor-initializer (12.6.2) persists until the constructor exits. A temporary bound to a reference parameter in a function call (5.2.2) persists until the completion of the complete expression containing the call. A temporary bound in a function return statement (6.6.3) persists until the function exits.

6 In all cases, temporaries are destroyed in reverse order of creation.

12.3 Conversions [class.conv]

1 Type conversions of class objects can be specified by constructors and by conversion functions.

2 Such conversions, often called *user-defined conversions*, are used implicitly in addition to standard conversions (4); see 13.3.1.3. [*Example:* a function expecting an argument of type X can be called not only with an argument of type X but also with an argument of type T where a conversion from T to X exists.] [*Note:* user-defined conversions are used similarly for conversion of initializers (8.5), function arguments (5.2.2, 8.3.5), function return values (6.6.3, 8.3.5), expression operands (5), expressions controlling iteration and selection statements (6.4, 6.5), and explicit type conversions (5.2.3, 5.4).]

3 User-defined conversions are applied only where they are unambiguous (10.2, 12.3.2). Conversions obey the access control rules (11). Access control is applied after ambiguity resolution (3.4).

4 [*Note:* See 13.3 for a discussion of the use of conversions in function calls as well as examples below.]

12.3.1 Conversion by constructor [class.conv.ctor]

1 A constructor declared without the *function-specifier* explicit that can be called with a single parameter specifies a conversion from the type of its first parameter to the type of its class. Such a constructor is called a converting constructor. [*Example:*

```
class X {
    // ...
public:
    X(int);
    X(const char*, int =0);
};
```

```
void f(X arg)
{
    X a = 1;         // a = X(1)
    X b = "Jessie";  // b = X("Jessie",0)
    a = 2;           // a = X(2)
    f(3);            // f(X(3))
}
```

—*end example*]

2 A nonconverting constructor constructs objects just like converting constructors, but does so only where a constructor call is explicitly indicated by the syntax. [*Example:*

```
class Z {
public:
        explicit Z(int);
        // ...
};

Z a1 = 1;        // error: no implicit conversion
Z a3 = Z(1);     // ok: explicit use of constructor
Z a2(1);         // ok: explicit use of constructor
Z* p = new Z(1); // ok: explicit use of constructor
```

—*end example*]

12.3.2 Conversion functions [class.conv.fct]

1 A member function of a class `X` with a name of the form

conversion-function-id:
    `operator` *conversion-type-id*

conversion-type-id:
    type-specifier-seq conversion-declarator$_{opt}$

conversion-declarator:
    ptr-operator conversion-declarator$_{opt}$

specifies a conversion from `X` to the type specified by the *conversion-type-id.* Such member functions are called conversion functions. Classes, enumerations, and *typedef-name*s shall not be declared in the *type-specifier-seq.* Neither parameter types nor return type can be specified. A conversion operator is never used to convert a (possibly qualified) object (or reference to an object) to the (possibly qualified) same object type (or a reference to it), or to a (possibly qualified) base class of that type (or a reference to it).[72)] If *conversion-type-id* is `void` or cv-qualified `void`, the program is ill-formed.

2 [*Example:*

```
class X {
    // ...
public:
    operator int();
};
```

72) Even though never directly called to perform a conversion, such conversion operators can be declared and can potentially be reached through a call to a virtual conversion operator in a base class

```
void f(X a)
{
    int i = int(a);
    i = (int)a;
    i = a;
}
```

In all three cases the value assigned will be converted by `X::operator int()`. —*end example*]

3 User-defined conversions are not restricted to use in assignments and initializations. [*Example:*

```
void g(X a, X b)
{
    int i = (a) ? 1+a : 0;
    int j = (a&&b) ? a+b : i;
    if (a) { // ...
    }
}
```

—*end example*]

4 The *conversion-type-id* in a *conversion-function-id* is the longest possible sequence of *conversion-declarator*s. [*Note:* this prevents ambiguities between the declarator operator * and its expression counterparts. [*Example:*

```
&ac.operator int*i; // syntax error:
                    // parsed as: ’&(ac.operator int *) i’
                    // not as: ’&(ac.operator int)*i’
```

The * is the pointer declarator and not the multiplication operator.]]

5 Conversion operators are inherited.

6 Conversion functions can be virtual.

7 At most one user-defined conversion (constructor or conversion function) is implicitly applied to a single value. [*Example:*

```
class X {
    // ...
public:
    operator int();
};

class Y {
    // ...
public:
    operator X();
};

Y a;
int b = a;    // illegal:
              // a.operator X().operator int() not tried
int c = X(a); // ok: a.operator X().operator int()
```

—*end example*]

8 User-defined conversions are used implicitly only if they are unambiguous. A conversion function in a derived class does not hide a conversion function in a base class unless the two functions convert to the same type. [*Example:*

```
class X {
public:
    // ...
    operator int();
};

class Y : public X {
public:
    // ...
    operator void*();
};

void f(Y& a)
{
    if (a) {    // error: ambiguous
        // ...
    }
}
```

—*end example*]

12.4 Destructors [class.dtor]

1 A member function of class `cl` named `~cl` is called a destructor; it is used to destroy objects of type `cl`. A destructor takes no parameters, and no return type can be specified for it (not even `void`). It is not possible to take the address of a destructor. A destructor can be invoked for a `const`, `volatile` or `const volatile` object.[73] A destructor shall not be declared `const`, `volatile` or `const volatile` (9.4.2). A destructor shall not be `static`.

2 If a class has no *user-declared* destructor, a destructor is declared implicitly. An *implicitly-declared* destructor is a `public` member of its class. A destructor is *trivial* if it is an implicitly-declared destructor and if:

— all of the direct base classes of its class have trivial destructors and

— for all of the non-static data members of its class that are of class type (or array thereof), each such class has a trivial destructor.

3 Otherwise, the destructor is *non-trivial*.

4 An implicitly-declared destructor is *implicitly defined* when it is used to destroy an object of its class type (3.7). A program is ill-formed if the class for which a destructor is implicitly defined has:

— a non-static data member of class type (or array thereof) with an inaccessible destructor, or

— a base class with an inaccessible destructor.

Before the implicitly-declared destructor for a class is implicitly defined, all the implicitly-declared destructors for its base classes and its nonstatic data members shall have been implicitly defined.

5 Bases and members are destroyed in reverse order of their construction (see 12.6.2). Destructors for elements of an array are called in reverse order of their construction (see 12.6).

6 Destructors are not inherited. A destructor can be declared `virtual` (10.3) or pure `virtual` (10.4); if any objects of that class or any derived class are created in the program, the destructor shall be defined. If a class has a base class with a virtual destructor, its destructor (whether user- or implicitly- declared) is virtual.

[73] Volatile semantics might or might not be used.

7 [*Note:* some language constructs have special semantics when used during destruction; see 12.7.]

8 A union member shall not be of a class type (or array thereof) that has a non-trivial destructor.

9 Destructors are invoked implicitly (1) when an automatic variable (3.7) or temporary (12.2, 8.5.3) object goes out of scope, (2) for constructed static (3.7) objects at program termination (3.6), and (3) through use of a *delete-expression* (5.3.5) for objects allocated by a *new-expression* (5.3.4). Destructors can also be invoked explicitly. A *delete-expression* invokes the destructor for the referenced object and passes the address of its memory to a deallocation function (5.3.5, 12.5). [*Example:*

```
class X {
    // ...
public:
    X(int);
    ~X();
};

void g(X*);

void f()        // common use:
{
    X* p = new X(111);  // allocate and initialize
    g(p);
    delete p;           // cleanup and deallocate
}
```

—*end example*]

10 [*Note:* explicit calls of destructors are rarely needed. One use of such calls is for objects placed at specific addresses using a *new-expression* with the placement option. Such use of explicit placement and destruction of objects can be necessary to cope with dedicated hardware resources and for writing memory management facilities. For example,

```
void* operator new(size_t, void* p) { return p; }

void f(X* p);

static char buf[sizeof(X)];

void g()        // rare, specialized use:
{
    X* p = new(buf) X(222);  // use buf[]
                             // and initialize
    f(p);
    p->X::~X();              // cleanup
}
```

—*end note*]

11 Invocation of destructors is subject to the usual rules for member functions (9.4), e.g., an object of the appropriate type is required (except invoking `delete` on a null pointer has no effect). Once a destructor is invoked for an object, the object no longer exists; the behavior is undefined if the destructor is invoked for an object whose lifetime has ended (3.8). [*Example:* if the destructor for an automatic object is explicitly invoked, and the block is subsequently left in a manner that would ordinarily invoke implicit destruction of the object, the behavior is undefined. —*end example*]

12 The notation for explicit call of a destructor can be used for any scalar type name. Using the notation for a type that does not have a destructor has no effect. [*Note:* allowing this makes it possible to write code without having to know if a destructor exists for a given type. [*Example:*

```
        int* p;
        // ...
        p->int::~int();
```

—end example] *—end note*]

13

12.5 Free store [class.free]

1 When an object is created with a *new-expression* (5.3.4), an *allocation function* (`operator new()` for non-array objects or `operator new[]()` for arrays) is (implicitly) called to get the required storage (3.7.3.1).

2 When an object of class type `T` or an array of class `T` is created by a *new-expression*, the allocation function is looked up in the scope of class `T` using the usual rules.

3 When a *new-expression* is executed, the selected allocation function will be called with the amount of space requested (possibly zero) as its first argument.

4 Any allocation function for a class `X` is a static member (even if not explicitly declared `static`).

5 [*Example:*

```
        class Arena;  class Array_arena;
        struct B {
            void* operator new(size_t, Arena*);
        };
        struct D1 : B {
        };

        Arena*  ap;  Array_arena* aap;
        void foo(int i)
        {
            new (ap) D1;  // calls B::operator new(size_t, Arena*)
            new D1[i];    // calls ::operator new[](size_t)
            new D1;       // ill-formed: ::operator new(size_t) hidden
        }
```

—end example]

6 When an object is deleted with a *delete-expression* (5.3.5), a *deallocation function* (`operator delete()` for non-array objects or `operator delete[]()` for arrays) is (implicitly) called to reclaim the storage occupied by the object (3.7.3.2).

7 When an object is deleted by a *delete-expression*, the deallocation function is looked up in the scope of the class of the executed destructor (see 5.3.5) using the usual rules.

8 When a *delete-expression* is executed, the selected deallocation function will be called with the address of the block of storage to be reclaimed as its first argument and (if the two-parameter style is used) the size of the block as its second argument.[74)]

9 Any deallocation function for a class `X` is a static member (even if not explicitly declared `static`). [*Example:*

```
        class X {
            // ...
            void operator delete(void*);
            void operator delete[](void*, size_t);
        };
```

[74)] If the static class in the *delete-expression* is different from the dynamic class and the destructor is not virtual the size might be incorrect, but that case is already undefined; see 5.3.5.

```
class Y {
    // ...
    void operator delete(void*, size_t);
    void operator delete[](void*);
};
```

—*end example*]

10 Since member allocation and deallocation functions are `static` they cannot be virtual. However, the deallocation function actually called is determined by the destructor actually called, so if the destructor is virtual the effect is the same. [*Example:*

```
struct B {
    virtual ~B();
    void operator delete(void*, size_t);
};

struct D : B {
    void operator delete(void*);
    void operator delete[](void*, size_t);
};

void f(int i)
{
    B* bp = new D;
    delete bp;     // uses D::operator delete(void*)
    D* dp = new D[i];
    delete [] dp;  // uses D::operator delete[](void*, size_t)
}
```

Here, storage for the non-array object of class `D` is deallocated by `D::operator delete()`, due to the virtual destructor.]

11 For a virtual destructor (whether user- or implicitly- declared), the deallocation function to be called is determined by looking up the name of `operator delete` in the context of the outermost block of that destructor's definition (ignoring any names defined in that block). If the result of the lookup is ambiguous or inaccessible, the program is ill-formed.[75)]

12 Access to the deallocation function is checked statically. Hence, even though a different one might actually be executed, the statically visible deallocation function is required to be accessible. [*Example:* if `B::operator delete()` had been `private`, the delete expression would have been ill-formed.]

12.6 Initialization [class.init]

1 If `T` is either a class type or an array of class type, an object of type `T` is default-initialized (8.5) if:

— the object has static storage duration and no *initializer* is specified in its declaration (see 8.5), or

— the object is created with a *new-expression* of the form `new T()` (see 5.3.4), or

— the object is a temporary object created using the functional notation for type conversions `T()` (see 5.2.3), or

— the object is a subobject, either a base of type `T` or a member `m` of type `T`, of a class object being created by a constructor that specifies a *mem-initializer* of the form `T()` or `m()`, respectively (see 12.6.2).

75) This applies to destructor definitions, not mere declarations. A similar restriction is not needed for the array version of the `delete operator` because 5.3.5 requires that in all other situations, the static type of the *delete-expression*'s operand be the same as its dynamic type.

2 Furthermore, if an object of class type `T` (or array thereof)

— has automatic storage duration and no *initializer* is specified in its declaration, or

— is created with a *new-expression* with an omitted *new-initializer* (see 5.3.4), or

— is a subobject, either a base of type `T` or a member `m` of type `T` (or array thereof), of a class object created by a constructor that does not specify a *mem-initializer* for `T` or `m`, respectively (see 12.6.2),

then that object (or, for an array, each element of the array) shall be initialized by the default constructor for `T` (and the initialization is ill-formed if `T` has no accessible default constructor).

3 An object of class type (or array thereof) can be explicitly initialized; see 12.6.1 and 12.6.2.

4 When an array of class objects is initialized (either explicitly or implicitly), the constructor shall be called for each element of the array, following the subscript order; see 8.3.4. [*Note:* destructors for the array elements are called in reverse order of their construction.]

12.6.1 Explicit initialization [class.expl.init]

1 An object of class type can be initialized with a parenthesized *expression-list*, where the *expression-list* is construed as an argument list for a constructor that is called to initialize the object. Alternatively, a single *assignment-expression* can be specified as an *initializer* using the = form of initialization. Either direct-initialization semantics or copy-initialization semantics apply; see 8.5. [*Example:*

```
class complex {
    // ...
public:
    complex();
    complex(double);
    complex(double,double);
    // ...
};

complex sqrt(complex,complex);

complex a(1);             // initialize by a call of
                          // complex(double)
complex b = a;            // initialize by a copy of ‘a’
complex c = complex(1,2); // construct complex(1,2)
                          // using complex(double,double)
                          // copy it into ‘c’
complex d = sqrt(b,c);    // call sqrt(complex,complex)
                          // and copy the result into ‘d’
complex e;                // initialize by a call of
                          // complex()
complex f = 3;            // construct complex(3) using
                          // complex(double)
                          // copy it into ‘f’
complex g = { 1, 2 };     // error; constructor is required
```

—*end example*] [*Note:* overloading of the assignment operator (13.5.3) = has no effect on initialization.]

2 When an aggregate (whether class or array) contains members of class type and is initialized by a brace-enclosed *initializer-list* (8.5.1), each such member is copy-initialized (see 8.5) by the corresponding *assignment-expression*. If there are fewer *initializer*s in the *initializer-list* than members of the aggregate, each member not explicitly initialized shall be copy-initialized (8.5) with an *initializer* of the form `T()` (5.2.3), where `T` represents the type of the uninitialized member. [*Note:* subclause 8.5.1 describes how *assignment-expression*s in an *initializer-list* are paired with the aggregate members they initialize.] [*Example:*

```
complex v[6] = { 1,complex(1,2),complex(),2 };
```

Here, `complex::complex(double)` is called for the initialization of `v[0]` and `v[3]`, `complex::complex(double,double)` is called for the initialization of `v[1]`, `complex::complex()` is called for the initialization `v[2]`, `v[4]`, and `v[5]`. For another example,

```
        class X {
                int i;
                float f;
                complex c;
        } x = { 99, 88.8, 77.7 };
```

Here, `x.i` is initialized with 99, `x.f` is initialized with 88.8, and `complex::complex(double)` is called for the initialization of `x.c`.] [*Note:* braces can be elided in the *initializer-list* for any aggregate, even if the aggregate has members of a class type with user-defined type conversions; see 8.5.1.]

3 [*Note:* if `T` is a class type with no default constructor, any declaration of an object of type `T` (or array thereof) is ill-formed if no *initializer* is explicitly specified (see 12.6 and 8.5).]

4 [*Note:* the order in which objects with static storage duration are initialized is described in 3.6.2 and 6.7.]

12.6.2 Initializing bases and members [class.base.init]

1 In the definition of a constructor for a class, initializers for direct and virtual base subobjects and nonstatic data members can be specified by a *ctor-initializer*, which has the form

ctor-initializer:
  `:` *mem-initializer-list*

mem-initializer-list:
  mem-initializer
  mem-initializer `,` *mem-initializer-list*

mem-initializer:
  mem-initializer-id `(` *expression-list*$_{opt}$ `)`

mem-initializer-id:
  `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name*
  identifier

2 Unless the *mem-initializer-id* names a nonstatic data member of the constructor's class or a direct or virtual base of that class, the *mem-initializer* is ill-formed. A *mem-initializer-list* can initialize a base class using any name that denotes that base class type. [*Example:*

```
        struct A { A(); };
        typedef A global_A;
        struct B { };
        struct C: public A, public B { C(); };
        C::C(): global_A() { }    // mem-initializer for base A
```

—*end example*] If a *mem-initializer-id* is ambiguous because it designates both a direct non-virtual base class and an inherited virtual base class, the *mem-initializer* is ill-formed. [*Example:*

```
        struct A { A(); };
        struct B: public virtual A { };
        struct C: public A, public B { C(); };
        C::C(): A() { }           // ill-formed: which A?
```

—*end example*] If a *ctor-initializer* specifies more than one *mem-initializer* for the same member or base, the *ctor-initializer* is ill-formed.

3 The *expression-list* in a *mem-initializer* is used to initialize the base class or nonstatic data member subobject denoted by the *mem-initializer-id*. The semantics of a *mem-initializer* are as follows:

— if the *expression-list* of the *mem-initializer* is omitted, the base class or member subobject is default-initialized (see 8.5);

— otherwise, the subobject indicated by *mem-initializer-id* is direct-initialized using *expression-list* as the *initializer* (see 8.5).

[*Note:* if class X has a member m of class type M and M has no default constructor, then a definition of a constructor for class X is ill-formed if it does not specify a *mem-initializer* for m.] [*Note:* when a constructor creates an object of class type X, if X has a nonstatic data member m that is of const or reference type and if the member is neither specified in a *mem-initializer* nor eligible for default-initialization (8.5), then m will have an indeterminate value. [*Example:*

```
struct B1 { B1(int); /* ... */ };
struct B2 { B2(int); /* ... */ };

struct D : B1, B2 {
    D(int);
    B1 b;
    const c;
};

D::D(int a) : B2(a+1), B1(a+2), c(a+3), b(a+4)
{ /* ... */ }

D d(10);
```

—*end example*]]

4 Initialization shall proceed in the following order:

— First, and only for the constructor of the most derived class as described below, virtual base classes shall be initialized in the order they appear on a depth-first left-to-right traversal of the directed acyclic graph of base classes, where “left-to-right” is the order of appearance of the base class names in the derived class *base-specifier-list.*

— Then, direct base classes shall be initialized in declaration order as they appear in the *base-specifier-list* (regardless of the order of the *mem-initializers*).

— Then, nonstatic data members shall be initialized in the order they were declared in the class definition (again regardless of the order of the *mem-initializers*).

— Finally, the body of the constructor is executed.

[*Note:* the declaration order is mandated to ensure that base and member subobjects are destroyed in the reverse order of initialization.]

5 If a complete object (1.6), a nonstatic data member, or an array element is of class type, its type, for purposes of construction, is considered the *most derived* class, to distinguish it from the class type of any base class subobject of the most derived class. All sub-objects representing virtual base classes are initialized by the constructor of the most derived class. If the constructor of the most derived class does not specify a *mem-initializer* for a virtual base class V, then V’s default constructor is called to initialize the virtual base class subobject. If V does not have an accessible default constructor, the initialization is ill-formed. A *mem-initializer* naming a virtual base class shall be ignored during execution of the constructor of any class that is not the most derived class. [*Example:*

```
class V {
public:
    V();
    V(int);
    // ...
};
```

```
class A : public virtual V {
public:
    A();
    A(int);
    // ...
};

class B : public virtual V {
public:
    B();
    B(int);
    // ...
};

class C : public A, public B, private virtual V {
public:
    C();
    C(int);
    // ...
};

A::A(int i) : V(i) { /* ... */ }
B::B(int i) { /* ... */ }
C::C(int i) { /* ... */ }

V v(1); // use V(int)
A a(2); // use V(int)
B b(3); // use V()
C c(4); // use V()
```

—*end example*]

6 Names in the *expression-list* of a *mem-initializer* are evaluated in the scope of the constructor for which the *mem-initializer* is specified. [*Example:*

```
class X {
    int a;
    int b;
public:
    const int& r;
    X(int i): r(a), b(i) {}
};
```

initializes `X::r` to refer to `X::a` and initializes `X::b` with the value of the constructor parameter `i`; this takes place each time an object of class `X` is created.] [*Note:* this implies that the `this` pointer can be used in the *expression-list* of a *mem-initializer* to refer to the object being initialized.]

7 Member functions (including virtual member functions, 10.3) can be called for an object under construction. Similarly, an object under construction can be the operand of the `typeid` operator (5.2.7) or of a `dynamic_cast` (5.2.6). However, if these operations are performed in a *ctor-initializer* (or in a function called directly or indirectly from a *ctor-initializer*) before all the *mem-initializer*s for base classes have completed, the result of the operation is undefined. [*Example:*

```
class A {
public:
        A(int);
};
```

```
        class B : public A {
                int j;
        public:
                int f();
        B() : A(f()),           // undefined: calls member function
                                // but base A not yet initialized
                j(f()) { }      // well-defined: bases are all initialized
        };

        class C {
        public:
                C(int);
        };

        class D : public B, C {
                int i;
        public:
                D() : C(f()),   // undefined: calls member function
                                // but base C not yet initialized
                i(f()) {}       // well-defined: bases are all initialized
        };
```

—*end example*]

8 [*Note:* Clause 12.7 describes the result of virtual function calls, `typeid` and `dynamic_cast`s during construction for the well-defined cases; that is, describes the *polymorphic behavior* of an object under construction.]

12.7 Construction and destruction [class.cdtor]

1 For an object of non-POD class type (9), before the constructor begins execution and after the destructor finishes execution, referring to any nonstatic member or base class of the object results in undefined behavior. [*Example:*

```
        struct X { int i; };
        struct Y : X { };
        struct A { int a; };
        struct B : public A { int j; Y y; };

        extern B bobj;
        B* pb = &bobj;          // ok
        int* p1 = &bobj.a;      // undefined, refers to base class member
        int* p2 = &bobj.y.i;    // undefined, refers to member’s member

        A* pa = &bobj;          // undefined, upcast to a base class type
        B bobj;                 // definition of bobj

        extern X xobj;
        int* p3 = &xobj.i;      // Ok, X is a POD class
        X xobj;
```

For another example,

```
        struct W { int j; };
        struct X : public virtual W { };
        struct Y {
                int *p;
                X x;
                Y() : p(&x.j)    // undefined, x is not yet constructed
                { }
        };
```

—*end example*]

2 To explicitly or implicitly convert a pointer to an object of class X to a pointer to a direct or indirect base class B, the construction of X and the construction of all of its direct or indirect bases that directly or indirectly derive from B shall have started and the destruction of these classes shall not have completed, otherwise the computation results in undefined behavior. To form a pointer to a direct nonstatic member of an object X given a pointer to X, the construction of X shall have started and the destruction of X shall not have completed, otherwise the computation results in undefined behavior. [*Example:*

```
struct A { };
struct B : virtual A { };
struct C : B { };
struct D : virtual A { D(A*); };
struct X { X(A*); };

struct E : C, D, X {
        E() : D(this),  // undefined: upcast from E* to A*
                        // might use path E* -> D* -> A*
                        // but D is not constructed
        // D((C*)this), // defined:
                        // E* -> C* defined because E() has started
                        // and C* -> A* defined because
                        // C fully constructed
        X(this)         // defined: upon construction of X,
                        // C/B/D/A sublattice is fully constructed
        { }
};
```

—*end example*]

3 Member functions, including virtual functions (10.3), can be called during construction or destruction (12.6.2). When a virtual function is called directly or indirectly from a constructor (including from its *ctor-initializer*) or from a destructor, the function called is the one defined in the constructor or destructor's own class or in one of its bases, but not a function overriding it in a class derived from the constructor or destructor's class or overriding it in one of the other base classes of the complete object (1.6). If the virtual function call uses an explicit class member access (5.2.4) and the object-expression's type is neither the constructor or destructor's own class or one of its bases, the result of the call is undefined. [*Example:*

```
class V {
public:
        virtual void f();
        virtual void g();
};

class A : public virtual V {
public:
        virtual void f();
};

class B : public virtual V {
public:
        virtual void g();
        B(V*, A*);
};

class D : public A, B {
public:
        virtual void f();
        virtual void g();
        D() : B((A*)this, this) { }
};
```

```
        B::B(V* v, A* a) {
                f();    // calls V::f, not A::f
                g();    // calls B::g, not D::g
                v->g(); // v is base of B, the call is well-defined, calls B::g
                a->f(); // undefined behavior, a's type not a base of B
        }
```

—*end example*]

4 The `typeid` operator (5.2.7) can be used during construction or destruction (12.6.2). When `typeid` is used in a constructor (including in its *ctor-initializer*) or in a destructor, or used in a function called (directly or indirectly) from a constructor or destructor, if the operand of `typeid` refers to the object under construction or destruction, `typeid` yields the `type_info` representing the constructor or destructor's class. If the operand of `typeid` refers to the object under construction or destruction and the static type of the operand is neither the constructor or destructor's class nor one of its bases, the result of `typeid` is undefined.

5 `Dynamic_cast`s (5.2.6) can be used during construction or destruction (12.6.2). When a `dynamic_cast` is used in a constructor (including in its *ctor-initializer*) or in a destructor, or used in a function called (directly or indirectly) from a constructor or destructor, if the operand of the `dynamic_cast` refers to the object under construction or destruction, this object is considered to be a complete object that has the type of the constructor or destructor's class. If the operand of the `dynamic_cast` refers to the object under construction or destruction and the static type of the operand is not a pointer to or object of the constructor or destructor's own class or one of its bases, the `dynamic_cast` results in undefined behavior.

6 [*Example:*

```
        class V {
        public:
                virtual void f();
        };

        class A : public virtual V { };

        class B : public virtual V {
        public:
                B(V*, A*);
        };

        class D : public A, B {
        public:
                D() : B((A*)this, this) { }
        };

        B::B(V* v, A* a) {
                typeid(this);   // type_info for B
                typeid(*v);     // well-defined: *v has type V, a base of B
                                // yields type_info for B
                typeid(*a);     // undefined behavior: type A not a base of B
                dynamic_cast<B*>(v); // well-defined: v of type V*, V base of B
                                // results in B*
                dynamic_cast<B*>(a); // undefined behavior,
                                // a has type A*, A not a base of B
        }
```

—*end example*]

12.8 Copying class objects [class.copy]

1 A class object can be copied in two ways, by initialization (12.1, 8.5), including for function argument passing (5.2.2) and for function value return (6.6.3), and by assignment (5.17). Conceptually, these two operations are implemented by a copy constructor (12.1) and copy assignment operator (13.5.3).

2 A constructor for class `X` is a *copy* constructor if its first parameter is of type `X&` or `const X&` and either there are no other parameters or else all other parameters have default arguments (8.3.6). [*Example:* `X::X(const X&)` and `X::X(X&, int=1)` are copy constructors.

```
class X {
    // ...
public:
    X(int);
    X(const X&, int = 1);
};
X a(1);         // calls X(int);
X b(a, 0);      // calls X(const X&, int);
X c = b;        // calls X(const X&, int);
```

—*end example*] [*Note:* both forms of copy constructor may be declared for a class. [*Example:*

```
class X {
        // ...
public:
        X(const X&);
        X(X&); // OK
};
```

—*end example*] —*end note*] [*Note:* if a class `X` only has a copy constructor with a parameter of type `X&`, an initializer of type `const X` cannot initialize an object of type (possibily cv-qualified) `X`. [*Example:*

```
struct X {
        X();    // default constructor
        X(X&);  // copy constructor with a nonconst parameter
};
const X cx;
X x = cx;       // error -- X::X(X&) cannot copy cx into x
```

—*end example*] —*end note*]

3 A declaration of a constructor for a class `X` is ill-formed if its first parameter is of type (optionally cv-qualified) `X` and either there are no other parameters or else all other parameters have default arguments.

4 If the class definition does not explicitly declare a copy constructor, one is declared *implicitly*. Thus, for the class definition

```
struct X {
        X(const X&, int);
};
```

a copy constructor is implicitly-declared. If the user-declared constructor is later defined as

```
X::X(const X& x, int i =0) { ... }
```

then any use of `X`'s copy constructor is ill-formed because of the ambiguity; no diagnostic is required.

5 The implicitly-declared copy constructor for a class `X` will have the form

```
X::X(const X&)
```

if

— each direct or virtual base class `B` of `X` has a copy constructor whose first parameter is of type `const B&` and

— for all the nonstatic data members of `X` that are of a class type `M` (or array thereof), each such class type has a copy constructor whose first parameter is of type `const M&`.[76)]

Otherwise, the implicitly declared copy constructor will have the form

```
X::X(X&)
```

An implicitly-declared copy constructor is a `public` member of its class. Copy constructors are not inherited.

6 A copy constructor for class `X` is *trivial* if it is implicitly declared and if

— class `X` has no virtual functions (10.3) and no virtual base classes (10.1), and

— each direct base class of `X` has a trivial copy constructor, and

— for all the nonstatic data members of `X` that are of class type (or array thereof), each such class type has a trivial copy constructor;

otherwise the copy constructor is *non-trivial*.

7 An implicitly-declared copy constructor is *implicitly defined* if it is used to copy an object of its class type, even if the implementation elided its use (12.2). A program is ill-formed if the class for which a copy constructor is implicitly defined has:

— a nonstatic data member of class type (or array thereof) with an inaccessible or ambiguous copy constructor, or

— a base class with an inaccessible or ambiguous copy constructor.

Before the implicitly-declared copy constructor for a class is implicitly defined, all implicitly-declared copy constructors for its direct and virtual base classes and its nonstatic data members shall have been implicitly defined.

8 The implicitly-defined copy constructor for class `X` performs a memberwise copy of its subobjects. The order of copying is the same as the order of initialization of bases and members in a user-defined constructor (see 12.6.2). Each subobject is copied in the manner appropriate to its type:

— if the subobject is of class type, the copy constructor for the class is used;

— if the subobject is an array, each element is copied, in the manner appropriate to the element type;

— if the subobject is of scalar or pointer-to-member type, the built-in assignment operator is used.

Virtual base class subobjects shall be copied only once by the implicitly-defined copy constructor (see 12.6.2).

9 A user-declared *copy* assignment operator `X::operator=` is a non-static member function of class `X` with exactly one parameter of type `X`, `X&` or `const X&`. [*Note:* more than one form of copy assignment operator may be declared for a class.] [*Note:* if a class `X` only has a copy assignment operator with a parameter of type `X&`, an expression of type const `X` cannot be assigned to an object of type `X` [*Example:*

```
struct X {
        X()
        X& operator=(X&);
};
const X cx;
X x;
x = cx; // error:
        // X::operator=(X&) cannot assign cx into x
```

—end example] *—end note*]

76) This implies that the reference parameter of the implicitly-declared copy constructor cannot bind to a `volatile` lvalue; see C.2.8.

10 If the class definition does not explicitly declare a copy assignment operator, one is declared *implicitly*. The implicitly-declared copy assignment operator for a class X will have the form

```
X& X::operator=(const X&)
```

if

— each direct base class B of X has a copy assignment operator whose parameter is of type const B& and

— for all the nonstatic data members of X that are of a class type M (or array thereof), each such class type has a copy assignment operator whose parameter is of type const M&.[77)]

Otherwise, the implicitly declared copy constructor will have the form

```
X& X::operator=(X&)
```

The implicitly-declared copy assignment operator for class X has the return type X&; it returns the object for which the assignment operator is invoked, that is, the object assigned to. An implicitly-declared copy assignment operator is a public member of its class. Because a copy assignment operator is implicitly declared for a class if not declared by the user, a base class copy assignment operator is always hidden by the copy assignment operator of a derived class (13.5.3).

11 A copy assignment operator for class X is *trivial* if it is implicitly declared and if

— each direct base class of X has a trivial copy assignment operator, and

— for all the nonstatic data members of X that are of class type (or array thereof), each such class type has a trivial copy assignment operator;

otherwise the copy assignment operator is *non-trivial*.

12 An implicitly-declared copy assignment operator is *implicitly defined* when an object of its class type is assigned. A program is ill-formed if the class for which a copy assignment operator is implicitly defined has:

— a nonstatic data member of const type, or

— a nonstatic data member of reference type, or

— a nonstatic data member of class type (or array thereof) with an inaccessible copy assignment operator, or

— a base class with an inaccessible copy assignment operator.

Before the implicitly-declared copy assignment operator for a class is implicitly defined, all implicitly-declared copy assignment operators for its direct base classes and its nonstatic data members shall have been implicitly defined.

13 The implicitly-defined copy assignment operator for class X performs memberwise assignment of its subobjects. The direct base classes of X are assigned first, in the order of their declaration in the *base-specifier-list*, and then the immediate nonstatic data members of X are assigned, in the order in which they were declared in the class definition. Each subobject is assigned in the manner appropriate to its type:

— if the subobject is of class type, the copy assignment operator for the class is used;

— if the subobject is an array, each element is assigned, in the manner appropriate to the element type;

— if the subobject is of scalar or pointer-to-member type, the built-in assignment operator is used.

It is unspecified whether subobjects representing virtual base classes are assigned more than once by the

[77)] This implies that the reference parameter of the implicitly-declared copy assignment operator cannot bind to a volatile lvalue; see C.2.8.

implicitly-defined copy assignment operator. [*Example:*

```
struct V {
struct A : virtual V { };
struct B : virtual V { };
struct C : B, A { };
```

it is unspecified whether the virtual base class subobject `V` is assigned twice by the implicitly-defined copy assignment operator for `C`. —*end example*]

14 [*Note:* Copying one object into another using the copy constructor or the copy assignment operator does not change the layout or size of either object.]

15 Whenever a class object is copied and the implementation can prove that either the original or the copy will never again be used, an implementation is permitted to treat the original and the copy as two different ways of referring to the same object and not perform a copy at all. In that case, the object is destroyed at the later of times when the original and the copy would have been destroyed without the optimization.[78] [*Example:*

```
class Thing {
public:
        Thing();
        ~Thing();
        Thing(const Thing&);
        Thing operator=(const Thing&);
        void fun();
};

void f(Thing t) { }
void g(Thing t) { t.fun(); }

int main()
{
        Thing t1, t2, t3;
        f(t1);
        g(t2);
        g(t3);
t3.fun();
}
```

Here `t1` does not need to be copied when calling `f` because `f` does not use its formal parameter again after copying it. Although `g` uses its parameter, the call to `g(t2)` does not need to copy `t2` because `t2` is not used again after it is passed to `g`. On the other hand, `t3` is used after passing it to `g` so calling `g(t3)` is required to copy `t3`.]

[78] Because only one object is destroyed instead of two, and one copy constructor is not executed, there is still one object destroyed for each one constructed.

13 Overloading [over]

1 When two or more different declarations are specified for a single name in the same scope, that name is said to be *overloaded*. By extension, two declarations in the same scope that declare the same name but with different types are called *overloaded declarations*. Only function declarations can be overloaded; object and type declarations cannot be overloaded.

2 When an overloaded function name is used in a call, which overloaded function declaration is being referenced is determined by comparing the types of the arguments at the point of use with the types of the parameters in the overloaded declarations that are visible at the point of use. This function selection process is called *overload resolution* and is defined in 13.3. [*Example:*

```
double abs(double);
int abs(int);

abs(1);        // call abs(int);
abs(1.0);      // call abs(double);
```

—*end example*]

13.1 Overloadable declarations [over.load]

1 Not all function declarations can be overloaded. Those that cannot be overloaded are specified here. A program is ill-formed if it contains two such non-overloadable declarations in the same scope.

2 Certain function declarations cannot be overloaded:

— Function declarations that differ only in the return type cannot be overloaded.

— Member function declarations with the same name and the same parameter types cannot be overloaded if any of them is a `static` member function declaration (9.5). The types of the implicit object parameters constructed for the member functions for the purpose of overload resolution (13.3.1) are not considered when comparing parameter types for enforcement of this rule. In contrast, if there is no `static` member function declaration among a set of member function declarations with the same name and the same parameter types, then these member function declarations can be overloaded if they differ in the type of their implicit object parameter. [*Example:* the following illustrates this distinction:

```
class X {
    static void f();
    void f();                 // ill-formed
    void f() const;           // ill-formed
    void f() const volatile;  // ill-formed
    void g();
    void g() const;           // Ok: no static g
    void g() const volatile;  // Ok: no static g
};
```

—*end example*]

3 [*Note:* as specified in 8.3.5, function declarations that have equivalent parameter declarations declare the same function and therefore cannot be overloaded:

— Parameter declarations that differ only in the use of equivalent typedef “types” are equivalent. A `typedef` is not a separate type, but only a synonym for another type (7.1.3). [*Example:*

```
typedef int Int;

void f(int i);
void f(Int i);                  // OK: redeclaration of f(int)
void f(int i) { /* ... */ }
void f(Int i) { /* ... */ }     // error: redefinition of f(int)
```

—*end example*]

Enumerations, on the other hand, are distinct types and can be used to distinguish overloaded function declarations. [*Example:*

```
enum E { a };

void f(int i) { /* ... */ }
void f(E i)   { /* ... */ }
```

—*end example*]

— Parameter declarations that differ only in a pointer `*` versus an array `[]` are equivalent. That is, the array declaration is adjusted to become a pointer declaration (8.3.5). Only the second and subsequent array dimensions are significant in parameter types (8.3.4). [*Example:*

```
f(char*);
f(char[]);      // same as f(char*);
f(char[7]);     // same as f(char*);
f(char[9]);     // same as f(char*);

g(char(*)[10]);
g(char[5][10]);  // same as g(char(*)[10]);
g(char[7][10]);  // same as g(char(*)[10]);
g(char(*)[20]);  // different from g(char(*)[10]);
```

—*end example*]

— Parameter declarations that differ only in the presence or absence of `const` and/or `volatile` are equivalent. That is, the `const` and `volatile` type-specifiers for each parameter type are ignored when determining which function is being declared, defined, or called. [*Example:*

```
typedef const int cInt;

int f (int);
int f (const int);      // redeclaration of f (int);
int f (int) { ... }     // definition of f (int)
int f (cInt) { ... }    // error: redefinition of f (int)
```

—*end example*]

Only the `const` and `volatile` type-specifiers at the outermost level of the parameter type specification are ignored in this fashion; `const` and `volatile` type-specifiers buried within a parameter type specification are significant and can be used to distinguish overloaded function declarations.[79] In particular, for any type `T`, "pointer to `T`," "pointer to `const T`," and "pointer to `volatile T`" are considered distinct parameter types, as are "reference to `T`," "reference to `const T`," and "reference to `volatile T`."

— Two parameter declarations that differ only in their default arguments are equivalent. [*Example:* consider the following:

[79] When a parameter type includes a function type, such as in the case of a paramater type that is a pointer to function, the `const` and `volatile` type-specifiers at the outermost level of the parameter type specifications for the inner function type are also ignored.

```
void f (int i, int j);
void f (int i, int j = 99);         // Ok: redeclaration of f (int, int)
void f (int i = 88);                // Ok: redeclaration of f (int, int)
void f ();                          // Ok: overloaded declaration of f

void prog ()
{
    f (1, 2);  // Ok: call f (int, int)
    f (1);     // Ok: call f (int, int)
    f ();      // Error: f (int, int) or f ()?
}
```

—*end example*] —*end note*]

13.2 Declaration matching [over.dcl]

1 Two function declarations of the same name refer to the same function if they are in the same scope and have equivalent parameter declarations (13.1). A function member of a derived class is *not* in the same scope as a function member of the same name in a base class. [*Example:*

```
class B {
public:
    int f(int);
};

class D : public B {
public:
    int f(char*);
};
```

Here `D::f(char*)` hides `B::f(int)` rather than overloading it.

```
void h(D* pd)
{
    pd->f(1);       // error:
                    // D::f(char*) hides B::f(int)
    pd->B::f(1);    // ok
    pd->f("Ben");   // ok, calls D::f
}
```

—*end example*]

2 A locally declared function is not in the same scope as a function in a containing scope. [*Example:*

```
int f(char*);
void g()
{
    extern f(int);
    f("asdf");  // error: f(int) hides f(char*)
                // so there is no f(char*) in this scope
}

void caller ()
{
    void callee (int, int);
    {
        void callee (int);  // hides callee (int, int)
        callee (88, 99);    // error: only callee (int) in scope
    }
)
```

—*end example*]

3 Different versions of an overloaded member function can be given different access rules. [*Example:*

```
class buffer {
private:
    char* p;
    int size;

protected:
    buffer(int s, char* store) { size = s; p = store; }
    // ...

public:
    buffer(int s) { p = new char[size = s]; }
    // ...
};
```

—*end example*]

13.3 Overload resolution [over.match]

1 Overload resolution is a mechanism for selecting the best function to call given a list of expressions that are to be the arguments of the call and a set of *candidate functions* that can be called based on the context of the call. The selection criteria for the best function are the number of arguments, how well the arguments match the types of the parameters of the candidate function, how well (for nonstatic member functions) the object matches the implied object parameter, and certain other properties of the candidate function. [*Note:* the function selected by overload resolution is not guaranteed to be appropriate for the context. Other restrictions, such as the accessibility of the function, can make its use in the calling context ill-formed.]

2 Overload resolution selects the function to call in five distinct contexts within the language:

— invocation of a function named in the function call syntax (13.3.1.1.1);

— invocation of a function call operator, a pointer-to-function conversion function, or a reference-to-function conversion function of a class object named in the function call syntax (13.3.1.1.2);

— invocation of the operator referenced in an expression (13.3.1.2);

— invocation of a constructor for direct-initialization (8.5) of a class object (13.3.1.4); and

— invocation of a user-defined conversion for copy-initialization (8.5) of a class object, or initialization of an object of a built-in type from an expression of class type (13.3.1.3).

3 Each of these contexts defines the set of candidate functions and the list of arguments in its own unique way. But, once the candidate functions and argument lists have been identified, the selection of the best function is the same in all cases:

— First, a subset of the candidate functions—those that have the proper number of arguments and meet certain other conditions—is selected to form a set of *viable functions*.

— Then the best viable function is selected based on the implicit conversion sequences (13.3.3.1) needed to match each argument to the corresponding parameter of each viable function.

4 If a best viable function exists and is unique, overload resolution succeeds and produces it as the result. Otherwise overload resolution fails and the invocation is ill-formed.

13.3.1 Candidate functions and argument lists [over.match.funcs]

1 The following subclauses describe the set of candidate functions and the argument list submitted to overload resolution in each of the five contexts in which overload resolution is used. The source transformations and constructions defined in these subclauses are only for the purpose of describing the overload resolution process. An implementation is not required to use such transformations and constructions.

2 The set of candidate functions can contain both member and non-member functions to be resolved against the same argument list. So that argument and parameter lists are comparable within this heterogeneous set, a member function is considered to have an extra parameter, called the *implicit object parameter*, which represents the object for which the member function has been called. For the purposes of overload resolution, both static and non-static member functions have an implicit object parameter, but constructors do not.

3 Similarly, when appropriate, the context can construct an argument list that contains an *implied object argument* to denote the object to be operated on. Since arguments and parameters are associated by position within their respective lists, the convention is that the implicit object parameter, if present, is always the first parameter and the implied object argument, if present, is always the first argument.

4 For non-static member functions, the type of the implicit object parameter is "reference to *cv* `X`" where `X` is the class that defines the member function and *cv* is the cv-qualification on the member function declaration. [*Example:* for a `const` member function of class `X`, the extra parameter is assumed to have type "reference to `const X`".] For static member functions, the implicit object parameter is considered to match any object (since if the function is selected, the object is discarded).

5 During overload resolution, the implied object argument is indistinguishable from other arguments. The implicit object parameter, however, retains its identity since conversions on the corresponding argument shall obey these additional rules:

— no temporary object can be introduced to hold the argument for the implicit object parameter;

— no user-defined conversions can be applied to achieve a type match with it; and

— even if the implicit object parameter is not `const`-qualified, an rvalue temporary can be bound to the parameter as long as in all other respects the temporary can be converted to the type of the implicit object parameter.

6 In each case where a candidate is a function template, candidate template functions are generated using template argument deduction (14.10.3, 14.10.2). Those candidates are then handled as candidate functions in the usual way.[80] A given name can refer to one or more function templates and also to a set of overloaded non-template functions. In such a case, the candidate functions generated from each function template are combined with the set of non-template candidate functions.

13.3.1.1 Function call syntax [over.match.call]

1 Recall from 5.2.2, that a *function call* is a *postfix-expression*, possibly nested arbitrarily deep in parentheses, followed by an optional *expression-list* enclosed in parentheses:

`(`...`(`$_{\texttt{opt}}$ *postfix-expression* `)`...`)`$_{\texttt{opt}}$ `(`*expression-list*$_{\texttt{opt}}$`)`

Overload resolution is required if the *postfix-expression* yields the name of a function, a function template (14.10), an object of class type, or a set of pointers-to-function.

2 Subclauses 13.3.1.1.1 and 13.3.1.1.2, respectively, describe how overload resolution is used in the first two cases to determine the function to call.

3 The third case arises from a *postfix-expression* of the form `&F`, where `F` names a set of overloaded functions. In the context of a function call, the set of functions named by `F` shall contain only non-member functions and static member functions[81]. And in this context using `&F` behaves the same as using the name `F` by itself. Thus, `(&F)(`*expression-list*$_{\texttt{opt}}$`)` is simply `(F)(`*expression-list*$_{\texttt{opt}}$`)`, which is discussed in 13.3.1.1.1. (The resolution of `&F` in other contexts is described in 13.4.)

[80] The process of argument deduction fully determines the parameter types of the template functions, i.e., the parameters of template functions contain no template parameter types. Therefore the template functions can be treated as normal (non-template) functions for the remainder of overload resolution.
[81] If `F` names a non-static member function, `&F` is a pointer-to-member, which cannot be used with the function call syntax.

13.3.1.1.1 Call to named function [over.call.func]

1 Of interest in this subclause are only those function calls in which the *postfix-expression* ultimately contains a name that denotes one or more functions that might be called. Such a *postfix-expression*, perhaps nested arbitrarily deep in parentheses, has one of the following forms:

postfix-expression:
  postfix-expression `.` *id-expression*
  postfix-expression `->` *id-expression*
  primary-expression

These represent two syntactic subcategories of function calls: qualified function calls and unqualified function calls.

2 In qualified function calls, the name to be resolved is an *id-expression* and is preceded by an `->` or `.` operator. Since the construct `A->B` is generally equivalent to `(*A).B`, the rest of this clause assumes, without loss of generality, that all member function calls have been normalized to the form that uses an object and the `.` operator. Furthermore, this clause assumes that the *postfix-expression* that is the left operand of the `.` operator has type "*cv* `T`" where `T` denotes a class[82)]. Under this assumption, the *id-expression* in the call is looked up as a member function of `T` following the rules for looking up names in classes (10). If a member function is found, that function and its overloaded declarations constitute the set of candidate functions[83)]. The argument list is the *expression-list* in the call augmented by the addition of the left operand of the `.` operator in the normalized member function call as the implied object argument.

3 In unqualified function calls, the name is not qualified by an `->` or `.` operator and has the more general form of a *primary-expression.* The name is looked up in the context of the function call following the normal rules for name lookup. If the name resolves to a non-member function declaration, that function and its overloaded declarations constitute the set of candidate functions[84)]. The argument list is the same as the *expression-list* in the call. If the name resolves to a nonstatic member function, then the function call is actually a member function call. If the keyword `this` is in scope and refers to the class of that member function, or a derived class thereof, then the function call is transformed into a normalized qualified function call using `(*this)` as the *postfix-expression* to the left of the `.` operator. The candidate functions and argument list are as described for qualified function calls above. If the keyword `this` is not in scope or refers to another class, then name resolution found a static member of some class `T`. In this case, all overloaded declarations of the function name in `T` become candidate functions and a contrived object of type `T` becomes the implied object argument[85)]. The call is ill-formed, however, if overload resolution selects one of the non-static member functions of `T` in this case.

13.3.1.1.2 Call to object of class type [over.call.object]

1 If the *primary-expression* `E` in the function call syntax evaluates to a class object of type "*cv* `T`", then the set of candidate functions includes at least the function call operators of `T`. The function call operators of `T` are obtained by ordinary lookup of the name `operator()` in the context of `(E).operator()`[86)].

2 In addition, for each conversion function declared in `T` of the form

`operator` *conversion-type-id* `()` *cv-qualifier*`;`

where *conversion-type-id* denotes the type "pointer to function with parameters of type `P1`,...,`Pn` and returning `R`" or type "reference to function with parameters of type `P1`,...,`Pn` and returning `R`", a *surrogate*

82) Note that cv-qualifiers on the type of objects are significant in overload resolution for both lvalue and rvalue objects.

83) Because of the usual name hiding rules, these will all be declared in `T` or they will all be declared in the same base class of `T`; see 10.2.

84) Because of the usual name hiding rules, these will be introduced by declarations or by using directives all found found in the same block or all found at namespace scope.

85) An implied object argument must be contrived to correspond to the implicit object parameter attributed to member functions during overload resolution. It is not used in the call to the selected function. Since the member functions all have the same implicit object parameter, the contrived object will not be the cause to select or reject a function.

86) Because of the usual name hiding rules, these will all be declared in `T` or they will all be declared in the same base class of `T`.

call function with the unique name *call-function* and having the form

```
R call-function (conversion-type-id F, P1 a1,…,Pn an) { return F (a1,…,an); }
```

is also considered as a candidate function. Similarly, surrogate call functions are added to the set of candidate functions for each conversion function declared in an accessible base class provided the function is not hidden within `T` by another intervening declaration[87].

3 If such a surrogate call function is selected by overload resolution, its body, as defined above, will be executed to convert `E` to the appropriate function and then to invoke that function with the arguments of the call.

4 The argument list submitted to overload resolution consists of the argument expressions present in the function call syntax preceded by the implied object argument `(E)`. [*Note:* when comparing the call against the function call operators, the implied object argument is compared against the implicit object parameter of the function call operator. When comparing the call against a surrogate call function, the implied object argument is compared against the first parameter of the surrogate call function. The conversion function from which the surrogate call function was derived will be used in the conversion sequence for that parameter since it converts the implied object argument to the appropriate function pointer or reference required by that first parameter.] [*Example:*

```
int f1(int);
int f2(float);
typedef int (*fp1)(int);
typedef int (*fp2)(float);
struct A {
    operator fp1() { return f1; }
    operator fp2() { return f2; }
} a;
int i = a(1);     // Calls f1 via pointer returned from
                  // conversion function
```

—*end example*]

13.3.1.2 Operators in expressions [over.match.oper]

1 If no operand of an operator in an expression has a type that is a class or an enumeration, the operator is assumed to be a built-in operator and interpreted according to clause 5. [*Note:* because `.`, `.*`, `::`, and `?:` cannot be overloaded, these operators are always built-in operators interpreted according to clause 5.] [*Example:*

```
class String {
public:
   String (const String&);
   String (char*);
         operator char* ();
};
String operator + (const String&, const String&);

void f(void)
{
   char* p= "one" + "two"; // ill-formed because neither
                           // operand has user defined type
   int I = 1 + 1;          // Always evaluates to 2 even if
                           // user defined types exist which
                           // would perform the operation.
}
```

87) Note that this construction can yield candidate call functions that cannot be differentiated one from the other by overload resolution because they have identical declarations or differ only in their return type. The call will be ambiguous if overload resolution cannot select a match to the call that is uniquely better than such undifferentiable functions.

—*end example*]

2 If either operand has a type that is a class or an enumeration, a user-defined operator function might be declared that implements this operator or a user-defined conversion can be necessary to convert the operand to a type that is appropriate for a built-in operator. In this case, overload resolution is used to determine which operator function or builtin operator is to be invoked to implement the operator. Therefore, the operator notation is first transformed to the equivalent function-call notation as summarized in Table 8 (where @ denotes one of the operators covered in the specified subclause).

Table 8—relationship between operator and function call notation

Subclause	Expression	As member function	As non-member function
13.5.1	`@a`	`(a).operator@ ()`	`operator@ (a)`
13.5.2	`a@b`	`(a).operator@ (b)`	`operator@ (a, b)`
13.5.3	`a=b`	`(a).operator= (b)`	
13.5.5	`a[b]`	`(a).operator[](b)`	
13.5.6	`a->`	`(a).operator-> ()`	
13.5.7	`a@`	`(a).operator@ (0)`	`operator@ (a, 0)`

3 For a type `T` whose fully-qualified name is `::N1::...::Nn::C1::...::Cm::T` where each `Ni` is a namespace name and each `Ci` is a class name, the fully-qualified namespace name `::N1::...::Nn` is called the "namespace of the type `T`." To look up `X` in the "context of the namespace of the type `T`" means to perform the qualified name lookup of `::N1::...::Nn::X` (13.3.1.1.1).

4 For a unary operator @ with an operand of type `T1` or reference to *cv* `T1`, and for a binary operator @ with a left operand of type `T1` or reference to *cv* `T1` and a right operand of type `T2` or reference to *cv* `T2`, three sets of candidate functions, designated *member candidates*, *non-member candidates* and *built-in candidates*, are constructed as follows:

— If `T1` is a class type, the set of member candidates is the result of the qualified lookup of `T1::operator@` (13.3.1.1.1); otherwise, the set of member candidates is empty.

— The set of non-member candidates is the union of the functions found in the following name lookups:

  — The unqualified `operator@` is looked up in the context of the expression according to the usual rules for name lookup except that all member functions are ignored.

  — For each type `Z`, where `Z` is either a `Ti` of class type or a direct or indirect base class of a `Ti` of class type, `operator@` is looked up in the context of type `Z` according to the usual rules for name lookup.

  — For each `Ti` of enumeration type, `operator@` is looked up in the context of the namespace of that type according to the usual rules for name lookup.

— For the operator `,`, the unary operator `&`, or the operator `->`, the built-in candidates set is empty. For all other operators, the built-in candidates include all of the candidate operator functions defined in 13.6 that, compared to the given operator,

  — have the same operator name, and

  — accept the same number of operands, and

  — accept operand types to which the given operand or operands can be converted according to 13.3.3.1.

5 For the built-in assignment operators, conversions of the left operand are restricted as follows:

— no temporaries are introduced to hold the left operand, and

— no user-defined conversions are applied to achieve a type match with it.

6 For all other operators, no such restrictions apply.

7 The set of candidate functions for overload resolution is the union of the member candidates, the non-member candidates, and the built-in candidates. The argument list contains all of the operands of the operator. The best function from the set of candidate functions is selected according to 13.3.2 and 13.3.3.[88) [*Example:*

```
        struct A {
            operator int();
        };
        A operator+(const A&, const A&);
        void m() {
            A a, b;
            a + b;        // a.operator+(b) chosen over int(a) + int(b)
        }
```

—*end example*]

8 If a built-in candidate is selected by overload resolution, any class operands are first converted to the appropriate type for the operator. Then the operator is treated as the corresponding built-in operator and interpreted according to clause 5.

9 The second operand of operator `->` is ignored in selecting an `operator->` function, and is not an argument when the `operator->` function is called. When `operator->` returns, the built-in operator `->` is applied to the value returned, with the original second operand.

10 If the operator is the operator `,`, the unary operator `&`, or the operator `->`, and overload resolution is unsuccessful, then the operator is assumed to be the built-in operator and interpreted according to clause 5.

11 [*Note:* the look up rules for operators in expressions are different than the lookup rules for operator function names in a function call, as shown in the following example:

```
struct A { };
void operator + (A, A);

struct B {
  void operator + (B);
  void f ();
};

A a;

void B::f() {
  operator+ (a,a);      // ERROR - global operator hidden by member
  a + a;                // OK - calls global operator+
}
```

—*end note*]

13.3.1.3 Initialization by user-defined conversions [over.match.user]

1 Under the conditions specified in 8.5 and 8.5.3, as part of an initialization a user-defined conversion can be invoked to convert the initializer expression to the type of an object or temporary being initialized. Overload resolution is used to select the user-defined conversion to be invoked. Assuming that "*cv1* `T`" is the type of the object or temporary being initialized, the candidate functions are selected as follows:

— When `T` is a class type, the constructors of `T` are candidate functions.

— When the type of the initializer expression is a class type "*cv* `S`", the conversion functions of `S` and its

88) If the set of candidate functions is empty, overload resolution is unsuccessful.

base classes are considered. Those that are not hidden within S and yield type "*cv2* T" or a type that can be converted to type "*cv2* T," for any *cv2* that is the same cv-qualification as, or lesser cv-qualification than, *cv1*, via a standard conversion sequence (13.3.3.1.1) are candidate functions.

2 In both cases, the argument list has one argument, which is the initializer expression. [*Note:* this argument will be compared against the first parameter of the constructors and against the implicit object parameter of the conversion functions.]

3 Because only one user-defined conversion is allowed in an implicit conversion sequence, special rules apply when selecting the best user-defined conversion (13.3.3, 13.3.3.1).

13.3.1.4 Initialization by constructor [over.match.ctor]

1 When objects of class type are direct-initialized (8.5), overload resolution selects the constructor. The candidate functions are all the constructors of the class of the object being initialized. The argument list is the *expression-list* within the parentheses of the initializer.

2 [*Note:* when no constructor for class T accepts the given type, no attempt is made to find other constructors to convert the *assignment-expression* into a type that can be converted to T. [*Example:*

```
class T {
public:
        T();
        // ...
};

class C : T {
public:
        C(int);
        // ...
};
T a = 1;                // ill-formed: T(C(1)) not tried
```

—*end example*] —*end note*]

13.3.2 Viable functions [over.match.viable]

1 From the set of candidate functions constructed for a given context (13.3.1), a set of viable functions is chosen, from which the best function will be selected by comparing argument conversion sequences for the best fit (13.3.3). The selection of viable functions considers relationships between arguments and function parameters other than the ranking of conversion sequences.

2 First, to be a viable function, a candidate function shall have enough parameters to agree in number with the arguments in the list.

— If there are *m* arguments in the list, all candidate functions having exactly *m* parameters are viable.

— A candidate function having fewer than *m* parameters is viable only if it has an ellipsis in its parameter list (8.3.5). For the purposes of overload resolution, any argument for which there is no corresponding parameter is considered to ‘‘match the ellipsis’’ (13.3.3.1.3) .

— A candidate function having more than *m* parameters is viable only if the *(m+1)*–st parameter has a default argument (8.3.6).[89] For the purposes of overload resolution, the parameter list is truncated on the right, so that there are exactly *m* parameters.

3 Second, for F to be a viable function, there shall exist for each argument an *implicit conversion sequence* (13.3.3.1) that converts that argument to the corresponding parameter of F. If the parameter has reference type, the implicit conversion sequence includes the operation of binding the reference, and the fact that a reference to non-const cannot be bound to an rvalue can affect the viability of the function (see

[89] According to subclause 8.3.6, parameters following the *(m+1)*–st parameter must also have default arguments.

13.3.3.1.4).

13.3.3 Best Viable Function [over.match.best]

1 Let ICS*i*(F) denote the implicit conversion sequence that converts the *i*-th argument in the list to the type of the *i*-th parameter of viable function F. Subclause 13.3.3.1 defines the implicit conversion sequences and subclause 13.3.3.2 defines what it means for one implicit conversion sequence to be a better conversion sequence or worse conversion sequence than another. Given these definitions, a viable function F1 is defined to be a *better* function than another viable function F2 if for all arguments *i*, ICS*i*(F1) is not a worse conversion sequence than ICS*i*(F2), and then

— for some argument *j*, ICS*j*(F1) is a better conversion sequence than ICS*j*(F2), or, if not that,

— F1 is a non-template function and F2 is a template function, or, if not that,

— F1 and F2 are template functions with the same signature, and the function template for F1 is more specialized than the template for F2 according to the partial ordering rules described in _temp.over.order_, or, if not that,

— the context is an initialization by user-defined conversion (see 8.5 and 13.3.1.3) and the standard conversion sequence from the return type of F1 to the destination type (i.e., the type of the entity being initialized) is a better conversion sequence than the standard conversion sequence from the return type of F2 to the destination type. [*Example:*

```
struct A {
    A();
    operator int();
    operator double();
} a;
int i = a;     // a.operator int() followed by no conversion is better
               // than a.operator double() followed by a conversion
               // to int
float x = a;   // ambiguous: both possibilities require conversions,
               // and neither is better than the other
```

—*end example*]

2 If there is exactly one viable function that is a better function than all other viable functions, then it is the one selected by overload resolution; otherwise the call is ill-formed[90].

3 [*Example:*

[90] The algorithm for selecting the best viable function is linear in the number of viable functions. Run a simple tournament to find a function W that is not worse than any opponent it faced. Although another function F that W did not face might be better than W, F cannot be the best function because at some point in the tournament F encountered another function G such that F was not better than G. Hence, W is either the best function or there is no best function. So, make a second pass over the viable functions to verify that W is better than all other functions.

```
        void Fcn(const int*,  short);
        void Fcn(int*, int);

        int i;
        short s = 0;

        Fcn(&i, s);     // is ambiguous because
                        // &i -> int* is better than &i -> const int*
                        // but s -> short is also better than s -> int

        Fcn(&i, 1L);    // calls Fcn(int*, int), because
                        // &i -> int* is better than &i -> const int*
                        // and 1L -> short and 1L -> int are indistinguishable

        Fcn(&i,'c');    // calls Fcn(int*, int), because
                        // &i -> int* is better than &i -> const int*
                        // and 'c' -> int is better than 'c' -> short
```

—*end example*]

13.3.3.1 Implicit conversion sequences [over.best.ics]

1 An *implicit conversion sequence* is a sequence of conversions used to convert an argument in a function call to the type of the corresponding parameter of the function being called. The sequence of conversions is governed by the rules for initialization of an object or reference by a single expression (8.5, 8.5.3).

2 Implicit conversion sequences are concerned only with the type, cv-qualification, and lvalue-ness of the argument and how these are converted to match the corresponding properties of the parameter. Other properties, such as the lifetime, storage class, alignment, or accessibility of the argument and whether or not the argument is a bit-field are ignored. So, although an implicit conversion sequence can be defined for a given argument-parameter pair, the conversion from the argument to the parameter might still be ill-formed in the final analysis.

3 Except in the context of an initialization by user-defined conversion (13.3.1.3), a well-formed implicit conversion sequence is one of the following forms:

— a *standard conversion sequence* (13.3.3.1.1),

— a *user-defined conversion sequence* (13.3.3.1.2), or

— an *ellipsis conversion sequence* (13.3.3.1.3).

4 In the context of an initialization by user-defined conversion (i.e., when considering the argument of a user-defined conversion function; see 13.3.1.3), only standard conversion sequences and ellipsis conversion sequences are allowed.

5 When initializing a reference, the operation of binding the reference to an object or temporary occurs after any conversion. The binding operation is not a conversion, but it is considered to be part of a standard conversion sequence, and it can affect the rank of the conversion sequence. See 13.3.3.1.4.

6 In all contexts, when converting to the implicit object parameter or when converting to the left operand of an assignment operation only standard conversion sequences that create no temporary object for the result are allowed.

7 If no conversions are required to match an argument to a parameter type, the implicit conversion sequence is the standard conversion sequence consisting of the identity conversion (13.3.3.1.1).

8 If no sequence of conversions can be found to convert an argument to a parameter type or the conversion is otherwise ill-formed, an implicit conversion sequence cannot be formed.

9 If several different sequences of conversions exist that each convert the argument to the parameter type, the implicit conversion sequence is a sequence among these that is not worse than all the rest according to 13.3.3.2[91]. If that conversion sequence in not better than all the rest and a function that uses such an

[91] This rule prevents a function from becoming non-viable because of an ambiguous conversion sequence for one of its parameters. Consider this example,

implicit conversion sequence is selected as the best viable function, then the call will be ill-formed because the conversion of one of the arguments in the call is ambiguous.

10 The three forms of implicit conversion sequences mentioned above are defined in the following subclauses.

13.3.3.1.1 Standard conversion sequences [over.ics.scs]

1 Table 9 summarizes the conversions defined in clause 4 and partitions them into four disjoint categories: Lvalue Transformation, Qualification Adjustment, Promotion, and Conversion. Note that these categories are orthogonal with respect to lvalue-ness, cv-qualification, and data representation: the Lvalue Transformations do not change the cv-qualification or data representation of the type; the Qualification Adjustments do not change the lvalue-ness or data representation of the type; and the Promotions and Conversions do not change the lvalue-ness or cv-qualification of the type.

2 A standard conversion sequence is either the Identity conversion by itself or consists of one to four conversions from the other four categories. At most one conversion from each category is allowed in a single standard conversion sequence. If there are two or more conversions in the sequence, the conversions are applied in the canonical order: **Lvalue Transformation**, **Promotion**, **Conversion**, **Qualification Adjustment**.

3 Each conversion in Table 9 also has an associated rank (Exact Match, Promotion, or Conversion). These are used to rank standard conversion sequences (13.3.3.2). The rank of a conversion sequence is determined by considering the rank of each conversion in the sequence and the rank of any reference binding (13.3.3.1.4). If any of those has Conversion rank, the sequence has Conversion rank; otherwise, if any of those has Promotion rank, the sequence has Promotion rank; otherwise, the sequence has Exact Match rank.

```
        class B;
        class A { A (B&); };
        class B { operator A (); };
        class C { C (B&); };
        f(A) { }
        f(C) { }
        B b;
        f(b);   // ambiguous since b -> C via constructor and
                // b -> A via constructor or conversion function.
```

If it were not for this rule, `f(A)` would be eliminated as a viable function for the call `f(b)` causing overload resolution to select `f(C)` as the function to call even though it is not clearly the best choice. On the other hand, if an `f(B)` were to be declared then `f(b)` would resolved to that `f(B)` because the exact match with `f(B)` is better than any of the sequences required to match `f(A)`.

Table 9—conversions

Conversion	Category	Rank	Subclause
No conversions required	Identity	Exact Match	
Lvalue-to-rvalue conversion	Lvalue Transformation		4.1
Array-to-pointer conversion			4.2
Function-to-pointer conversion			4.3
Qualification conversions	Qualification Adjustment		4.4
Integral promotions	Promotion	Promotion	4.5
Floating point promotion			4.6
Integral conversions	Conversion	Conversion	4.7
Floating point conversions			4.8
Floating-integral conversions			4.9
Pointer conversions			4.10
Pointer to member conversions			4.11
Base class conversion			4.12
Boolean conversions			4.13

13.3.3.1.2 User-defined conversion sequences [over.ics.user]

1 A user-defined conversion sequence consists of an initial standard conversion sequence followed by a user-defined conversion (12.3) followed by a second standard conversion sequence. If the user-defined conversion is specified by a constructor (12.3.1), the initial standard conversion sequence converts the source type to the type required by the argument of the constructor. If the user-defined conversion is specified by a conversion function (12.3.2), the initial standard conversion sequence converts the source type to the implicit object parameter of the conversion function.

2 The second standard conversion sequence converts the result of the user-defined conversion to the target type for the sequence. Since an implicit conversion sequence is an initialization, the special rules for initialization by user-defined conversion apply when selecting the best user-defined conversion for a user-defined conversion sequence (see 13.3.3 and 13.3.3.1).

3 If the user-defined conversion is specified by a template conversion function, the second standard conversion sequence must have exact match rank.

4 A conversion of an expression of class type to the same class type or to a base class of that type is a standard conversion rather than a user-defined conversion in spite of the fact that a copy constructor (i.e., a user-defined conversion function) is called.

13.3.3.1.3 Ellipsis conversion sequences [over.ics.ellipsis]

1 An ellipsis conversion sequence occurs when an argument in a function call is matched with the ellipsis parameter specification of the function called.

13.3.3.1.4 Reference binding [over.ics.ref]

1 The operation of binding a reference is not a conversion, but for the purposes of overload resolution it is considered to be part of a standard conversion sequence (specifically, it is the last step in such a sequence).

2 A standard conversion sequence cannot be formed if it requires binding a reference to non-`const` to an rvalue (except when binding an implicit object parameter; see the special rules for that case in 13.3.1). [*Note:* this means, for example, that a candidate function cannot be a viable function if it has a non-`const` reference parameter (other than the implicit object parameter) and the corresponding argument is a

temporary or would require one to be created to initialize the reference (see 8.5.3).]

3 Other restrictions on binding a reference to a particular argument do not affect the formation of a standard conversion sequence, however. [*Example:* a function with a "reference to `int`" parameter can be a viable candidate even if the corresponding argument is an `int` bit-field. The formation of implicit conversion sequences treats the `int` bit-field as an `int` lvalue and finds an exact match with the parameter. If the function is selected by overload resolution, the call will nonetheless be ill-formed because of the prohibition on binding a non-`const` reference to a bit-field (8.5.3).]

4 A reference binding in general has no effect on the rank of a standard conversion sequence, but there are two exceptions:

— the binding of a reference to a (possibly cv-qualified) class to an expression of a (possibly cv-qualified) class derived from that class gives the overall standard conversion sequence Conversion rank. [*Example:*

```
struct A {};
struct B : public A {} b;
int f(A&);
int f(B&);
int i = f(b);     // Calls f(B&), an exact match, rather than
                  // f(A&), a conversion
```

—*end example*]

— the binding of a reference to an expression that is *reference-compatible with added qualification* influences the rank of a standard conversion; see 13.3.3.2 and 8.5.3.

13.3.3.2 Ranking implicit conversion sequences [over.ics.rank]

1 This clause defines a partial ordering of implicit conversion sequences based on the relationships *better conversion sequence* and *better conversion*. If an implicit conversion sequence S1 is defined by these rules to be a better conversion sequence than S2, then it is also the case that S2 is a *worse conversion sequence* than S1. If conversion sequence S1 is neither better than nor worse than conversion sequence S2, S1 and S2 are said to be *indistinguishable conversion sequences*.

2 When comparing the basic forms of implicit conversion sequences (as defined in 13.3.3.1)

— a standard conversion sequence (13.3.3.1.1) is a better conversion sequence than a user-defined conversion sequence or an ellipsis conversion sequence, and

— a user-defined conversion sequence (13.3.3.1.2) is a better conversion sequence than an ellipsis conversion sequence (13.3.3.1.3).

3 Two implicit conversion sequences of the same form are indistinguishable conversion sequences unless one of the following rules apply:

— Standard conversion sequence `S1` is a better conversion sequence than standard conversion sequence `S2` if

  — `S1` is a proper subsequence of `S2`, or, if not that,

  — the rank of `S1` is better than the rank of `S2` (by the rules defined below), or, if not that,

  — `S1` and `S2` differ only in their qualification conversion and they yield types identical except for cv-qualifiers and `S2` adds all the qualifiers that `S1` adds (and in the same places) and `S2` adds yet more cv-qualifiers than `S1`, or the similar case with reference binding[92]. [*Example:*

[92] See the definition of *reference-compatible with added qualification* in 8.5.3.

```
int f(const int *);
int f(int *);
int g(const int &);
int g(int &);
int i;
int j = f(&i);    // Calls f(int *)
int k = g(i);     // Calls g(int &)

class X {
public:
    void f() const;
    void f();
};
void g(const X& a, X b)
{
    a.f();        // Calls X::f() const
    b.f();        // Calls X::f()
}
```

—*end example*]

— User-defined conversion sequence U1 is a better conversion sequence than another user-defined conversion sequence U2 if they contain the same user-defined conversion operator or constructor and if the second standard conversion sequence of U1 is better than the second standard conversion sequence of U2. [*Example:*

```
struct A {
    operator short();
} a;
int f(int);
int f(float);
int i = f(a);     // Calls f(int), because short -> int is
                  // better than short -> float.
```

—*end example*]

4 Standard conversions are ordered by their ranks: an Exact Match is a better conversion than a Promotion, which is a better conversion than a Conversion. Two conversions with the same rank are indistinguishable unless one of the following rules applies:

— A conversion that is not a conversion of a pointer, or pointer to member, to bool is better than another conversion that is such a conversion.

— If class B is derived directly or indirectly from class A, conversion of B* to A* is better than conversion of B* to void*, and conversion of A* to void* is better than conversion of B* to void*.

— If class B is derived directly or indirectly from class A and class C is derived directly or indirectly from B,

 — conversion of C* to B* is better than conversion of C* to A*,

 — binding of an expression of type C to a reference of type B& is better than binding an expression of type C to a reference of type A&,

 — conversion of A::* to B::* is better than conversion of A::* to C::*,

 — conversion of C to B is better than conversion of C to A,

 — conversion of B* to A* is better than conversion of C* to A*,

 — binding an expression of type B to a reference of type A& is better than binding an expression of type C to a reference of type A&,

 — conversion of B::* to C::* is better than conversion of A::* to C::*, and

— conversion of B to A is better than conversion of C to A. [*Example:*

```
struct A {};
struct B : public A {};
struct C : public B {};
C *pc;
int f(A *);
int f(B *);
int i = f(pc);    // Calls f(B *)
```

—*end example*]

13.4 Address of overloaded function [over.over]

1 A use of an overloaded function name without arguments is resolved in certain contexts to a pointer to function or pointer to member function for a specific function from the overload set. The function selected is the one whose type matches the target type required in the context. It is required that exactly one function matches the target type. The target can be

— an object being initialized (8.5),

— the left side of an assignment (5.17),

— a parameter of a function (5.2.2),

— a parameter of a user-defined operator (13.5),

— the return value of a function, operator function, or conversion (6.6.3), or

— an explicit type conversion (5.2.3, 5.4).

An overloaded function name shall not be used without arguments in contexts other than those listed. The reference to the overloaded function name can be preceded by &.

2 If the name is a function template, template argument deduction is done (14.10.2), and if the argument deduction succeeds, the deduced template arguments are used to generate a single template function, which is added to the set of overloaded functions considered.

3 Non-member functions and static member functions match targets of type "pointer-to-function;" nonstatic member functions match targets of type "pointer-to-member-function." If a nonstatic member function is selected, the reference to the overloaded function name is required to have the form of a pointer to member as described in 5.3.1.

4 [*Note:* if f() and g() are both overloaded functions, the cross product of possibilities must be considered to resolve f(&g), or the equivalent expression f(g).

5 [*Example:*

```
int f(double);
int f(int);
(int (*)(int))&f;           // cast expression as selector
int (*pfd)(double) = &f;    // selects f(double)
int (*pfi)(int) = &f;       // selects f (int)
int (*pfe)(...) = &f;       // error: type mismatch
```

The last initialization is ill-formed because no f() with type int(...) has been defined, and not because of any ambiguity. —*end example*]

6 Also note that there are no standard conversions (4) of one pointer-to-function type into another. In particular, even if B is a public base of D we have

```
D* f();
B* (*p1)() = &f;       // error
```

```
void g(D*);
void (*p2)(B*) = &g;   // error
```

7 Note that if the target type is a pointer to member function, the function type of the pointer to member is used to select the member function from a set of overloaded member functions. [*Example:*

```
struct X {
    int f(int);
    static int f(long);
};

int (X::*p1)(int)  = &X::f;   // OK
int    (*p2)(int)  = &X::f;   // error: mismatch
int    (*p3)(long) = &X::f;   // OK
int (X::*p4)(long) = &X::f;   // error: mismatch
int (X::*p5)(int)  = &(X::f); // error: wrong syntax for
                              // pointer to member
int    (*p6)(long) = &(X::f); // OK
```

—*end example*] —*end note*]

13.5 Overloaded operators [over.oper]

1 A function declaration having one of the following *operator-function-id*s as its name declares an *operator function*. An operator function is said to *implement* the operator named in its *operator-function-id*.

operator-function-id:
    `operator` *operator*

operator: one of

```
new  delete    new[]     delete[]
+    -    *    /    %    ^    &    |    ~
!    =    <    >    +=   -=   *=   /=   %=
^=   &=   |=   <<   >>   >>=  <<=  ==   !=
<=   >=   &&   ||   ++   --   ,    ->*  ->
()   []
```

[*Note:* the last two operators are function call (5.2.2) and subscripting (5.2.1).]

2 Both the unary and binary forms of

```
+    -    *     &
```

can be overloaded.

3 The following operators cannot be overloaded:

```
.    .*   ::    ?:
```

nor can the preprocessing symbols # and ## (16).

4 Operator functions are usually not called directly; instead they are invoked to evaluate the operators they implement (13.5.1 - 13.5.7). They can be explicitly called, however, using the *operator-function-id* as the name of the function in the function call syntax (5.2.2). [*Example:*

```
complex z = a.operator+(b);  // complex z = a+b;
void* p = operator new(sizeof(int)*n);
```

—*end example*]

5 The allocation and deallocation functions, `operator new`, `operator new[]`, `operator delete` and `operator delete[]`, are described completely in 12.5. The attributes and restrictions found in the rest of this section do not apply to them unless explicitly stated in 12.5.

6 An operator function shall either be a non-static member function or be a non-member function and have at least one parameter whose type is a class, a reference to a class, an enumeration, or a reference to an enumeration. It is not possible to change the precedence, grouping, or number of operands of operators. The meaning of the operators `=`, (unary) `&`, and `,` (comma), predefined for each type, can be changed for specific types by defining operator functions that implement these operators. Operator functions are inherited the same as other functions, but because an instance of `operator=` is automatically constructed for each class (12.8, 13.5.3), `operator=` is never inherited by a class from its bases.

7 The identities among certain predefined operators applied to basic types (for example, `++a` ≡ `a+=1`) need not hold for operator functions. Some predefined operators, such as `+=`, require an operand to be an lvalue when applied to basic types; this is not required by operator functions.

8 An operator function cannot have default arguments (8.3.6), except where explicitly stated below. Operator functions cannot have more or fewer parameters than the number required for the corresponding operator, as described in the rest of this section.

9 Operators not mentioned explicitly below in 13.5.3 to 13.5.7 act as ordinary unary and binary operators obeying the rules of section 13.5.1 or 13.5.2.

13.5.1 Unary operators [over.unary]

1 A prefix unary operator shall be implemented by a non-static member function (9.4) with no parameters or a non-member function with one parameter. Thus, for any prefix unary operator `@`, `@x` can be interpreted as either `x.operator@()` or `operator@(x)`. If both forms of the operator function have been declared, the rules in 13.3.1.2 determine which, if any, interpretation is used. See 13.5.7 for an explanation of the postfix unary operators `++` and `--`.

2 The unary and binary forms of the same operator are considered to have the same name. [*Note:* consequently, a unary operator can hide a binary operator from an enclosing scope, and vice versa.]

13.5.2 Binary operators [over.binary]

1 A binary operator shall be implemented either by a non-static member function (9.4) with one parameter or by a non-member function with two parameters. Thus, for any binary operator `@`, `x@y` can be interpreted as either `x.operator@(y)` or `operator@(x,y)`. If both forms of the operator function have been declared, the rules in 13.3.1.2 determines which, if any, interpretation is used.

13.5.3 Assignment [over.ass]

1 An assignment operator shall be implemented by a non-static member function with exactly one parameter. Because a copy assignment operator `operator=` is implicitly declared for a class if not declared by the user (12.8), a base class assignment operator is always hidden by the copy assignment operator of the derived class.

2 Any assignment operator, even the copy assignment operator, can be virtual. [*Note:* for a derived class `D` with a base class `B` for which a virtual copy assignment has been declared, the copy assignment operator in `D` does not override `B`'s virtual copy assignment operator. [*Example:*

```
struct B {
        virtual int operator= (int);
        virtual B& operator= (const B&);
};
struct D : B {
        virtual int operator= (int);
        virtual D& operator= (const B&);
};
```

```
D dobj1;
D dobj2;
B* bptr = &dobj1;
void f() {
        bptr->operator=(99);    // calls D::operator(int)
        *bptr = 99;             // ditto
        bptr->operator=(dobj2); // calls D::operator(const B&)
        *bptr = dobj2;          // ditto
        dobj1 = dobj2;          // calls D::operator(const D&)
}
```

—*end example*] —*end note*]

13.5.4 Function call [over.call]

1 `operator()` shall be a non-static member function with an arbitrary number of parameters. It can have default arguments. It implements the function call syntax

postfix-expression (*expression-list*$_{opt}$)

where the *postfix-expression* evaluates to a class object and the possibly empty *expression-list* matches the parameter list of an `operator()` member function of the class. Thus, a call `x(arg1,...)` is interpreted as `x.operator()(arg1,...)` for a class object `x` of type `T` if `T::operator()(T1, T2, T3)` exists and if the operator is selected as the best match function by the overload resolution mechanism (13.3.3).

13.5.5 Subscripting [over.sub]

1 `operator[]` shall be a non-static member function with exactly one parameter. It implements the subscripting syntax

postfix-expression [*expression*]

Thus, a subscripting expression `x[y]` is interpreted as `x.operator[](y)` for a class object `x` of type `T` if `T::operator[](T1)` exists and if the operator is selected as the best match function by the overload resolution mechanism (13.3.3).

13.5.6 Class member access [over.ref]

1 `operator->` shall be a non-static member function taking no parameters. It implements class member access using `->`

postfix-expression `->` *primary-expression*

An expression `x->m` is interpreted as `(x.operator->())->m` for a class object `x` of type `T` if `T::operator->()` exists and if the operator is selected as the best match function by the overload resolution mechanism (13.3). `operator->` shall return either a pointer to a class or an object of or a reference to a class for which `operator->` is defined, except in some cases when it is a member of a template (see 14.3.3). `T::operator->` shall not return an object of or reference to its own class type `T`.

13.5.7 Increment and decrement [over.inc]

1 The prefix and postfix increment operators shall be implemented by a function called `operator++`. If this function is a member function with no parameters, or a non-member function with one class or enumeration parameter, it defines the prefix increment operator `++` for objects of that type. If the function is a member function with one parameter (which shall be of type `int`) or a non-member function with two parameters (the second shall be of type `int`), it defines the postfix increment operator `++` for objects of that type. When the postfix increment is called, the `int` argument will have value zero. [*Example:*

```
        class X {
        public:
            const X&   operator++();     // prefix ++a
            const X&   operator++(int);  // postfix a++
        };

        class Y {
        public:
        };
        const Y&   operator++(Y&);       // prefix ++b
        const Y&   operator++(Y&, int);  // postfix b++

        void f(X a, Y b)
        {
            ++a;        // a.operator++();
            a++;        // a.operator++(0);
            ++b;        // operator++(b);
            b++;        // operator++(b, 0);

            a.operator++();    // explicit call: like ++a;
            a.operator++(0);   // explicit call: like a++;
            operator++(b);     // explicit call: like ++b;
            operator++(b, 0);  // explicit call: like b++;
        }
```

—*end example*]

2 The prefix and postfix decrement operators `--` are handled similarly.

13.6 Built-in operators [over.built]

1 The candidate operator functions that represent the built-in operators defined in 5 are specified in this section. These candidate functions participate in the operator overload resolution process as described in 13.3.1.2 and are used for no other purpose.

2 [*Note:* since built-in operators take only operands with non-class type, and operator overload resolution occurs only when an operand expression originally has class or enumeration type, operator overload resolution can resolve to a built-in operator only when an operand has a class type that has a user-defined conversion to a non-class type appropriate for the operator, or when an operand has an enumeration type that can be converted to a type appropriate for the operator.]

3 In this section, the term *promoted integral type* is used to refer to those integral types which are preserved by integral promotion (including e.g. `int` and `long` but excluding e.g. `char`). Similarly, the term *promoted arithmetic type* refers to promoted integral types plus floating types.

4 For every pair (*T*, *VQ*), where *T* is an arithmetic type, and *VQ* is either `volatile` or empty, there exist candidate operator functions of the form

```
        VQ T&    operator++(VQ T&);
        VQ T&    operator--(VQ T&);
        T        operator++(VQ T&, int);
        T        operator--(VQ T&, int);
```

5 For every pair (*T*, *VQ*), where *T* is a cv-qualified or cv-unqualified complete object type, and *VQ* is either `volatile` or empty, there exist candidate operator functions of the form

```
        T*VQ&    operator++(T*VQ&);
        T*VQ&    operator--(T*VQ&);
        T*       operator++(T*VQ&, int);
        T*       operator--(T*VQ&, int);
```

6 For every cv-qualified or cv-unqualified complete object type *T*, there exist candidate operator functions of the form

```
T&      operator*(T*);
```

7 For every function type *T*, there exist candidate operator functions of the form

```
T&      operator*(T*);
```

8 For every type *T*, there exist candidate operator functions of the form

```
T*      operator+(T*);
```

9 For every promoted arithmetic type *T*, there exist candidate operator functions of the form

```
T       operator+(T);
T       operator-(T);
```

10 For every promoted integral type *T*, there exist candidate operator functions of the form

```
T       operator~(T);
```

11 For every quadruple (*C*, *T*, *CV1*, *CV2*), where *C* is a class type, *T* is a complete object type or a function type, and *CV1* and *CV2* are *cv-qualifier-seq*s, there exist candidate operator functions of the form

```
CV12 T&  operator->*(CV1 C*, CV2 T C::*);
```

where *CV12* is the union of *CV1* and *CV2*.

12 For every pair of promoted arithmetic types *L* and *R*, there exist candidate operator functions of the form

```
LR      operator*(L, R);
LR      operator/(L, R);
LR      operator+(L, R);
LR      operator-(L, R);
bool    operator<(L, R);
bool    operator>(L, R);
bool    operator<=(L, R);
bool    operator>=(L, R);
bool    operator==(L, R);
bool    operator!=(L, R);
```

where *LR* is the result of the usual arithmetic conversions between types *L* and *R*.

13 For every pair of types *T* and *I*, where *T* is a cv-qualified or cv-unqualified complete object type and *I* is a promoted integral type, there exist candidate operator functions of the form

```
T*      operator+(T*, I);
T&      operator[](T*, I);
T*      operator-(T*, I);
T*      operator+(I, T*);
T&      operator[](I, T*);
```

14 For every triple (*T*, *CV1*, *CV2*), where *T* is a complete object type, and *CV1* and *CV2* are *cv-qualifier-seq*s, there exist candidate operator functions of the form[93)]

```
ptrdiff_t operator-(CV1 T*, CV2 T*);
```

93) When T is itself a pointer type, the interior *cv*-qualifiers of the two parameter types need not be identical. The two pointer types are converted to a common type (which need not be the same as either parameter type) by implicit pointer conversions.

15 For every triple (*T*, *CV1*, *CV2*), where *T* is any type, and *CV1* and *CV2* are *cv-qualifier-seq*s, there exist candidate operator functions of the form[94)]

```
bool    operator<(CV1 T*, CV2 T*);
bool    operator>(CV1 T*, CV2 T*);
bool    operator<=(CV1 T*, CV2 T*);
bool    operator>=(CV1 T*, CV2 T*);
bool    operator==(CV1 T*, CV2 T*);
bool    operator!=(CV1 T*, CV2 T*);
```

16 For every quadruple (*C*, *T*, *CV1*, *CV2*), where *C* is a class type, *T* is any type, and *CV1* and *CV2* are *cv-qualifier-seq*s, there exist candidate operator functions of the form[95)]

```
bool    operator==(CV1 T C::*, CV2 T C::*);
bool    operator!=(CV1 T C::*, CV2 T C::*);
```

17 For every pair of promoted integral types *L* and *R*, there exist candidate operator functions of the form

```
LR      operator%(L, R);
LR      operator&(L, R);
LR      operator^(L, R);
LR      operator|(L, R);
L       operator<<(L, R);
L       operator>>(L, R);
```

where *LR* is the result of the usual arithmetic conversions between types *L* and *R*.

18 For every triple (*L*, *VQ*, *R*), where *L* is an arithmetic type, *VQ* is either `volatile` or empty, and *R* is a promoted arithmetic type, there exist candidate operator functions of the form

```
VQ L&   operator=(VQ L&, R);
VQ L&   operator*=(VQ L&, R);
VQ L&   operator/=(VQ L&, R);
VQ L&   operator+=(VQ L&, R);
VQ L&   operator-=(VQ L&, R);
```

19 For every pair (*T*, *VQ*), where *T* is any type and *VQ* is either `volatile` or empty, there exist candidate operator functions of the form

```
T*VQ&   operator=(T*VQ&, T*);
```

20 For every triple (*T*, *VQ*, *I*), where *T* is a cv-qualified or cv-unqualified complete object type, *VQ* is either `volatile` or empty, and *I* is a promoted integral type, there exist candidate operator functions of the form

```
T*VQ&   operator+=(T*VQ&, I);
T*VQ&   operator-=(T*VQ&, I);
```

21 For every triple (*L*, *VQ*, *R*), where *L* is an integral type, *VQ* is either `volatile` or empty, and *R* is a promoted integral type, there exist candidate operator functions of the form

```
VQ L&   operator%=(VQ L&, R);
VQ L&   operator<<=(VQ L&, R);
VQ L&   operator>>=(VQ L&, R);
VQ L&   operator&=(VQ L&, R);
VQ L&   operator^=(VQ L&, R);
VQ L&   operator|=(VQ L&, R);
```

[94)] When `T` is itself a pointer type, the interior *cv*-qualifiers of the two parameter types need not be identical. The two pointer types are converted to a common type (which need not be the same as either parameter type) by implicit pointer conversions.

[95)] When `T` is itself a pointer type, the interior *cv*-qualifiers of the two parameter types need not be identical. The two pointer types are converted to a common type (which need not be the same as either parameter type) by implicit pointer conversions.

22 There also exist candidate operator functions of the form

```
        bool    operator!(bool);
        bool    operator&&(bool, bool);
        bool    operator||(bool, bool);
```

14 Templates [temp]

1 A class *template* defines the layout and operations for an unbounded set of related types. [*Example:* a single class template `List` might provide a common definition for list of `int`, list of `float`, and list of pointers to `Shape`s.] A function *template* defines an unbounded set of related functions. [*Example:* a single function template `sort()` might provide a common definition for sorting all the types defined by the `List` class template.]

2 A *template* defines a family of types or functions.

template-declaration:
  `template <` *template-parameter-list* `>` *declaration*

template-parameter-list:
  template-parameter
  template-parameter-list `,` *template-parameter*

The *declaration* in a *template-declaration* shall declare or define a function or a class, define a static data member of a template class, or define a template member of a class. A *template-declaration* is a *declaration*. A *template-declaration* is a definition (also) if its *declaration* defines a function, a class, or a static data member of a template class. There shall be exactly one definition for each template in a program. [*Note:* there can be many declarations.] However, if the multiple definitions are in different translation units, the behavior is undefined (and no diagnostic is required).

3 The name of a template obeys the usual scope and access control rules. A *template-declaration* can appear only as a global declaration, as a member of a namespace, as a member of a class, or as a member of a class template. A member template shall not be `virtual`. A destructor shall not be a template. A local class shall not have a member template.

4 A template shall not have C linkage. If the linkage of a template is something other than C or C++, the behavior is implementation-defined.

5 [*Example:* An array class template might be declared like this:

```
template<class T> class Array {
    T* v;
    int sz;
public:
    explicit Array(int);
    T& operator[](int);
    T& elem(int i) { return v[i]; }
    // ...
};
```

The prefix `template <class T>` specifies that a template is being declared and that a *type-name* `T` will be used in the declaration. In other words, `Array` is a parameterized type with `T` as its parameter.]

6 [*Note:* a class template definition specifies how individual classes can be constructed much as a class definition specifies how individual objects can be constructed.]

7 A member template can be defined within its class or separately. [*Example:*

```
template<class T> class string {
public:
        template<class T2> int compare(const T2&);
        template<class T2> string(const string<T2>& s) { /* ... */ }
        // ...
};

template<class T> template<class T2> int string<T>::compare(const T2& s)
{
        // ...
}
```

—*end example*]

14.1 Template names [temp.names]

1 A template can be referred to by a *template-id*:

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

2 A *template-id* that names a template class is a *class-name* (9).

3 A *template-id* that names a defined template class can be used exactly like the names of other defined classes. [*Example:*

```
Array<int> v(10);
Array<int>* p = &v;
```

—*end example*] [*Note: template-id*s that name functions are discussed in 14.10.]

4 A *template-id* that names a template class that has been declared but not defined can be used exactly like the names of other declared but undefined classes. [*Example:*

```
template<class T> class X; // X is a class template

X<int>* p; // ok: pointer to declared class X<int>
X<int> x;  // error: object of undefined class X<int>
```

—*end example*]

5 The name of a template followed by a `<` is always taken as the beginning of a *template-id* and never as a name followed by the less-than operator. Similarly, the first non-nested `>` is taken as the end of the *template-argument-list* rather than a greater-than operator. [*Example:*

```
template<int i> class X { /* ... */ }

X< 1>2 >x1; // syntax error
X<(1>2)>x2; // ok

template<class T> class Y { /* ... */ }
Y< X<1> > x3; // ok
```

—*end example*]

6 The name of a class template shall not be declared to refer to any other template, class, function, object, enumeration, enumerator, namespace, value, or type in the same scope. Unless explicitly specified to have internal linkage, a template in namespace scope has external linkage (3.5). A global template name shall be unique in a program.

7 In a *template-argument*, an ambiguity between a *type-id* and an *expression* is resolved to a *type-id*. [*Example:*

```
template<class T> void f();
template<int I> void f();

void g()
{
        f<int()>(); // ‘‘int()’’ is a type-id: call the first f()
}
```

—*end example*]

14.2 Name resolution [temp.res]

1 A name used in a template is assumed not to name a type unless it has been explicitly declared to refer to a type in the context enclosing the template declaration or is qualified by the keyword `typename`. [*Example:*

```
// no B declared here

class X;

template<class T> class Y {
        class Z; // forward declaration of member class

        void f() {
                X* a1;    // declare pointer to X
                T* a2;    // declare pointer to T
                Y* a3;    // declare pointer to Y
                Z* a4;    // declare pointer to Z
                typedef typename T::A TA;
                TA* a5;   // declare pointer to T’s A
                typename T::A* a6;   // declare pointer to T’s A
                T::A* a7; // T::A is not a type name:
                          // multiply T::A by a7
                B* a8;    // B is not a type name:
                          // multiply B by a8
        }
};
```

—*end example*]

2 In a template, any use of a *qualified-name* where the qualifier depends on a *template-parameter* can be prefixed by the keyword `typename` to indicate that the *qualified-name* denotes a type.

elaborated-type-specifier:
...
`typename` `::`$_{opt}$ *nested-name-specifier identifier full-template-argument-list*$_{opt}$

full-template-argument-list:
`<` *template-argument-list* `>`

3 If a specialization of that template is generated for a *template-argument* such that the *qualified-name* does not denote a type, the specialization is ill-formed. The keyword `typename` states that the following *qualified-name* names a type. [*Note:* but gives no clue to what that type might be.] The *qualified-name* shall include a qualifier containing a template parameter or a template class name.

4 Knowing which names are type names allows the syntax of every template declaration to be checked. Syntax errors in a template declaration can therefore be diagnosed at the point of the declaration exactly as errors for non-template constructs. Other errors, such as type errors involving template parameters, cannot be diagnosed until later; such errors shall be diagnosed at the point of instantiation or at the point where member functions are generated (14.3). Errors that can be diagnosed at the point of a template declaration shall be diagnosed there or later together with the dependent type errors. [*Example:*

```
template<class T> class X {
        // ...
        void f(T t, int i, char* p)
        {
                t = i;  // typecheck at point of instantiation,
                        //        or at function generation
                p = i;  // typecheck immediately at template declaration,
                        //        at point of instantiation,
                        //        or at function generation
        }
};
```

—*end example*] No diagnostics shall be issued for a template definition for which a valid specialization can be generated.

5 Three kinds of names can be used within a template definition:

— The name of the template itself, the names of the *template-parameter*s (14.7), and names declared within the template itself.

— Names from the scope of the template definition.

— Names dependent on a *template-argument* (14.8) from the scope of a template instantiation.

6 [*Example:*

```
#include <iostream>
using namespace std;

template<class T> class Set {
        T* p;
        int cnt;
public:
        Set();
        Set<T>(const Set<T>&);
        void printall()
        {
                for (int i = 0; i<cnt; i++)
                        cout << p[i] << '\n';
        }
        // ...
};
```

—*end example*] When looking for the declaration of a name used in a template definition the usual lookup rules (9.3) are first applied. [*Note:* in the example, `i` is the local variable `i` declared in `printall`, `cnt` is the member `cnt` declared in `Set`, and `cout` is the standard output stream declared in `iostream`. However, not every declaration can be found this way; the resolution of some names must be postponed until the actual *template-argument* is known. For example, even though the name `operator<<` is known within the definition of `sum()` an a declaration of it can be found in `<iostream>`, the actual declaration of `operator<<` needed to print `p[i]` cannot be known until it is known what type `T` is (14.2.3).]

7 If a name can be bound at the point of the template definition and it is not a function called in a way that depends on a *template-parameter* (as defined in 14.2.3), it will be bound at the template definition point and the binding is not affected by later declarations. [*Example:*

```
void f(char);

template<class T> void g(T t)
{
        f(1);     // f(char)
        f(T(1));  // dependent
        f(t);     // dependent
}

void f(int);

void h()
{
        g(2);   // will cause one call of f(char) followed
                //  by two calls of f(int)
        g('a'); // will cause three calls of f(char)
}
```

—*end example*]

14.2.1 Locally declared names [temp.local]

1 Within the scope of a class template or a specialization of a template the name of the template is equivalent to the name of the template followed by the *template-parameter*s enclosed in `<>`. [*Example:* the constructor for `Set` can be referred to as `Set()` or `Set<T>()`.] Other specializations (14.5) of the class can be referred to by explicitly qualifying the template name with appropriate *template-argument*s. [*Example:*

```
template<class T> class X {
        X* p;           // meaning X<T>
        X<T>* p2;
        X<int>* p3;
};

template<class T> class Y;

class Y<int> {
        Y* p;           // meaning Y<int>
};
```

—*end example*] [*Note:* see 14.7 for the scope of *template-parameter*s.]

2 A template *type-parameter* can be used in an *elaborated-type-specifier*. [*Example:*

```
template<class T> class A {
        friend class T;
        class T* p;
        class T;        // error: redeclaration of template parameter T
                        // (a name declaration, not an elaboration)
        // ...
}
```

—*end example*]

3 However, a specialization of a template for which a *type-parameter* used this way is not in agreement with the *elaborated-type-specifier* (7.1.5) is ill-formed. [*Example:*

```
class C { /* ... */ };
struct S { /* ... */ };
union U { /* ... */ };
enum E { /* ... */ };

A<C> ac;        // ok
A<S> as;        // ok
A<U> au;        // error: parameter T elaborated as a class,
                // but the argument supplied for T is a union
A<int> ai;      // error: parameter T elaborated as a class,
                // but the argument supplied for T is an int
A<E> ae;        // error: parameter T elaborated as a class,
                // but the argument supplied for T is an enumeration
```

—*end example*]

14.2.2 Names from the template's enclosing scope [temp.encl]

1 If a name used in a template isn't defined in the template definition itself, names declared in the scope enclosing the template are considered. If the name used is found there, the name used refers to the name in the enclosing context. [*Example:*

```
void g(double);
void h();

template<class T> class Z {
public:
        void f() {
                g(1); // calls g(double)
                h++;  // error: cannot increment function
        }
};

void g(int); // not in scope at the point of the template
             // definition, not considered for the call g(1)
```

—*end example*] [*Note:* a template definition behaves exactly like other definitions.] [*Example:*

```
void g(double);
void h();

class ZZ {
public:
        void f() {
                g(1); // calls g(double)
                h++;  // error: cannot increment function
        }
};

void g(int); // not in scope at the point of class ZZ
             // definition, not considered for the call g(1)
```

—*end example*]

14.2.3 Dependent names [temp.dep]

1 Some names used in a template are neither known at the point of the template definition nor declared within the template definition. Such names shall depend on a *template-argument* and shall be in scope at the point of the template instantiation (14.3). [*Example:*

```
class Horse { /* ... */ };

ostream& operator<<(ostream&,const Horse&);

void hh(Set<Horse>& h)
{
        h.printall();
}
```

In the call of `Set<Horse>::printall()`, the meaning of the `<<` operator used to print `p[i]` in the definition of `Set<T>::printall()` (14.2), is

```
operator<<(ostream&,const Horse&);
```

This function takes an argument of type `Horse` and is called from a template with a *template-parameter* `T` for which the *template-argument* is `Horse`. Because this function depends on a *template-argument* the call is well-formed.]

2 A function call *depends on* a *template-argument* if the call would have a different resolution or no resolution if a type, template, or named constant mentioned in the *template-argument* were missing from the program. [*Example:* some calls that depend on an argument type `T` are:

1) The function called has a parameter that depends on `T` according to the type deduction rules (14.10.2). For example: `f(T)`, `f(Array<T>)`, and `f(const T*)`.

2) The type of the actual argument depends on `T`. For example: `f(T(1))`, `f(t)`, `f(g(t))`, and `f(&t)` assuming that `t` has the type `T`.

3) A call is resolved by the use of a conversion to `T` without either an argument or a parameter of the called function being of a type that depended on `T` as specified in (1) and (2). For example:

```
struct B { };
struct T : B { };
struct X { operator T(); };

void f(B);

void g(X x)
{
        f(x);  // meaning f( B( x.operator T() ) )
               // so the call f(x) depends on T
}
```

3 This ill-formed template instantiation uses a function that does not depend on a *template-argument*:

```
template<class T> class Z {
public:
        void f() {
                g(1); // g() not found in Z's context.
                      // Look again at point of instantiation
        }
};

void g(int);

void h(const Z<Horse>& x)
{
        x.f(); // error: g(int) called by g(1) does not depend
               // on template-parameter ''Horse''
}
```

The call `x.f()` gives raise to the specialization:

```
Z<Horse>::f() { g(1); }
```

The call `g(1)` would call `g(int)`, but since that call in no way depends on the *template-argument* `Horse` and because `g(int)` wasn't in scope at the point of the definition of the template, the call `x.f()` is ill-formed.

4 On the other hand:

```
void h(const Z<int>& y)
{
        y.f(); // fine: g(int) called by g(1) depends
               // on template-parameter ‘‘int’’
}
```

Here, the call `y.f()` gives raise to the specialization:

```
Z<int>::f() { g(1); }
```

The call `g(1)` calls `g(int)`, and since that call depends on the *template-argument* `int`, the call `y.f()` is acceptable even though `g(int)` wasn't in scope at the point of the template definition.]

5 A name from a base class (of a non-dependent type) can hide the name of a *template-parameter*. [*Example:*

```
struct A {
        struct B { /* ... */ };
        int a;
        int Y;
};

template<class B, class a> struct X : A {
        B b;  // A’s B
        a b;  // error: A’s a isn’t a type name
};
```

—*end example*]

6 However, a name from a *template-argument* cannot hide a name declared within a template, a *template-parameter*, or a name from the template's enclosing scopes. [*Example:*

```
int a;

template<class T> struct Y : T {
        struct B { /* ... */ };
        B b;                       // The B defined in Y
        void f(int i) { a = i; }   // the global a;
        Y* p;                      // Y<T>
};

Y<A> ya;
```

The members `A::B`, `A::a`, and `A::Y` of the template argument `A` do not affect the binding of names in `Y<A>`.]

7 A name of a member can hide the name of a *template-parameter*. [*Example:*

```
template<class T> struct A {
        struct B { /* ... */ };
        void f();
};

template<class B> void A<B>::f()
{
        B b;  // A’s B, not the template parameter
}
```

—*end example*]

14.2.4 Non-local names declared within a template [temp.inject]

1 Names that are not template members can be declared within a template class or function. When a template is specialized, the names declared in it are declared as if the specialization had been explicitly declared at its point of instantiation. If a template is first specialized as the result of use within a block or class, names declared within the template shall be used only after the template use that caused the specialization. [*Example:*

```
// Assume that Y is not yet declared

template<class T> class X {
        friend class Y;
};

Y* py1;             // ill-formed: Y is not in scope

// Here is the point of instantiation for X<C>
void g()
{
        X<C>* pc;   // does not cause instantiation
        Y* py2;     // ill-formed: Y is not in scope
        X<C> c;     // causes instantiation of X<C>, so
                    // names from X<C> can be used
                    // here on
        Y* py3;     // ok
}
Y* py4;             // ok
```

—*end example*]

14.3 Template instantiation [temp.inst]

1 A class generated from a class template is called a generated class. A function generated from a function template is called a generated function. A static data member generated from a static data member template is called a generated static data member. A class defined with a *template-id* as its name is called an explicitly specialized class. A function defined with a *template-id* as its name is called an explicitly specialized function. A static data member defined with a *template-id* as its name is called an explicitly specialized static data member. A specialization is a class, function, or static data member that is either generated or explicitly specialized.

2 [*Note:* the act of generating a class, function, or static data member from a template is commonly referred to as template instantiation.]

14.3.1 Template linkage [temp.linkage]

1 A function template has external linkage, as does a static member of a class template. Every function template shall have the same definition in every translation unit in which it appears.

14.3.2 Point of instantiation [temp.point]

1 The point of instantiation of a template is the point where names dependent on the *template-argument* are bound. That point is immediately before the declaration in the nearest enclosing global or namespace scope containing the first use of the template requiring its definition. [*Note:* this implies that names used in a template definition cannot be bound to local names or class member names from the scope of the template use. They can, however, be bound to names of namespace members. For example:

```
// void g(int); not declared here

template<class T> class Y {
public:
        void f() { g(1); }
};

void k(const Y<int>& h)
{
        void g(int);
        h.f(); // error: g(int) called by g(1) not found
               //        local g() not considered
}

class C {
        void g(int);

        void m(const Y<int>& h)
        {
                h.f(); // error: g(int) called by g(1) not found
                       //        C::g() not considered
        }
};

namespace N {
        void g(int);

        void n(const Y<int>& h)
        {
                h.f(); // N::g(int) called by g(1)
        }
}
```

—*end note*]

2 Names from both the namespace of the template itself and of the namespace containing the point of instantiation of a specialization are used to resolve names for the specialization. Overload resolution is used to chose between functions with the same name in these two namespaces. [*Example:*

```
namespace NN {
        void g(int);
        void h(int);
        template<class T> void f(T t)
        {
                g(t);
                h(t);
                k(t);
        }
}
```

```
namespace MM {
        void g(double);
        void k(double);

        // instantiation point for NN::f(int) and NN::f(double)

        void m()
        {
                NN::f(1);    // indirectly calls NN::g(int),
                             //                  NN::h, and MM::k.
                NN::f(1.0);  // indirectly calls MM::g(double),
                             //                  NN::h, and MM::k.
        }
}
```

—*end example*] If a name is found in both namespaces and overload resolution cannot resolve a use, the program is ill-formed.

3 Each translation unit in which the definition of a template is used in a way that require definition of a specialization has a point of instantiation for the template. If this causes names used in the template definition to bind to different names in different translation units, the one-definition rule has been violated and any use of the template is ill-formed. Such violation does not require a diagnostic.

4 A template can be either explicitly instantiated for a given argument list or be implicitly instantiated. A template that has been used in a way that require a specialization of its definition will have the specialization implicitly generated unless it has either been explicitly instantiated (14.4) or explicitly specialized (14.5). A specialization will not be implicitly generated unless the definition of a template specialization is required. [*Example:*

```
template<class T> class Z {
        void f();
        void g();
};

void h()
{
        Z<int> a;     // instantiation of class Z<int> required
        Z<char>* p;   // instantiation of class Z<char> not required
        Z<double>* q; // instantiation of class Z<double> not required

        a.f();  // instantiation of Z<int>::f() required
        p->g(); // instantiation of class Z<char> required, and
                // instantiation of Z<char>::g() required
}
```

Nothing in this example requires `class Z<double>`, `Z<int>::g()`, or `Z<char>::f()` to be instantiated.] An implementation shall not instantiate a function or a class that does not require instantiation. However, virtual functions can be instantiated for implementation purposes.

5 If a virtual function is instantiated, its point of instantiation is immediately following the point of instantiation for its class.

6 The point of instantiation for a template used inside another template and not instantiated previous to an instantiation of the enclosing template is immediately before the point of instantiation of the enclosing template. [*Example:*

```
namespace N {
        template<class T> class List {
        public:
                T* get();
                // ...
        };
}

template<class K, class V> class Map {
        List<V> lt;
        V get(K);
        //  ...
};

void g(Map<char*,int>& m)
{
        int i = m.get("Nicholas");
        // ...
}
```

—*end example*] This allows instantiation of a used template to be done before instantiation of its user.

7 Implicitly generated template classes, functions, and static data members are placed in the namespace where the template was defined. [*Example:* a call of `lt.get()` from `Map<char*,int>::get()` would place `List<int>::get()` in the namespace `N` rather than in the global namespace.]

8 If a template for which a definition is in scope is used in a way that involves overload resolution or conversion to a base class, the definition of a template specialization is required. [*Example:*

```
template<class T> class B { /* ... */ };
template<class T> class D : public B<T> { /* ... */ };

void f(void*);
void f(B<int>*);

void g(D<int>* p, D<char>* pp)
{
        f(p); // instantiation of D<int> required: call f(B<int>*)

        B<char>* q = pp; // instantiation of D<char> required:
                         // convert D<char>* to B<char>*
}
```

—*end example*]

9 If an instantiation of a class template is required and the template is declared but not defined, the program is ill-formed. [*Example:*

```
template<class T> class X;

X<char> ch; // error: definition of X required
```

—*end example*]

10 Recursive instantiation is possible. [*Example:*

```
template<int i> int fac() { return i>1 ? i*fac<i-1>() : 1; }

int fac<0>() { return 1; }

int f()
{
        return fac<17>();
}
```

—*end example*]

11 There shall be an implementation quantity that specifies the limit on the depth of recursive instantiations.

12 The result of an infinite recursion in instantiation is undefined. In particular, an implementation is allowed to report an infinite recursion as being ill-formed. [*Example:*

```
template<class T> class X {
        X<T>* p; // ok
        X<T*> a; // instantiation of X<T> requires
                 // the instantiation of X<T*> which requires
                 // the instantiation of X<T**> which ...
};
```

—*end example*]

13 No program shall explicitly instantiate any template more than once, both explicitly instantiate and explicitly specialize a template, or specialize a template more than once for a given set of *template-argument*s. An implementation is not required to diagnose a violation of this rule.

14 An explicit specialization or explicit instantiation of a template shall be in the namespace in which the template was defined. [*Example:*

```
namespace N {
        template<class T> class X { /* ... */ };
        template<class T> class Y { /* ... */ };
        template<class T> class Z {
                void f(int i) { g(i); }
                // ...
        };

        class X<int> { /* ... */ }; // ok: specialization
                                    //     in same namespace
}

template class Y<int>; // error: explicit instantiation
                       //        in different namespace

template class N::Y<char*>; // ok: explicit instantiation
                            //     in same namespace

class N::Y<double> { /* ... */ }; // ok: specialization
                                  //     in same namespace
```

—*end example*]

15 A member function of an explicitly specialized class shall not be implicitly generated from the general template. Instead, the member function shall itself be explicitly specialized. [*Example:*

```
template<class T> struct A {
        void f() { /* ... */ }
};

struct A<int> {
        void f();
};

void h()
{
        A<int> a;
        a.f();  // A<int>::f must be defined somewhere
}

void A<int>::f() { /* ... */ };
```

—*end example*] Thus, an explicit specialization of a class implies the declaration of specializations of all of its members. The definition of each such specialized member which is used shall be provided in some translation unit.

14.3.3 Instantiation of `operator->` [temp.opref]

1 If a template class has an `operator->`, that `operator->` can have a return type that cannot be dereferenced by `->` as long as that `operator->` is neither invoked, nor has its address taken, isn't virtual, nor is explicitly instantiated. [*Example:*

```
template<class T> class Ptr {
        // ...
        T* operator->();
};

Ptr<int> pi; // ok
Ptr<Rec> pr; // ok

void f()
{
        pi->m = 7; // error: Ptr<int>::operator->() returns a type
                   //        that cannot be dereference by ->
        pr->m = 7; // ok if Rec has an accessible member m
                   // of suitable type
}
```

—*end example*]

14.4 Explicit instantiation [temp.explicit]

1 A class or function specialization can be explicitly instantiated from its template.

2 The syntax for explicit instantiation is:

explicit-instantiation:
`template` *declaration*

Where the *unqualifier-id* in the *declaration* shall be a *template-id*. [*Example:*

```
template class Array<char>;

template void sort<char>(Array<char>&);
```

—*end example*]

3 A declaration of the template shall be in scope at the point of explicit instantiation.

4 A trailing *template-argument* can be left unspecified in an explicit instantiation or explicit specialization of a template function provided it can be deduced from the function argument type. [*Example:*

```
// instantiate sort(Array<int>&):
// deduce template-argument:
template void sort<>(Array<int>&);
```

—*end example*]

5 The explicit instantiation of a class implies the instantiation of all of its members not previously explicitly specialized in the translation unit containing the explicit instantiation.

14.5 Template specialization [temp.spec]

1 Except for a type member or template class member of a non-specialized template class, the following can be declared by a declaration where the declared name is a *template-id*: a specialized template function, a template class, or a static member of a template; that is:

specialization:
  declaration

[*Note:* a static member of a template can only be specialized in a definition due to syntactic restrictions.] [*Example:*

```
template<class T> class stream;

class stream<char> { /* ... */ };

template<class T> void sort(Array<T>& v) { /* ... */ }

void sort<char*>(Array<char*>&) ;
```

Given these declarations, `stream<char>` will be used as the definition of streams of `char`s; other streams will be handled by template classes generated from the class template. Similarly, `sort<char*>` will be used as the sort function for arguments of type `Array<char*>`; other `Array` types will be sorted by functions generated from the template.]

2 A declaration of the template being specialized shall be in scope at the point of declaration of a specialization. [*Example:*

```
class X<int> { /* ... */ }; // error: X not a template

template<class T> class X { /* ... */ };

class X<char*> { /* ... */ }; // fine: X is a template
```

—*end example*]

3 If a template is explicitly specialized then that specialization shall be declared before the first use of that specialization in every translation unit in which it is used. [*Example:*

```
template<class T> void sort(Array<T>& v) { /* ... */ }

void f(Array<String>& v)
{
        sort(v); // use general template
                 // sort(Array<T>&), T is String
}

void sort<String>(Array<String>& v); // error: specialize after use
void sort<>(Array<char*>& v); // fine sort<char*> not yet used
```

—*end example*] If a function or class template has been explicitly specialized for a *template-argument* list no specialization will be implicitly generated for that *template-argument* list.

4 It is possible for a specialization with a given function signature to be generated by more than one function template. In such cases, explicit specification of the template arguments must be used to uniquely identify the template function instance that is being specialized. [*Example:*

```
template <class T> void f(T);
template <class T> void f(T*);
void f<>(int*);        // Ambiguous
void f<int>(int*);     // OK
void f<>(int);         // OK
```

—*end example*]

5 Note that a function with the same name as a template and a type that exactly matches that of a template is not a specialization (14.10.5).

14.6 Class template specializations [temp.class.spec]

1 A primary class template declaration is one in which the class template name is an identifier. A template declaration in which the class template name is a *template-id*, is a partial specialization of the class template named in the *template-id*. The primary template shall be declared before any specializations of that template.

2 [*Example:*

3

```
template<class T1, class T2, int I> class A             { }; // #1
template<class T, int I>            class A<T, T*, I>   { }; // #2
template<class T1, class T2, int I> class A<T1*, T2, I> { }; // #3
template<class T>                   class A<int, T*, 5> { }; // #4
template<class T1, class T2, int I> class A<T1, T2*, I> { }; // #5
```

4 The first declaration declares the primary (unspecialized) class template. The second and subsequent declarations declare specializations of the primary template.]

5 The template parameters are specified in the angle bracket enclosed list that immediately follows the keyword `template`. A template also has a template argument list. For specializations, this list is explicitly written immediately following the class template name. For primary templates, this list is implicitly described by the template parameter list. Specifically, the order of the template parameters is the sequence in which they appear in the template parameter list. [*Example:* the template argument list for the primary template in the example above is `<T1, T2, I>`.]

6 A nontype argument is nonspecialized if it is the name of a nontype parameter. All other nontype arguments are specialized.

7 Within the argument list of a class template specialization, the following restrictions apply:

— A specialized nontype argument expression shall not involve a template parameter of the specialization.

— The type of a specialized nontype argument shall not depend on another type parameter of the specialization.

— The argument list of the specialization shall not be identical to the implicit argument list of the primary template.

8

14.6.1 Matching of class template specializations [temp.class.spec.match]

1 When a template class is used in a context that requires a complete instantiation of the class, it is necessary to determine whether the instantiation is to be generated using the primary template or one of the partial specializations. This is done by matching the template arguments of the template class being used with the template argument lists of the partial specializations.

— If no matches are found, the instantiation is generated from the primary template.

— If exactly one matching specialization is found, the instantiation is generated from that specialization.

— If more than one specialization is found, the partial order rules (14.6.2) are used to determine whether one of the specializations is more specialized than the others. If none of the specializations is more specialized than all of the other matching specializations, then the use of the template class is ambiguous and the program is ill-formed.

2 A specialization matches a given actual template argument list if the template arguments of the specialization can be deduced from the actual template argument list (14.10.2). A nontype template parameter can also be deduced from the value of an actual template argument of a nontype parameter of the primary template. [*Example:*

3

```
A<int, int, 1>   a1;  // uses #1
A<int, int*, 1>  a2;  // uses #2, T is int, I is 1
A<int, char*, 5> a3;  // uses #4, T is int
A<int, char*, 1> a4;  // uses #5, T1 is int, T2 is char, I is 1
A<int*, int*, 2> a5;  // ambiguous: matches #3 and #5
```

—*end example*]

4 In a class template reference, (e.g., `A<int, int, 1>`) the argument list must match the template parameter list of the primary template. The template arguments of a specialization are deduced from the arguments of the primary template. The template parameter list of a specialization shall not contain default template argument values.[96)]

14.6.2 Partial ordering of class template specializations [temp.class.order]

1 For two class template partial specializations, the first is at least as specialized as the second if:

— the type arguments of the first template's argument list are at least as specialized as those of the second template's argument list using the ordering rules for function templates (14.10.6), and

— each nontype argument of the first template's argument list is at least as specialized as that of the second template's argument list.

2 A nontype argument is at least as specialized as another nontype argument if:

— both are formal arguments,

— the first is a value and the second is a formal argument, or

— both are the same value.

3 A template class partial specialization is more specialized than another if, and only if, it is at least as specialized as the other template class partial specialization and that template class partial specialization is not at least as specialized as the first. Otherwise the two template class partial specializations are unordered.

14.7 Template parameters [temp.param]

1 The syntax for *template-parameter*s is:

template-parameter:
    type-parameter
    parameter-declaration

96) There is no way in which they could be used.

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*

[*Example:*

```
template<class T> class myarray { /* ... */ };

template<class K, class V, template<class T> class C = myarray>
class Map {
        C<K> key;
        C<V> value;
        // ...
};
```

—*end example*]

2 Default arguments shall not be specified in a declaration or a definition of a specialization.

3 A *type-parameter* defines its *identifier* to be a *type-name* in the scope of the template declaration. A *type-parameter* shall not be redeclared within its scope (including nested scopes). A non-type *template-parameter* shall not be assigned to or in any other way have its value changed. [*Example:*

```
template<class T, int i> class Y {
        int T;  // error: template-parameter redefined
        void f() {
                char T; // error: template-parameter redefined
                i++;    // error: change of template-argument value
        }
};

template<class X> class X; // error: template-parameter redefined
```

—*end example*]

4 A *template-parameter* that could be interpreted as either an *parameter-declaration* or a *type-parameter* (because its *identifier* is the name of an already existing class) is taken as a *type-parameter*. A *template-parameter* hides a variable, type, constant, etc. of the same name in the enclosing scope. [*Example:*

```
class T { /* ... */ };
int i;

template<class T, T i> void f(T t)
{
        T t1 = i;      // template-arguments T and i
        ::T t2 = ::i;  // globals T and i
}
```

Here, the template `f` has a *type-parameter* called `T,` rather than an unnamed non-type parameter of class `T`.] There is no semantic difference between `class` and `typename` in a *template-parameter*.

5 There are no restrictions on what can be a *template-argument* type beyond the constraints imposed by the set of argument types (14.8). In particular, reference types and types containing *cv-qualifiers* are allowed. A non-reference *template-argument* cannot have its address taken. When a non-reference *template-argument* is used as an initializer for a reference a temporary is always used. [*Example:*

```
template<const X& x, int i> void f()
{
        &x; // ok
        &i; // error: address of non-reference template-argument

        int& ri = i; // error: non-const reference bound to temporary
        const int& cri = i; // ok: reference bound to temporary
}
```

—*end example*]

6 A non-type *template-parameter* shall not be of floating type. [*Example:*

```
template<double d> class X;    // error
template<double* pd> class X;  // ok
template<double& rd> class X;  // ok
```

—*end example*]

7 A default *template-argument* is a type, value, or template specified after = in a *template-parameter*. A default *template-argument* can be specified in a template declaration or a template definition. The set of default *template-argument*s available for use with a template in a translation unit shall be provided by the first declaration of the template in that unit.

8 If a *template-parameter* has a default argument, all subsequent *template-parameter*s shall have a default argument supplied. [*Example:*

```
template<class T1 = int, class T2> class B; // error
```

—*end example*]

9 The scope of a *template-argument* extends from its point of declaration until the end of its template. In particular, a *template-parameter* can be used in the declaration of subsequent *template-parameter*s and their default arguments. [*Example:*

```
template<class T, T* p, class U = T> class X { /* ... */ };
template<class T> void f(T* p = new T);
```

—*end example*] A *template-parameter* cannot be used in preceding *template-parameters* or their default arguments.

10 A *template-parameter* can be used in the specification of base classes. [*Example:*

```
template<class T> class X : public Array<T> { /* ... */ };
template<class T> class Y : public T { /* ... */ };
```

—*end example*] [*Note:* the use of a *template-parameter* as a base class implies that a class used as a *template-argument* must be defined and not just declared.]

14.8 Template arguments [temp.arg]

1 The types of the *template-argument*s specified in a *template-id* shall match the types specified for the template in its *template-parameter-list*. [*Example:* `Array`s as defined in 14 can be used like this:

```
Array<int> v1(20);
typedef complex<double> dcomplex; // complex is a standard
                                  // library template
Array<dcomplex> v2(30);
Array<dcomplex> v3(40);

v1[3] = 7;
v2[3] = v3.elem(4) = dcomplex(7,8);
```

—*end example*]

2 A non-type non-reference *template-argument* shall be a *constant-expression* of non-floating type, the address of an object or a function with external linkage, or a non-overloaded pointer to member. The address of an object or function shall be expressed as `&f`, plain `f` (for function only), or `&X::f` where `f` is the function or object name. In the case of `&X::f`, `X` shall be a (possibly qualified) name of a class and `f` the name of a static member of `X`. A pointer to member shall be expressed as `&X::m` where `X` is a (possibly qualified) name of a class and `m` is the name of a nonstatic member of `X`. In particular, a string literal (2.9.4) is *not* an acceptable *template-argument* because a string literal is the address of an object with static linkage. [*Example:*

```
template<class T, char* p> class X {
        // ...
        X(const char* q) { /* ... */ }
};

X<int,"Studebaker"> x1; // error: string literal as template-argument

char* p = "Vivisectionist";
X<int,p> x2; // ok
```

—*end example*]

3 Similarly, addresses of array elements and non-static class members are not acceptable as `template-argument`s. [*Example:*

```
int a[10];
struct S { int m; static int s; } s;

X<&a[2],p> x3; // error: address of element
X<&s.m,p> x4;  // error: address of member
X<&s.s,p> x5;  // error: address of member (dot operator used)
X<&S::s,p> x6; // ok: address of static member
```

—*end example*]

4 Nor is a local type or a type with no linkage name an acceptable *template-argument*. [*Example:*

```
void f()
{
        struct S { /* ... */ };

        X<S,p> x3; // error: local type used as template-argument
}
```

—*end example*]

5 Similarly, a reference *template-parameter* shall not be bound to a temporary, an unnamed lvalue, or a named lvalue with no linkage. [*Example:*

```
template<const int& CRI> struct B { /* ... */ };

B<1> b2; // error: temporary required for template argument

int c = 1;
B<c> b1; // ok
```

—*end example*]

6 An argument to a *template-parameter* of pointer to function type shall have exactly the type specified by the *template* parameter. This allows selection from a set of overloaded functions. [*Example:*

```
void f(char);
void f(int);

template<void (*pf)(int)> struct A { /* ... */ };

A<&f> a; // selects f(int)
```

—*end example*]

7 If a *template-argument* to a template class is a function type and that causes a declaration that does not use the syntactic form of a function declarator to have function type, the program is ill-formed. [*Example:*

```
template<class T>
struct A {
        static T t;
};
typedef int function();
A<function> a;  // ill-formed: would declare A<function>::t
                // as a static member function
```

—*end example*]

8 A template has no special access rights to its *template-argument* types. A *template-argument* shall be accessible at the point where it is used as a *template-argument*. [*Example:*

```
template<class T> class X { /* ... */ };

class Y {
private:
        struct S { /* ... */ };
        X<S> x;  // ok: S is accessible
};

X<Y::S> y; // error: S not accessible
```

—*end example*]

9 When default *template-arguments* are used, a *template-argument* list can be empty. In that case the empty <> brackets shall still be used. [*Example:*

```
template<class T = char> class String;
String<>* p; // ok: String<char>
String* q;   // syntax error
```

—*end example*] The notion of “ array type decay” does not apply to *template-parameter*s. [*Example:*

```
template<int a[5]> struct S { /* ... */ };
int v[5];
int* p = v;
S<v> x; // fine
S<p> y; // error
```

—*end example*]

14.9 Type equivalence [temp.type]

1 Two *template-id*s refer to the same class or function if their *template* names are identical and in the same scope and their *template-argument*s have identical values. [*Example:*

```
template<class E, int size> class buffer;

buffer<char,2*512> x;
buffer<char,1024> y;
```

declares `x` and `y` to be of the same type, and

```
template<class T, void(*err_fct)()> class list { /* ... */ };

list<int,&error_handler1> x1;
list<int,&error_handler2> x2;
list<int,&error_handler2> x3;
list<char,&error_handler2> x4;
```

declares `x2` and `x3` to be of the same type. Their type differs from the types of `x1` and `x4`.]

14.10 Function templates [temp.fct]

1 A function template specifies how individual functions can be constructed. [*Example:* a family of sort functions, might be declared like this:

```
template<class T> void sort(Array<T>&);
```

—*end example*] A function template specifies an unbounded set of (overloaded) functions. A function generated from a function template is called a template function, so is an explicit specialization of a function template. Template arguments can either be explicitly specified in a call or be deduced from the function arguments.

14.10.1 Explicit template argument specification [temp.arg.explicit]

1 Template arguments can be specified in a call by qualifying the template function name by the list of *template-argument*s exactly as *template-argument*s are specified in uses of a class template. [*Example:*

```
void f(Array<dcomplex>& cv, Array<int>& ci)
{
    sort<dcomplex>(cv); // sort(Array<dcomplex>)
    sort<int>(ci);      // sort(Array<int>)
}
```

and

```
template<class U, class V> U convert(V v);

void g(double d)
{
        int i = convert<int,double>(d);  // int convert(double)
        char c = convert<char,double>(d); // char convert(double)
}
```

—*end example*] Implicit conversions (4) are accepted for a function argument for which the parameter has been fixed by explicit specification of *template-argument*s. [*Example:*

```
template<class T> void f(T);

class Complex {
        // ...
        explicit Complex(double);
};

void g()
{
        f<Complex>(1); // ok, means f<Complex>(Complex(1))
}
```

—*end example*]

2 For a template function name to be explicitly qualified by template arguments, the name must be known to refer to a template. When the name appears after `.` or `->` in a *postfix-expression*, or after `::` in a *qualified-id* where the *nested-name-specifier* depends on a template parameter, the member template name must be prefixed by the keyword `template`. Otherwise the name is assumed to name a non-template. [*Example:*

3

```
class X {
public:
        template<size_t> X* alloc();
};
void f(X* p)
{
        X* p1 = p->alloc<200>();
                // ill-formed: < means less than

        X* p2 = p->template alloc<200>();
                // fine: < starts explicit qualification
}
```

4 —*end example*] If a name prefixed by the keyword `template` in this way is not the name of a member template function, the program is ill-formed.

14.10.2 Template argument deduction [temp.deduct]

1 Template arguments that can be deduced from the function arguments of a call need not be explicitly specified. [*Example:*

```
void f(Array<dcomplex>& cv, Array<int>& ci)
{
    sort(cv);   // call sort(Array<dcomplex>)
    sort(ci);   // call sort(Array<int>)
}
```

and

```
void g(double d)
{
        int i = convert<int>(d);   // call convert<int,double>(double)
        int c = convert<char>(d);  // call convert<char,double>(double)
}
```

—*end example*]

2 Type deduction is done for each parameter of a function template that contains a reference to a template parameter that is not explicitly specified. The type of the parameter of the function template (call it `P`) is compared to the type of the corresponding argument of the call (call it `A`), and an attempt is made to find types for the template type arguments, and values for the template non-type arguments, that will make `P` after substitution of the deduced values and explicitly-specified values (call that the deduced `P`) compatible with the call argument. Type deduction is done independently for each parameter/argument pair, and the deduced template argument types and values are then combined. If type deduction cannot be done for any parameter/argument pair, or if different parameter/argument pairs yield different deduced values for a given template argument, or if any template argument remains neither deduced nor explicitly specified, template argument deduction fails.

3 If `P` is not a reference type:

— if `A` is an array type, the pointer type produced by the array-to-pointer standard conversion (4.2) is used in place of `A` for type deduction; otherwise,

— if `A` is a function type, the pointer type produced by the function-to-pointer standard conversion (4.3) is used in place of `A` for type deduction; otherwise,

— the cv-unqualified version of `A` is used in place of `A` for type deduction.

If `P` is a reference type, the type referred to by `P` is used in place of `P` for type deduction.

4 In general, the deduction process attempts to find template argument values that will make the deduced `P` identical to `A`. However, there are three cases that allow a difference:

— If the original `P` is a reference type, the deduced `P` (i.e., the type referred to by the reference) can be more cv-qualified than `A`.

— If `P` is a pointer or pointer to member type, `A` can be another pointer or pointer to member type that can be converted to the deduced `P` via a qualification conversion (4.4).

— If `P` is a class, `A` can be a derived class of the deduced `P` having the form *class-template-name*<arguments>. Likewise, if `P` is a pointer to a class, `A` can be a pointer to a derived class of the underlying type of the deduced `P` having the form *class-template-name*<arguments>. These alternatives are considered only if type deduction cannot be done otherwise. If they yield more than one possible deduced `P`, the type deduction fails.

When deducing arguments in the context of taking the address of an overloaded function (13.4), these inexact deductions are not considered.

5 A template type argument `T` or a template non-type argument `i` can be deduced if `P` and `A` have one of the following forms:

6

`T`
cv-list `T`
`T*`
`T&`
`T[`*integer-constant*`]`
class-template-name`<T>`
type`(*)(T)`
type `T::*`
`T(*)()`
`T(*)(T)`
type`[i]`
class-template-name`<i>`

where `(T)` represents parameter lists where at least one parameter type contains a `T`, and `()` represents parameter lists where no parameter contains a `T`. Similarly, `<T>` represents template argument lists where at least one argument contains a `T`, and `<i>` represents template argument lists where at least one argument contains an `i`. These forms can be used in the same way as `T` is for further composition of types. [*Example:*

```
X<int>(*)(char[6])
```

is of the form

class-template-name`<T> (*)(`*type*`[i])`

which is a variant of

type `(*)(T)`

where *type* is `X<int>` and `T` is `char[6]`.]

7 In addition, a *template-parameter* can be deduced from a function or pointer to member function argument if at most one of a set of overloaded functions provides a unique match. [*Example:*

```
template<class T> void f(void(*)(T,int));

void g(int,int);
void g(char,int);

void h(int,int,int);
void h(char,int);

int m()
{
        f(&g);  // error: ambiguous
        f(&h);  // ok: void h(char,int) is a unique match
}
```

—*end example*] Template arguments cannot be deduced from function arguments involving constructs other than the ones specified in here (14.10.2).

8 Template arguments of an explicit instantiation or explicit specialization are deduced (14.4, 14.5) according to these rules specified for deducing function arguments.

9 [*Note:* a major array bound is not part of a function parameter type so it can't be deduced from an argument:

```
template<int i> void f1(int a[10][i]);
template<int i> void f2(int a[i][20]);

void g(int v[10][20])
{
        f1(v);     // ok: i deduced to be 20
        f1<10>(v); // ok
        f2(v);     // error: cannot deduce template-argument i
        f2<10>(v); // ok
}
```

—*end note*]

10 Nontype parameters shall not be used in expressions in the function declaration. The type of the function *template-parameter* shall match the type of the *template-argument* exactly. [*Example:*

```
template<char c> class A { /* ... */ };
template<int i> void f(A<i>);   // error: conversion not allowed
template<int i> void f(A<i+1>); // error: expression not allowed
```

—*end example*]

11 If function *template-argument*s are explicitly specified in a call they are specified in declaration order. Trailing arguments can be left out of a list of explicit *template-argument*s. [*Example:*

```
template<class X, class Y, class Z> X f(Y,Z);

void g()
{
        f<int,char*,double>("aa",3.0);
        f<int,char*>("aa",3.0); // Z is deduced to be double
        f<int>("aa",3.0); // Y is deduced to be char*, and
                          // Z is deduced to be double
        f("aa",3.0); // error X cannot be deduced

}
```

—*end example*]

12 A *template-parameter* cannot be deduced from a default function argument. [*Example:*

```
template <class T> void f(T = 5, T = 7);

void g()
{
        f(1);     // fine: call f<int>(1,7)
        f();      // error: cannot deduce T
        f<int>(); // fine: call f<int>(5,7)
}
```

13 Here is example in which different parameter/argument pairs produce inconsistent template argument deductions:

```
template<class T> void f(T x, T y) { /* ... */ }

struct A { /* ... */ };
struct B : A { /* ... */ };

int g(A a, B b)
{
        f(a,a);  // ok: T is A
        f(b,b);  // ok: T is B
        f(a,b);  // error T could be A or B
        f(b,a);  // error: T could be A or B
}
```

14 Here is an example where a qualification conversion applies between the call argument type and the deduced parameter type:

```
template<class T> void f(const T*) {}
int *p;
void s()
{
        f(p);  // f(const int *)
}
```

15 Here is an example where the deduced parameter type is a derived class of a class template reference:

```
template <class T> struct B { };
template <class T> struct D : public B<T> {};
struct D2 : public B<int> {};
template <class T> void f(B<T>&){}

void main()
{
        D<int> d;
        D2     d2;

        f(d);  // calls f(B<int>&)
        f(d2); // calls f(B<int>&)
}
```

—*end example*]

14.10.3 Overload resolution [temp.over]

1 A function template can be overloaded either by (other) functions of its name or by (other) function templates of that same name. When a call to that name is written (explicitly, or implicitly using the operator notation), template argument deduction (14.10.2) is performed on each function template to find the template argument values (if any) that can be used with that function template to generate a function that can be invoked with the call arguments. For each function template, if the argument deduction succeeds, the deduced template arguments are used to generate a single template function, which is added to the

candidate functions set to be used in overload resolution. The complete set of candidate functions includes all the template functions generated in this way and all of the non-template overloaded functions of the same name. The template functions are treated like any other functions in the remainder of overload resolution, except as explicitly noted.[97)]

2 [*Example:*

```
template<class T> T max(T a, T b) { return a>b?a:b; };

void f(int a, int b, char c, char d)
{
    int m1 = max(a,b);  // max(int a, int b)
    char m2 = max(c,d); // max(char a, char b)
    int m3 = max(a,c);  // error: cannot generate max(int,char)
}
```

3 Adding

```
int max(int,int);
```

to the example above would resolve the third call, by providing a function that could be called for `max(a,c)` after using the standard conversion of `char` to `int` for `c`.

4 Here is an example involving conversions on a function argument involved in *template-parameter* deduction:

```
template<class T> struct B { /* ... */ };
template<class T> struct D : public B<T> { /* ... */ };
template<class T> void f(B<T>&);

void g(B<int>& bi, D<int>& di)
{
        f(bi);  // f(bi)
        f(di);  // f( (B<int>&)di )
}
```

5 Here is an example involving conversions on a function argument not involved in *template-parameter* deduction:

```
template<class T> void f(T*,int);  // #1
template<class T> void f(T,char);  // #2

void h(int* pi, int i, char c)
{
        f(pi,i);  // #1: f<int>(pi,i)
        f(pi,c);  // #2: f<int*>(pi,c)

        f(i,c);   // #2: f<int>(i,c);
        f(i,i);   // #2: f<int>(i,char(i))
}
```

—*end example*]

6 The template definition is needed to generate specializations of a template. However, only a function template declaration is needed to call a specialization. [*Example:*

97) The parameters of template functions contain no template parameter types. The set of conversions allowed on deduced arguments is limited, because the argument deduction process produces template functions with parameters that either match the call arguments exactly or differ only in ways that can be bridged by the allowed limited conversions. Non-deduced arguments allow the full range of conversions.

```
template<class T> void f(T);    // declaration

void g()
{
        f("Annemarie"); // call of f<char*>
}
```

The call of `f` is well formed because of the declaration of `f`, and the program will be ill-formed unless a definition of `f` is present in some translations unit.

7 Here is a case involving explicit specification of some of the template arguments and deduction of the rest:

```
template<class X, class Y> void f(X,Y*);  // #1
template<class X, class Y> void f(X*,Y);  // #2

void g(char* pc, int* pi)
{
        f(0,0); // error: ambiguous: f<int,int>(int,int*)
                  //                 or f<int,int>(int*,int) ?
        f<char*>(pc,pi); // #1: f<char*,int>(char*,int*)
        f<char>(pc,pi);  // #2: f<char,int*>(char*,int*)
}
```

—*end example*]

14.10.4 Overloading and linkage [temp.over.link]

1 It is possible to overload template functions so that specializations of two different template functions have the same type. [*Example:*

```
// file1.c                      // file2.c
template<class T>               template<class T>
void f(T*);                     void f(T);
void g(int* p) {                void h(int* p) {
        f(p); // call f_PT_pi           f(p); // call f_T_pi
}                               }
```

—*end example*]

2 Such specializations are distinct functions and do not violate the ODR.

3 The signature of a specialization of a template function consists of the actual template arguments (whether explicitly specified or deduced) and the signature of the function template.

4 The signature of a function template consists of its function signature and its return type and template parameter list. The names of the template parameters are significant only for establishing the relationship between the template parameters and the rest of the signature.

14.10.5 Overloading and specialization [temp.over.spec]

1 A template function can be overloaded by a function with the same type as a potentially generated function. [*Example:*

```
template<class T> T max(T a, T b) { return a>b?a:b; }
int max(int a, int b);

int min(int a, int b);
template<class T> T min(T a, T b) { return a<b?a:b; }
```

—*end example*] Such an overloaded function is a specialization but not an explicit specialization. The declaration simply guides the overload resolution. [*Note:* this implies that a definition of `max(int,int)` and `min(int,int)` will be implicitly generated from the templates. If such implicit instantiation is not wanted, the explicit specialization syntax should be used instead:

```
template<class T> T max(T a, T b) { return a>b?a:b; }
int max<int>(int a, int b);
```

—*end note*]

2 Defining a function with the same type as a template specialization that is called is ill-formed. [*Example:*

```
template<class T> T max(T a, T b) { return a>b?a:b; }
int max(int a, int b) { return a>b?a:b; }

void f(int x, int y)
{
        max(x,y); // error: duplicate definition of max()
}
```

If the two definitions of `max()` are not in the same translation unit the diagnostic is not required. If a separate definition of a function `max(int,int)` is needed, the specialization syntax can be used. If the conversions enabled by an ordinary declaration are also needed, both can be used.

```
template<class T> T max(T a, T b) { return a>b?a:b; }
int max<>(int a, int b) { /* ... */ }

void g(char x, int y)
{
        max(x,y); // error: no exact match, and no conversions allowed
}

int max(int,int);

void f(char x, int y)
{
        max(x,y); // max<int>(int(x),y)
}
```

—*end example*]

3 An explicit specialization of a function template shall be `inline` or `static` only if it is explicitly declared to be, and independently of whether its function template is. [*Example:*

```
template<class T> void f(T) { /* ... */ }
template<class T> inline T g(T) { /* ... */ }

inline void f<>(int) { /* ... */ } // ok: inline
int g<>(int) { /* ... */ } // ok: not inline
```

—*end example*]

14.10.6 Partial ordering of function templates [temp.func.order]

1 Given two function templates, whether one is more specialized than another can be determined by transforming each template in turn and using argument deduction to compare it to the other.

2 The transformation used is:

— For each type template parameter, synthesize a unique type and substitute that for each occurrence of that parameter in the function parameter list.

— for each nontype template parameter, synthesize a unique value of the appropriate type and substitute that for each occurrence of that parameter in the function parameter list.

3 Using the transformed function parameter list, perform argument deduction against the other function template (14.10.2). The transformed template is at least as specialized as the other if, and only if, the deduction succeeds.

4 A template is more specialized than another if, and only if, it is at least as specialized as the other template and that template is not at least as specialized as the first. [*Example:*

```
template<class T> class A {};

template<class T> void f(T);
template<class T> void f(T*);
template<class T> void f(const T*);

template<class T> void g(T);
template<class T> void g(T&);

template<class T> void h(const T&);
template<class T> void h(A<T>);

void m() {
        const int *p;
        f(p);    // f(const T*) is more specialized than f(T) or f(T*)
        float x;
        g(x);    // Ambiguous: g(T) or g(T&)
        A<int> z;
        h(z);    // h(A<T>) is more specialized than f(const T&)
        const A<int> z2;
        h(z2);   // h(const T&) is called because h(A<T>) is not callable
}
```

—*end example*]

14.11 Member function templates [temp.mem.func]

1 A member function of a template class is implicitly a template function with the *template-parameter*s of its class as its *template-parameter*s. [*Example:*

```
template<class T> class Array {
    T* v;
    int sz;
public:
    explicit Array(int);
    T& operator[](int);
    T& elem(int i) { return v[i]; }
    // ...
};
```

declares three function templates. The subscript function might be defined like this:

```
template<class T> T& Array<T>::operator[](int i)
{
    if (i<0 || sz<=i) error("Array: range error");
    return v[i];
}
```

2 The *template-argument* for `Array<T>::operator[]()` will be determined by the `Array` to which the subscripting operation is applied.

```
Array<int> v1(20);
Array<dcomplex> v2(30);

v1[3] = 7;              // Array<int>::operator[]()
v2[3] = dcomplex(7,8);  // Array<dcomplex>::operator[]()
```

—*end example*]

14.12 Friends [temp.friend]

1 A friend function of a template can be a template function or a non-template function. [*Example:*

```
template<class T> class task {
    // ...
    friend void next_time();
    friend task<T>* preempt(task<T>*);
    friend task* prmt(task*);           // task is task<T>
    friend class task<int>;
    // ...
};
```

Here, `next_time()` and `task<int>` become friends of all `task` classes, and each `task` has appropriately typed functions `preempt()` and `prmt()` as friends. The `preempt` functions might be defined as a template.

```
template<class T> task<T>* preempt(task<T>* t) { /* ... */ }
```

—*end example*]

2 A friend template shall not be defined within a class. [*Example:*

```
class A {
        template<class T> friend B;    // ok
        template<class T> friend void f(T); // ok

        template<class T> friend BB { /* ... /* }; // error
        template<class T> friend void ff(T){ /* ... /* } // error
};
```

—*end example*] [*Note:* a `friend` declaration can add a name to an enclosing scope (14.2.4).]

14.13 Static members and variables [temp.static]

1 Each template class or function generated from a template has its own copies of any static variables or members. [*Example:*

```
template<class T> class X {
    static T s;
    // ...
};

X<int> aa;
X<char*> bb;
```

Here `X<int>` has a static member `s` of type `int` and `X<char*>` has a static member `s` of type `char*`.]

2 Static class member templates are defined similarly to member function templates. [*Example:*

```
template<class T> T X<T>::s = 0;

int X<int>::s = 3;
```

3 Similarly,

```
template<class T> f(T* p)
{
    static T s;
    // ...
};
```

```
        void g(int a, char* b)
        {
            f(&a);  // call f<int>(int*)
            f(&b);  // call f<char*>(char**)
        }
```

Here `f<int>(int*)` has a static member `s` of type `int` and `f<char*>(char**)` has a static member `s` of type `char*`.]

15 Exception handling [except]

1 Exception handling provides a way of transferring control and information from a point in the execution of a program to an *exception handler* associated with a point previously passed by the execution. A handler will be invoked only by a *throw-expression* invoked in code executed in the handler's *try-block* or in functions called from the handler's *try-block*.

> *try-block:*
> `try` *compound-statement handler-seq*
>
> *function-try-block:*
> `try` *ctor-initializer-opt function-body 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*$_{opt}$

A *try-block* is a *statement* (6). A *throw-expression* is of type `void`. A *throw-expression* is sometimes referred to as a "*throw-point*." Code that executes a *throw-expression* is said to "throw an exception;" code that subsequently gets control is called a "*handler*." [*Note:* within this clause "try block" is taken to mean both *try-block* and *function-try-block*.]

2 A `goto`, `break`, `return`, or `continue` statement can be used to transfer control out of a try block or handler, but not into one. When this happens, each variable declared in the try block will be destroyed in the context that directly contains its declaration. [*Example:*

```
lab:  try {
           T1 t1;
           try {
                  T2 t2;
                  if (condition)
                        goto lab;
           } catch(...) { /* handler 2 */ }
      } catch(...) { /* handler 1 */ }
```

Here, executing `goto lab;` will destroy first `t2`, then `t1`. Any exception raised while destroying `t2` will result in executing *handler 2*; any exception raised while destroying `t1` will result in executing *handler 1*.]

3 A *function-try-block* associates a *handler-seq* with the *ctor-initializer*, if present, and the *function-body*. An exception thrown during the execution of the initializer expressions in the *ctor-initializer* or during the execution of the *function-body* transfers control to a handler in a *function-try-block* in the same way as an exception thrown during the execution of a *try-block* transfers control to other handlers.

15.1 Throwing an exception [except.throw]

1 Throwing an exception transfers control to a handler. An object is passed and the type of that object determines which handlers can catch it. [*Example:*

```
throw "Help!";
```

can be caught by a *handler* of some `char*` type:

```
try {
    // ...
}
catch(const char* p) {
    // handle character string exceptions here
}
```

and

```
class Overflow {
    // ...
public:
    Overflow(char,double,double);
};

void f(double x)
{
    // ...
    throw Overflow('+',x,3.45e107);
}
```

can be caught by a handler

```
try {
    // ...
    f(1.2);
    // ...
}
catch(Overflow& oo) {
    // handle exceptions of type Overflow here
}
```

—*end example*]

2 When an exception is thrown, control is transferred to the nearest handler with an appropriate type; "nearest" means the handler whose try block was most recently entered by the thread of control and not yet exited; "appropriate type" is defined in 15.3.

3 A *throw-expression* initializes a temporary object of the static type of the operand of `throw`, ignoring the top-level *cv-qualifier*s of the operand's type, and uses that temporary to initialize the appropriately-typed variable named in the handler. Except for the restrictions on type matching mentioned in 15.3 and the use of a temporary object, the operand of `throw` is treated exactly as a function argument in a call (5.2.2) or the operand of a `return` statement.

4 The memory for the temporary copy of the exception being thrown is allocated in an implementation-defined way. The temporary persists as long as there is a handler being executed for that exception. In particular, if a handler exits by executing a `throw;` statement, that passes control to another handler for the same exception, so the temporary remains. If the use of the temporary object can be eliminated without changing the meaning of the program except for the execution of constructors and destructors associated with the use of the temporary object (12.2), then the exception in the handler can be initialized directly with

the argument of the throw expression.

5 A *throw-expression* with no operand rethrows the exception being handled without copying it. [*Example:* code that must be executed because of an exception yet cannot completely handle the exception can be written like this:

```
try {
    // ...
}
catch (...) {  // catch all exceptions

    // respond (partially) to exception

    throw;     // pass the exception to some
               // other handler
}
```

—*end example*]

6 The exception thrown is the one most recently caught and not finished. An exception is considered caught when initialization is complete for the formal parameter of the corresponding catch clause, or when `terminate()` or `unexpected()` is entered due to a throw. An exception is considered finished when the corresponding catch clause exits.

7 If no exception is presently being handled, executing a *throw-expression* with no operand calls `terminate()` (15.5.1).

15.2 Constructors and destructors [except.ctor]

1 As control passes from a throw-point to a handler, destructors are invoked for all automatic objects constructed since the try block was entered.

2 An object that is partially constructed will have destructors executed only for its fully constructed subobjects. Should a constructor for an element of an automatic array throw an exception, only the constructed elements of that array will be destroyed. If the object or array was allocated in a *new-expression*, the storage occupied by that object is sometimes deleted also (5.3.4).

3 [*Note:* the process of calling destructors for automatic objects constructed on the path from a try block to a *throw-expression* is called "*stack unwinding*."]

15.3 Handling an exception [except.handle]

1 The *exception-declaration* in a *handler* describes the type(s) of exceptions that can cause that handler to be executed. The *exception-declaration* shall not denote an incomplete type.

2 A *handler* with type `T`, `const T`, `T&`, or `const T&` is a match for a *throw-expression* with an object of type `E` if

[1] `T` and `E` are the same type, or

[2] `T` is a public base class of `E`, or

[3] `T` is a pointer type and `E` is a pointer type that can be converted to `T` by a standard pointer conversion (4.10) not involving conversions to pointers to private or protected base classes.

[*Example:*

```
class Matherr { /* ... */ virtual vf(); };
class Overflow: public Matherr { /* ... */ };
class Underflow: public Matherr { /* ... */ };
class Zerodivide: public Matherr { /* ... */ };
```

```
void f()
{
    try {
        g();
    }

    catch (Overflow oo) {
        // ...
    }
    catch (Matherr mm) {
        // ...
    }
}
```

Here, the `Overflow` handler will catch exceptions of type `Overflow` and the `Matherr` handler will catch exceptions of type `Matherr` and all types publicly derived from `Matherr` including `Underflow` and `Zerodivide`.]

3 The handlers for a try block are tried in order of appearance. That makes it possible to write handlers that can never be executed, for example by placing a handler for a derived class after a handler for a corresponding base class.

4 A `...` in a handler's *exception-declaration* functions similarly to `...` in a function parameter declaration; it specifies a match for any exception. If present, a `...` handler shall be the last handler for its try block.

5 If no match is found among the handlers for a try block, the search for a matching handler continues in a dynamically surrounding try block.

6 An exception is considered handled upon entry to a handler. [*Note:* the stack will have been unwound at that point.]

7 If no matching handler is found in a program, the function `terminate()` (15.5.1) is called. Whether or not the stack is unwound before calling `terminate()` is implementation-defined.

8 Referring to any non-static member or base class of the object in the handler of a *function-try-block* of a constructor or destructor of the object results in undefined behavior.

9 The fully constructed base classes and members of an object shall be destroyed before entering the handler of a *function-try-block* of a constructor or destructor for that object.

10 The scope and lifetime of the parameters of a function or constructor extend into the handlers of a *function-try-block.*

11 If the handlers of a *function-try-block* contain a jump into the body of a constructor or destructor, the program is ill-formed.

12 If a return statement appears in a handler of *function-try-block* of a constructor, the program is ill-formed.

13 The exception being handled shall be rethrown if control reaches the end of a handler of the *function-try-block* of a constructor or destructor. Otherwise, the function shall return when control reaches the end of a handler for the *function-try-block* (6.6.3).

14

15.4 Exception specifications [except.spec]

1 A function declaration lists exceptions that its function might directly or indirectly throw by using an *exception-specification* as a suffix of its declarator.

exception-specification:
    `throw (` *type-id-list*$_{opt}$ `)`

type-id-list:
type-id
type-id-list , *type-id*

An *exception-specification* shall appear only on a function declarator in a declaration or definition. An *exception-specification* shall not appear in a typedef declaration. [*Example:*

```
void f() throw(int);             // OK
void (*fp) throw (int);          // OK
void g(void pfa() throw(int));   // OK
typedef int (*pf)() throw(int);  // ill-formed
```

—*end example*]

2 If any declaration of a function has an *exception-specification*, all declarations, including the definition, of that function shall have an *exception-specification* with the same set of *type-id*s. If a virtual function has an *exception-specification*, all declarations, including the definition, of any function that overrides that virtual function in any derived class shall have an *exception-specification* at least as restrictive as that in the base class. [*Example:*

```
struct B {
    virtual void f() throw (int, double);
    virtual void g();
};

struct D: B {
    void f();                    // ill-formed
    void g() throw (int);        // OK
};
```

—*end example*] The declaration of `D::f` is ill-formed because it allows all exceptions, whereas `B::f` allows only `int` and `double`. Similarly, any function or pointer to function assigned to, or initializing, a pointer to function shall have an *exception-specification* at least as restrictive as that of the pointer or function being assigned to or initialized. [*Example:*

```
void (*pf1)();    // no exception specification
void (*pf2) throw(A);

void f()
{
        pf1 = pf2;  // ok: pf1 is less restrictive
        pf2 = pf1;  // error: pf2 is more restrictive
}
```

—*end example*]

3 In such an assignment or initialization, *exception-specification*s on return types and parameter types shall match exactly.

4 In other assignments or initializations, *exception-specification*s shall match exactly.

5 Calling a function through a declaration whose *exception-specification* is less restrictive that that of the function's definition is ill-formed. No diagnostic is required.

6 Types shall not be defined in *exception-specification*s.

7 An *exception-specification* can include the same class more than once and can include classes related by inheritance, even though doing so is redundant. An exception specification can include identifiers that represent incomplete types. An exception can also include the name of the predefined class `bad_exception`.

8 If a class `X` is in the *type-id-list* of the *exception-specification* of a function, that function is said to *allow* exception objects of class `X` or any class publicly and unambiguously derived from `X`. Similarly, if a pointer type `Y*` is in the *type-id-list* of the *exception-specification* of a function, the function allows

exceptions of type `Y*` or that are pointers to any type publicly and unambiguously derived from `Y*`.

9 Whenever an exception is thrown and the search for a handler (15.3) encounters the outermost block of a function with an *exception-specification*, the function `unexpected()` is called (15.5.2) if the *exception-specification* does not allow the exception. [*Example:*

```
class X { };
class Y { };
class Z: public X { };
class W { };

void f() throw (X, Y)
{
    int n = 0;
    if (n) throw X();        // OK
    if (n) throw Z();        // also OK
    throw W();               // will call unexpected()
}
```

—*end example*]

10 The function `unexpected()` may throw an exception that will satisfy the *exception-specification* for which it was invoked, and in this case the search for another handler will continue at the call of the function with this *exception-specification* (see 15.5.2), or it may call terminate.

11 An implementation shall not reject an expression merely because when executed it throws or might throw an exception that the containing function does not allow. [*Example:*

```
extern void f() throw(X, Y);

void g() throw(X)
{
        f();                 // OK
}
```

the call to `f` is well-formed even though when called, `f` might throw exception `Y` that `g` does not allow.]

12 A function with no *exception-specification* allows all exceptions. A function with an empty *exception-specification*, `throw()`, does not allow any exceptions.

13 An *exception-specification* is not considered part of a function's type.

15.5 Special functions [except.special]

1 The exception handling mechanism relies on two functions, `terminate()` and `unexpected()`, for coping with errors related to the exception handling mechanism itself (18.6).

15.5.1 The `terminate()` function [except.terminate]

1 In the following situations exception handling must be abandoned for less subtle error handling techniques:

— when a exception handling mechanism, after completing evaluation of the object to be thrown but before completing the initialization of the *exception-declaration* in the matching handler, calls a user function that exits via an uncaught exception,[98)]

— when the exception handling mechanism cannot find a handler for a thrown exception (see 15.3),

— when the implementation's exception handling mechanism encounters some internal error, or

— when an attempt by the implementation to destroy an object during stack unwinding exits using an exception.

[98)] For example, if the object being thrown is of a class with a copy constructor, `terminate()` will be called if that copy constructor exits with an exception during a `throw`.

2 In such cases,

```
void terminate();
```

is called (18.6.2).

15.5.2 The `unexpected()` function [except.unexpected]

1 If a function with an *exception-specification* throws an exception that is not listed in the *exception-specification*, the function

```
void unexpected();
```

is called (18.6.1).

2 The `unexpected()` function shall not return, but it can throw (or re-throw) an exception. If it throws a new exception which is allowed by the exception specification which previously was violated, then the search for another handler will continue at the call of the function whose exception specification was violated. If it throws or rethrows an exception an exception that the *exception-specification* does not allow then the following happens: if the *exception-specification* does not include the name of the predefined exception `bad_exception` then the function `terminate()` is called, otherwise the thrown exception is replaced by an implementation-defined object of the type `bad_exception` and the search for another handler will continue at the call of the function whose *exception-specification* was violated.

3 Thus, an *exception-specification* guarantees that only the listed exceptions will be thrown. If the *exception-specification* includes the name `bad_exception` then any exception not on the list may be replaced by `bad_exception` within the function `unexpected()`.

15.6 Exceptions and access [except.access]

1 If the *exception-declaration* in a catch clause has class type, and the function in which the catch clause occurs does not have access to the destructor of that class, the program is ill-formed.

2 An object can be thrown if it can be copied and destroyed in the context of the function in which the throw occurs.

16 Preprocessing directives [cpp]

1 A preprocessing directive consists of a sequence of preprocessing tokens that begins with a # preprocessing token that is either the first character in the source file (optionally after white space containing no new-line characters) or that follows white space containing at least one new-line character, and is ended by the next new-line character.[99)]

preprocessing-file:
    group$_{opt}$

group:
    group-part
    group group-part

group-part:
    pp-tokens$_{opt}$ *new-line*
    if-section
    control-line

if-section:
    if-group elif-groups$_{opt}$ *else-group*$_{opt}$ *endif-line*

if-group:
    `# if` *constant-expression new-line group*$_{opt}$
    `# ifdef` *identifier new-line group*$_{opt}$
    `# ifndef` *identifier new-line group*$_{opt}$

elif-groups:
    elif-group
    elif-groups elif-group

elif-group:
    `# elif` *constant-expression new-line group*$_{opt}$

else-group:
    `# else` *new-line group*$_{opt}$

endif-line:
    `# endif` *new-line*

[99)] Thus, preprocessing directives are commonly called "lines." These "lines" have no other syntactic significance, as all white space is equivalent except in certain situations during preprocessing (see the # character string literal creation operator in 16.3.2, for example).

control-line:
`# include` *pp-tokens new-line*
`# define` *identifier replacement-list new-line*
`# define` *identifier lparen identifier-list*$_{opt}$ `)` *replacement-list new-line*
`# undef` *identifier new-line*
`# line` *pp-tokens new-line*
`# error` *pp-tokens*$_{opt}$ *new-line*
`# pragma` *pp-tokens*$_{opt}$ *new-line*
`#` *new-line*

lparen:
the left-parenthesis character without preceding white-space

replacement-list:
pp-tokens$_{opt}$

pp-tokens:
preprocessing-token
pp-tokens preprocessing-token

new-line:
the new-line character

2 The only white-space characters that shall appear between preprocessing tokens within a preprocessing directive (from just after the introducing # preprocessing token through just before the terminating new-line character) are space and horizontal-tab (including spaces that have replaced comments or possibly other white-space characters in translation phase 3).

3 The implementation can process and skip sections of source files conditionally, include other source files, and replace macros. These capabilities are called *preprocessing*, because conceptually they occur before translation of the resulting translation unit.

4 The preprocessing tokens within a preprocessing directive are not subject to macro expansion unless otherwise stated.

16.1 Conditional inclusion [cpp.cond]

1 The expression that controls conditional inclusion shall be an integral constant expression except that: it shall not contain a cast; identifiers (including those lexically identical to keywords) are interpreted as described below;[100] and it may contain unary operator expressions of the form

`defined` *identifier*

or

`defined (` *identifier* `)`

which evaluate to `1` if the identifier is currently defined as a macro name (that is, if it is predefined or if it has been the subject of a `#define` preprocessing directive without an intervening `#undef` directive with the same subject identifier), zero if it is not.

2 Each preprocessing token that remains after all macro replacements have occurred shall be in the lexical form of a token (2.5).

3 Preprocessing directives of the forms

`# if` *constant-expression new-line group*$_{opt}$
`# elif` *constant-expression new-line group*$_{opt}$

check whether the controlling constant expression evaluates to nonzero.

[100] Because the controlling constant expression is evaluated during translation phase 4, all identifiers either are or are not macro names — there simply are no keywords, enumeration constants, and so on.

4 Prior to evaluation, macro invocations in the list of preprocessing tokens that will become the controlling constant expression are replaced (except for those macro names modified by the `defined` unary operator), just as in normal text. If the token `defined` is generated as a result of this replacement process or use of the `defined` unary operator does not match one of the two specified forms prior to macro replacement, the behavior is undefined. After all replacements due to macro expansion and the `defined` unary operator have been performed, all remaining identifiers are replaced with the pp-number `0`, and then each preprocessing token is converted into a token. The resulting tokens comprise the controlling constant expression which is evaluated according to the rules of 5.19 using arithmetic that has at least the ranges specified in 18.2, except that `int` and `unsigned int` act as if they have the same representation as, respectively, `long` and `unsigned long`. This includes interpreting character literals, which may involve converting escape sequences into execution character set members. Whether the numeric value for these character literals matches the value obtained when an identical character literal occurs in an expression (other than within a `#if` or `#elif` directive) is implementation-defined.[101] Also, whether a single-character character literal may have a negative value is implementation-defined.

5 Preprocessing directives of the forms

`# ifdef` *identifier new-line group*$_{opt}$
`# ifndef` *identifier new-line group*$_{opt}$

check whether the identifier is or is not currently defined as a macro name. Their conditions are equivalent to `#if defined` *identifier* and `#if !defined` *identifier* respectively.

6 Each directive's condition is checked in order. If it evaluates to false (zero), the group that it controls is skipped: directives are processed only through the name that determines the directive in order to keep track of the level of nested conditionals; the rest of the directives' preprocessing tokens are ignored, as are the other preprocessing tokens in the group. Only the first group whose control condition evaluates to true (nonzero) is processed. If none of the conditions evaluates to true, and there is a `#else` directive, the group controlled by the `#else` is processed; lacking a `#else` directive, all the groups until the `#endif` are skipped.[102]

16.2 Source file inclusion [cpp.include]

1 A `#include` directive shall identify a header or source file that can be processed by the implementation.

2 A preprocessing directive of the form

`# include <`*h-char-sequence*`>` *new-line*

searches a sequence of implementation-defined places for a header identified uniquely by the specified sequence between the `<` and `>` delimiters, and causes the replacement of that directive by the entire contents of the header. How the places are specified or the header identified is implementation-defined.

3 A preprocessing directive of the form

`# include "`*q-char-sequence*`"` *new-line*

causes the replacement of that directive by the entire contents of the source file identified by the specified sequence between the `"` delimiters. The named source file is searched for in an implementation-defined manner. If this search is not supported, or if the search fails, the directive is reprocessed as if it read

`# include <`*h-char-sequence*`>` *new-line*

with the identical contained sequence (including `>` characters, if any) from the original directive.

[101] Thus, the constant expression in the following `#if` directive and `if` statement is not guaranteed to evaluate to the same value in these two contexts.

```
#if 'z' - 'a' == 25
if ('z' - 'a' == 25)
```

[102] As indicated by the syntax, a preprocessing token shall not follow a `#else` or `#endif` directive before the terminating new-line character. However, comments may appear anywhere in a source file, including within a preprocessing directive.

4 A preprocessing directive of the form

```
        # include pp-tokens new-line
```

(that does not match one of the two previous forms) is permitted. The preprocessing tokens after `include` in the directive are processed just as in normal text. (Each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens.)The directive resulting after all replacements shall match one of the two previous forms.[103)] The method by which a sequence of preprocessing tokens between a `<` and a `>` preprocessing token pair or a pair of `"` characters is combined into a single header name preprocessing token is implementation-defined.

5 There shall be an implementation-defined mapping between the delimited sequence and the external source file name. The implementation shall provide unique mappings for sequences consisting of one or more *nondigit*s (2.7) followed by a period (`.`) and a single *nondigit*. The implementation may ignore the distinctions of alphabetical case and restrict the mapping to six significant characters before the period.

6 A `#include` preprocessing directive may appear in a source file that has been read because of a `#include` directive in another file, up to an implementation-defined nesting limit.

7 [*Example:* The most common uses of `#include` preprocessing directives are as in the following:

```
        #include <stdio.h>
        #include "myprog.h"
```

—*end example*]

8 [*Example:* Here is a macro-replaced `#include` directive:

```
        #if VERSION == 1
                #define INCFILE  "vers1.h"
        #elif VERSION == 2
                #define INCFILE  "vers2.h"   /* and so on */
        #else
                #define INCFILE  "versN.h"
        #endif
        #include INCFILE
```

—*end example*]

16.3 Macro replacement [cpp.replace]

1 Two replacement lists are identical if and only if the preprocessing tokens in both have the same number, ordering, spelling, and white-space separation, where all white-space separations are considered identical.

2 An identifier currently defined as a macro without use of lparen (an *object-like* macro) may be redefined by another `#define` preprocessing directive provided that the second definition is an object-like macro definition and the two replacement lists are identical.

3 An identifier currently defined as a macro using lparen (a *function-like* macro) may be redefined by another `#define` preprocessing directive provided that the second definition is a function-like macro definition that has the same number and spelling of parameters, and the two replacement lists are identical.

4 The number of arguments in an invocation of a function-like macro shall agree with the number of parameters in the macro definition, and there shall exist a `)` preprocessing token that terminates the invocation.

5 A parameter identifier in a function-like macro shall be uniquely declared within its scope.

6 The identifier immediately following the `define` is called the *macro name*. There is one name space for macro names. Any white-space characters preceding or following the replacement list of preprocessing tokens are not considered part of the replacement list for either form of macro.

103) Note that adjacent string literals are not concatenated into a single string literal (see the translation phases in 2.1); thus, an expansion that results in two string literals is an invalid directive.

7 If a # preprocessing token, followed by an identifier, occurs lexically at the point at which a preprocessing directive could begin, the identifier is not subject to macro replacement.

8 A preprocessing directive of the form

`# define` *identifier replacement-list new-line*

defines an object-like macro that causes each subsequent instance of the macro name[104)] to be replaced by the replacement list of preprocessing tokens that constitute the remainder of the directive. The replacement list is then rescanned for more macro names as specified below.

9 A preprocessing directive of the form

`# define` *identifier lparen identifier-list*$_{opt}$ `)` *replacement-list new-line*

defines a function-like macro with parameters, similar syntactically to a function call. The parameters are specified by the optional list of identifiers, whose scope extends from their declaration in the identifier list until the new-line character that terminates the `#define` preprocessing directive. Each subsequent instance of the function-like macro name followed by a `(` as the next preprocessing token introduces the sequence of preprocessing tokens that is replaced by the replacement list in the definition (an invocation of the macro). The replaced sequence of preprocessing tokens is terminated by the matching `)` preprocessing token, skipping intervening matched pairs of left and right parenthesis preprocessing tokens. Within the sequence of preprocessing tokens making up an invocation of a function-like macro, new-line is considered a normal white-space character.

10 The sequence of preprocessing tokens bounded by the outside-most matching parentheses forms the list of arguments for the function-like macro. The individual arguments within the list are separated by comma preprocessing tokens, but comma preprocessing tokens between matching inner parentheses do not separate arguments. If (before argument substitution) any argument consists of no preprocessing tokens, the behavior is undefined. If there are sequences of preprocessing tokens within the list of arguments that would otherwise act as preprocessing directives, the behavior is undefined.

16.3.1 Argument substitution [cpp.subst]

1 After the arguments for the invocation of a function-like macro have been identified, argument substitution takes place. A parameter in the replacement list, unless preceded by a # or ## preprocessing token or followed by a ## preprocessing token (see below), is replaced by the corresponding argument after all macros contained therein have been expanded. Before being substituted, each argument's preprocessing tokens are completely macro replaced as if they formed the rest of the translation unit; no other preprocessing tokens are available.

16.3.2 The # operator [cpp.stringize]

1 Each # preprocessing token in the replacement list for a function-like macro shall be followed by a parameter as the next preprocessing token in the replacement list.

2 If, in the replacement list, a parameter is immediately preceded by a # preprocessing token, both are replaced by a single character string literal preprocessing token that contains the spelling of the preprocessing token sequence for the corresponding argument. Each occurrence of white space between the argument's preprocessing tokens becomes a single space character in the character string literal. White space before the first preprocessing token and after the last preprocessing token comprising the argument is deleted. Otherwise, the original spelling of each preprocessing token in the argument is retained in the character string literal, except for special handling for producing the spelling of string literals and character literals: a `\` character is inserted before each `"` and `\` character of a character literal or string literal (including the delimiting `"` characters). If the replacement that results is not a valid character string literal, the behavior is undefined. The order of evaluation of # and ## operators is unspecified.

[104)] Since, by macro-replacement time, all character literals and string literals are preprocessing tokens, not sequences possibly containing identifier-like subsequences (see 2.1.1.2, translation phases), they are never scanned for macro names or parameters.

16.3.3 The ## operator [cpp.concat]

1 A ## preprocessing token shall not occur at the beginning or at the end of a replacement list for either form of macro definition.

2 If, in the replacement list, a parameter is immediately preceded or followed by a ## preprocessing token, the parameter is replaced by the corresponding argument's preprocessing token sequence.

3 For both object-like and function-like macro invocations, before the replacement list is reexamined for more macro names to replace, each instance of a ## preprocessing token in the replacement list (not from an argument) is deleted and the preceding preprocessing token is concatenated with the following preprocessing token. If the result is not a valid preprocessing token, the behavior is undefined. The resulting token is available for further macro replacement. The order of evaluation of ## operators is unspecified.

16.3.4 Rescanning and further replacement [cpp.rescan]

1 After all parameters in the replacement list have been substituted, the resulting preprocessing token sequence is rescanned with all subsequent preprocessing tokens of the source file for more macro names to replace.

2 If the name of the macro being replaced is found during this scan of the replacement list (not including the rest of the source file's preprocessing tokens), it is not replaced. Further, if any nested replacements encounter the name of the macro being replaced, it is not replaced. These nonreplaced macro name preprocessing tokens are no longer available for further replacement even if they are later (re)examined in contexts in which that macro name preprocessing token would otherwise have been replaced.

3 The resulting completely macro-replaced preprocessing token sequence is not processed as a preprocessing directive even if it resembles one.

16.3.5 Scope of macro definitions [cpp.scope]

1 A macro definition lasts (independent of block structure) until a corresponding `#undef` directive is encountered or (if none is encountered) until the end of the translation unit.

2 A preprocessing directive of the form

`# undef` *identifier new-line*

causes the specified identifier no longer to be defined as a macro name. It is ignored if the specified identifier is not currently defined as a macro name.

3 [*Note:* The simplest use of this facility is to define a "manifest constant," as in

```
#define TABSIZE 100
int table[TABSIZE];
```

4 The following defines a function-like macro whose value is the maximum of its arguments. It has the advantages of working for any compatible types of the arguments and of generating in-line code without the overhead of function calling. It has the disadvantages of evaluating one or the other of its arguments a second time (including side effects) and generating more code than a function if invoked several times. It also cannot have its address taken, as it has none.

```
#define max(a, b) ((a) > (b) ? (a) : (b))
```

The parentheses ensure that the arguments and the resulting expression are bound properly.

5 To illustrate the rules for redefinition and reexamination, the sequence

```
#define x    3
#define f(a) f(x * (a))
#undef  x
#define x    2
#define g    f
#define z    z[0]
#define h    g(~
#define m(a) a(w)
#define w    0,1
#define t(a) a

f(y+1) + f(f(z)) % t(t(g)(0) + t)(1);
g(x+(3,4)-w) | h 5) & m
        (f)^m(m);
```

results in

```
f(2 * (y+1)) + f(2 * (f(2 * (z[0])))) % f(2 * (0)) + t(1);
f(2 * (2+(3,4)-0,1)) | f(2 * (~5)) & f(2 * (0,1))^m(0,1);
```

6 To illustrate the rules for creating character string literals and concatenating tokens, the sequence

```
#define str(s)      # s
#define xstr(s)     str(s)
#define debug(s, t) printf("x" # s "= %d, x" # t "= %s", \
                           x ## s, x ## t)
#define INCFILE(n)  vers ## n  /* from previous #include example */
#define glue(a, b)  a ## b
#define xglue(a, b) glue(a, b)
#define HIGHLOW     "hello"
#define LOW         LOW ", world"

debug(1, 2);
fputs(str(strncmp("abc\0d", "abc", '\4')  /* this goes away */
        == 0) str(: @\n), s);
#include xstr(INCFILE(2).h)
glue(HIGH, LOW);
xglue(HIGH, LOW)
```

results in

```
printf("x" "1" "= %d, x" "2" "= %s", x1, x2);
fputs("strncmp(\"abc\\0d\", \"abc\", '\\4') == 0" ": @\n", s);
#include "vers2.h"    (after macro replacement, before file access)
"hello";
"hello" ", world"
```

or, after concatenation of the character string literals,

```
printf("x1= %d, x2= %s", x1, x2);
fputs("strncmp(\"abc\\0d\", \"abc\", '\\4') == 0: @\n", s);
#include "vers2.h"    (after macro replacement, before file access)
"hello";
"hello, world"
```

Space around the # and ## tokens in the macro definition is optional.

7 And finally, to demonstrate the redefinition rules, the following sequence is valid.

```
#define OBJ_LIKE      (1-1)
#define OBJ_LIKE      /* white space */ (1-1) /* other */
#define FTN_LIKE(a)   ( a )
#define FTN_LIKE( a )(          /* note the white space */ \
                                a /* other stuff on this line
                                  */ )
```

But the following redefinitions are invalid:

```
        #define OBJ_LIKE    (0)     /* different token sequence */
        #define OBJ_LIKE    (1 - 1) /* different white space */
        #define FTN_LIKE(b) ( a )   /* different parameter usage */
        #define FTN_LIKE(b) ( b )   /* different parameter spelling */
```

—*end note*]

16.4 Line control [cpp.line]

1 The string literal of a `#line` directive, if present, shall be a character string literal.

2 The *line number* of the current source line is one greater than the number of new-line characters read or introduced in translation phase 1 (2.1) while processing the source file to the current token.

3 A preprocessing directive of the form

`# line` *digit-sequence new-line*

causes the implementation to behave as if the following sequence of source lines begins with a source line that has a line number as specified by the digit sequence (interpreted as a decimal integer). The digit sequence shall not specify zero, nor a number greater than 32767.

4 A preprocessing directive of the form

`# line` *digit-sequence* `"`*s-char-sequence*$_{opt}$`"` *new-line*

sets the line number similarly and changes the presumed name of the source file to be the contents of the character string literal.

5 A preprocessing directive of the form

`# line` *pp-tokens new-line*

(that does not match one of the two previous forms) is permitted. The preprocessing tokens after `line` on the directive are processed just as in normal text (each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens). The directive resulting after all replacements shall match one of the two previous forms and is then processed as appropriate.

16.5 Error directive [cpp.error]

1 A preprocessing directive of the form

`# error` *pp-tokens*$_{opt}$ *new-line*

causes the implementation to produce a diagnostic message that includes the specified sequence of preprocessing tokens.

16.6 Pragma directive [cpp.pragma]

1 A preprocessing directive of the form

`# pragma` *pp-tokens*$_{opt}$ *new-line*

causes the implementation to behave in an implementation-defined manner. Any pragma that is not recognized by the implementation is ignored.

16.7 Null directive [cpp.null]

1 A preprocessing directive of the form

`#` *new-line*

has no effect.

16.8 Predefined macro names [cpp.predefined]

1 The following macro names shall be defined by the implementation:

`__LINE__` The line number of the current source line (a decimal constant).

`__FILE__` The presumed name of the source file (a character string literal).

`__DATE__` The date of translation of the source file (a character string literal of the form `"Mmm dd yyyy"`, where the names of the months are the same as those generated by the `asctime` function, and the first character of `dd` is a space character if the value is less than 10). If the date of translation is not available, an implementation-defined valid date shall be supplied.

`__TIME__` The time of translation of the source file (a character string literal of the form `"hh:mm:ss"` as in the time generated by the `asctime` function). If the time of translation is not available, an implementation-defined valid time shall be supplied.

`__STDC__` Whether `__STDC__` is defined and if so, what its value is, are implementation-defined.

`__cplusplus` The name `__cplusplus` is defined (to an unspecified value) when compiling a C++ translation unit.

2 The values of the predefined macros (except for `__LINE__` and `__FILE__`) remain constant throughout the translation unit.

3 None of these macro names, nor the identifier `defined`, shall be the subject of a `#define` or a `#undef` preprocessing directive. All predefined macro names shall begin with a leading underscore followed by an uppercase letter or a second underscore.

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 (15).

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 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. Subclause 17.3 and Clauses 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. Clauses 18 through 27 each describe a single category of entities within the library.

— **comparison function:** An operator function (13.5) 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 template `basic_string` and the non-member template functions that operate on strings can be referred to as the *string 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.

— **object state:** The current value of all nonstatic class members of an object (9.2). 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), and other editorial conventions (17.2.2).

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:[105)]

— Summary

— Requirements

[105)] 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.

— Detailed specifications

— 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 interface requires ‘‘for any object `x` of class `X`, `++x` is defined.’’ That is, whether the operator is a member 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 (20.1, 23.1, 24.1). Any generic algorithm (25) 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.[106)]

17.2.1.3 Specifications [lib.structure.specifications]

1 The detailed specifications each contain the following elements:[107)]

— Name and brief description

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

— Restrictions on template arguments, if any

106) Although in some cases the code given is unambiguously the optimum implementation.
107) The form of these specifications was designed to follow the conventions established by existing C++ library vendors.

— Decription of class invariants

— Description of function semantics

2 Descriptions of class member functions follow the order (as appropriate):[108)]

— 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):[109)]

— **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

— **Complexity:** the time and/or space complexity of the function

4 For non-reserved replacement and handler functions, Clause 18 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.

5 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), and member functions (17.2.2.2).

17.2.2.1 Type descriptions [lib.type.descriptions]

1 The Requirements subclauses describe names that are used to specify constraints on template arguments.[110)] These names are used in Clauses 23, 25, and 26 to describe the types that may be supplied as arguments by a C++ program when instantiating template components from the library.

108) 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.
109) 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.
110) Examples include: `InputIterator`, `ForwardIterator`, `Function`, `Predicate`, etc. See subclause 24.1.

2 Certain types defined in Clause 27 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.[111)]

2 The enumerated type `enumerated` can be written:

```
enum enumerated { V0, V1, V2, V3, .....};

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, as an integer type, or as a `bitset` (23.2.1).

2 The bitmask type `bitmask` can be written:

```
enum bitmask {
  V0 = 1 << 0, V1 = 1 << 1, V2 = 1 << 2, V3 = 1 << 3, .....
};

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.

111) Such as an integer type, with constant integer values (3.9.1).

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.[112)]

— 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 Clauses 18 through 27 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*)`,[113)] or by a change to a `locale` object, as described in Clauses 22.1 and 27.

— 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` (3.9.1), 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).[114)]

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.[115)]

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.[116)]

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.

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.[117)]

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.

112) Note that this definition differs from the definition in ISO C subclause 7.1.1.
113) declared in `<clocale>` (22.3).
114) 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.
115) A string literal, such as `"abc"`, is a static NTBS.
116) 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.
117) Many of the objects manipulated by function signatures declared in `<cwchar>` are wide-character sequences or NTWCSs.

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.[118)]

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.

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

1 Clauses 18 through 27 do not specify the representation of classes, and intentionally omit specification of class members (9.2). An implementation may define static or non-static class members, or both, as needed to implement the semantics of the member functions specified in Clauses 18 through 27.

2 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, subclauses 22.1.1, 23.2.1, 24.4.3, 24.4.4, 27.4.3, 27.7.1, and 27.8.1.1 provide 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 are enclosed in a comment that ends with ***exposition only***, as in:

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

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

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

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

— Functions

— Objects

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

3 The C++ Standard library provides 54 standard macros from the C library (C.4).

4 The C++ Standard library provides 45 standard values from the C library (C.4).

5 The C++ Standard library provides 19 standard types from the C library (C.4), and 28 additional types, as shown in Table 11:

Table 11—Standard Types

filebuf	ostringstream	wfilebuf	wstreambuf
ifstream	streambuf	wifstream	wstreampos
ios	streamoff	wios	wstring
istream	streampos	wistream	wstringbuf
istringstream	string	wistringstream	
new_handler	stringbuf	wofstream	
ofstream	terminate_handler	wostream	
ostream	unexpected_handler	wostringstream	

6 The C++ Standard library provides 66 standard template classes, as shown in Table 12:

Table 12—Standard Template classes

allocator	mask_array
auto_ptr	messages
back_insert_iterator	messages_byname
basic_filebuf	moneypunct
basic_ifstream	moneypunct_byname
basic_ios	money_get
basic_istream	money_put
basic_istringstream	multimap
basic_ofstream	multiset
basic_ostream	numeric_limits
basic_ostringstream	numpunct
basic_streambuf	num_get
basic_string	num_put
basic_stringbuf	ostreambuf_iterator
binary_negate	ostream_iterator

binder1st	pointer_to_binary_function
binder2nd	pointer_to_unary_function
bitset	priority_queue
codecvt	queue
codecvt_byname	raw_storage_iterator
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	

7 The C++ Standard library provides 24 standard template structures, as shown in Table 13:

Table 13—Standard Template structs

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

8 The C++ Standard library provides 86 standard template operator functions, as shown in Table 14.

9 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 14—Standard Template operators

operator!= (basic_string) [5]	operator<< (basic_string)
operator!= (complex) [3]	operator<< (bitset)
operator!= (istreambuf_iterator)	operator<< (complex)
operator!= (ostreambuf_iterator)	operator<< (valarray) [3]
operator!= (T)	operator<<=(valarray) [2]
operator!= (valarray) [3]	operator<= (T)
operator% (valarray) [3]	operator<= (valarray) [3]
operator%= (valarray) [2]	operator== (basic_string) [5]
operator& (bitset)	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>> (bitset)
operator< (list)	operator>> (complex)
operator< (map)	operator>> (valarray) [3]
operator< (multimap)	operator>>=(valarray) [2]
operator< (multiset)	operator^ (bitset)
operator< (pair)	operator^ (valarray) [3]
operator< (queue)	operator^= (valarray) [2]
operator< (restrictor)	operator| (bitset)
operator< (reverse_iterator)	operator| (valarray) [3]
operator< (set)	operator|= (valarray) [2]
operator< (stack)	operator|| (valarray) [3]
operator< (valarray) [3]	
operator< (vector)	

10 The C++ Standard library provides 144 standard template functions, as shown in Table 15:

Table 15—Standard Template functions

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

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

11 The C++ Standard library provides 28 standard classes, as shown in Table 16.

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

Table 16—Standard Classes

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

13 The C++ Standard library provides 2 standard structures from the C library (C.4), and 16 additional structures, as shown in Table 17:

Table 17—Standard Structs

`bidirectional_iterator_tag`	`nothrow`
`codecvt_base`	`output_iterator`
`ctype_base`	`output_iterator_tag`
`forward_iterator_tag`	`random_access_iterator_tag`
`input_iterator_tag`	`string_char_traits<char>`
`ios_traits<char>`	`string_char_traits<wchar_t>`
`ios_traits<wchar_t>`	`time_base`
`money_base`	
`money_base::pattern`	

14 The C++ Standard library provides 12 standard operator functions, as shown in Table 18:

Table 18—Standard Operator functions

`operator delete`	`operator new[] (void*)`
`operator delete[]`	`operator< (vector<bool,allocator>)`
`operator new`	`operator<< (locale)`
`operator new (nothrow)`	`operator== (vector<bool,allocator>)`
`operator new (void*)`	`operator>> (locale)`
`operator new[]`	
`operator new[] (nothrow)`	

15 The C++ Standard library provides 208 standard functions from the C library (C.4), and 78 additional functions, as shown in Table 19:

Table 19—Standard Functions

`abs  (float)`	`mod  (long double)`
`abs  (long double)`	`modf (float,float*)`
`abs  (long)`	`modf (long double,long double*)`
`acos (float)`	`noshowbase`
`acos (long double)`	`noshowpoint`
`asin (float)`	`noshowpos`
`asin (long double)`	`noskipws`
`atan (float)`	`nouppercase`
`atan (long double)`	`oct`
`atan2(float,float)`	`pow  (float) [2]`
`atan2(long double,long double)`	`pow  (long double) [2]`
`ceil (float)`	`resetiosflags`
`ceil (long double)`	`right`
`cos  (float)`	`scientific`
`cos  (long double)`	`setbase`
`cosh (float)`	`setfill`
`cosh (long double)`	`setiosflags`
`dec`	`setprecision`
`div  (long,long)`	`setw`
`endl`	`set_new_handler`
`ends`	`set_terminate`
`exp  (float)`	`set_unexpected`
`exp  (long double)`	`showbase`
`fixed`	`showpoint`
`floor(float)`	`showpos`
`floor(long double)`	`sin  (float)`
`flush`	`sin  (long double)`
`frexp(float,int*)`	`sinh (float)`
`frexp(long double,int*)`	`sinh (long double)`
`hex`	`skipws`
`internal`	`tan  (float)`
`iterator_category`	`tan  (long double)`
`ldexp(float,int)`	`tanh (float)`
`ldexp(long double,int)`	`tanh (long double)`
`left`	`terminate`
`log  (float)`	`unexpected`
`log  (long double)`	`uppercase`
`log10(float)`	`ws`
`log10(long double)`	
`mod  (float)`	

16 The C++ Standard library provides 8 standard objects, as shown in Table 20:

Table 20—Standard Objects

`cerr`	`cin`	`clog`	`cout`
`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 *header*.[119)]

2 The C++ Standard library provides 32 *C++ headers*, as shown in Table 21:

Table 21—C++ Library Headers

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

3 The facilities of the Standard C Library are provided in 18 additional headers, as shown in Table 22:

Table 22—C++ Headers for C Library Facilities

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

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 7), 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 D.1, Standard C library headers, describes the effects of using the *name*`.h` (C header) form in a C++ program.[120)]

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 describes the set of available headers.

2 A freestanding implementation has has an implementation-defined set of headers. This set shall include at least the following headers, as shown in Table 23:

119) 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).

120) The `".h"` headers dump all their names into the global namespace, whereas the newer forms keep their names in namespace `std`. Therefore, the newer forms are the preferred forms for all uses except for C++ programs which are intended to be strictly compatible with C.

Table 23—C++ Headers for Freestanding Implementations

Subclause	Header(s)
18.1 Types	`<cstddef>`
18.2 Implementation properties	`<limits>`
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>`

3 The supplied version of the header `<cstdlib>` shall declare at least the functions `abort()`, `atexit()`, and `exit()` (18.3).

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`.[121)]

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 It is unspecified whether a name from the Standard C library declared with external linkage has either `extern "C"` or `extern "C++"` linkage.[122)]

3 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).

121) This is the same as the Standard C library.
122) 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.

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 A C++ program shall not extend the namespace `std`.

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

— Macros

— Global names

— Names with external linkage

3 If the program declares or defines a name in a context where it is reserved, other than as explicitly allowed by this clause, the behavior is undefined.

17.3.3.1.1 Macro names [lib.macro.names]

1 Each name defined as a macro in a header is reserved to the implementation for any use if the translation unit includes the header.[123)]

2 A translation unit that includes a header shall not contain any macros that define names declared or defined in that header. Nor shall such a translation unit define macros for names lexically identical to keywords.

17.3.3.1.2 Global names [lib.global.names]

1 Each header also optionally declares or defines names which are always reserved to the implementation for any use and names reserved to the implementation for use at file scope.

2 Certain sets of names and function signatures are reserved whether or not a translation unit includes a header:

3 Each name that begins with an underscore and either an uppercase letter or another underscore (2.8) is reserved to the implementation for any use.

4 Each name that begins with an underscore is reserved to the implementation for use as a name with file scope or within the namespace `std` in the ordinary name space.

17.3.3.1.3 External linkage [lib.extern.names]

1 Each name declared as an object with external linkage in a header is reserved to the implementation to designate that library object with external linkage.[124)]

2 Each global function signature declared with external linkage in a header is reserved to the implementation to designate that function signature with external linkage.[125)]

3 Each name having two consecutive underscores (2.8) is reserved to the implementation for use as a name with both `extern "C"` and `extern "C++"` linkage.

4 Each name from the Standard C library declared with external linkage is reserved to the implementation for use as a name with `extern "C"` linkage.

123) It is not permissible to remove a library macro definition by using the `#undef` directive.
124) The list of such reserved names includes `errno`, declared or defined in `<cerrno>`.
125) 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>`.

5 Each function signature from the Standard C library declared with external linkage is reserved to the implementation for use as a function signature with both `extern "C"` and `extern "C++"` linkage.[126)]

17.3.3.2 Headers [lib.alt.headers]

1 If a file has a name equivalent to the derived file name for one of the C++ Standard library headers, is not provided as part of the implementation, and is placed in any of the standard places for a source file to be included (16.2), the behavior is undefined.

17.3.3.3 Derived classes [lib.derived.classes]

1 Virtual member function signatures defined for a base class in the C++ Standard library may be overridden in a derived class defined in the program (10.3).

17.3.3.4 Replacement functions [lib.replacement.functions]

1 Clauses 18 through 27 describe the behavior of numerous functions defined by the C++ Standard library. Under some circumstances, however, certain of these function descriptions also apply to replacement functions defined in the program (17.1).

2 A C++ program may provide the definition for any of six (6) dynamic memory allocation function signatures declared in header `<new>` (3.7.3, 18):[127)]

— `operator new(size_t)`

— `operator new(size_t,nothrow)`

— `operator new[](size_t)`

— `operator new[](size_t,nothrow)`

— `operator delete(void*)`

— `operator delete[](void*)`

3 The program's definitions are used instead of the default versions supplied by the implementation (8.4). Such replacement occurs prior to program startup (3.2, 3.6).

17.3.3.5 Handler functions [lib.handler.functions]

1 The C++ Standard library provides default versions of the three handler functions (18):

— *new_handler*

— *unexpected_handler*

— *terminate_handler*

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

— `set_new_handler`

— `set_unexpected`

— `set_terminate`

SEE ALSO: subclauses 18.4.2, Storage allocation errors, and 18.6, Exception handling.

[126)] 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.

17.3.3.6 Other functions [lib.res.on.functions]

1 In certain cases (replacement functions, handler functions, operations on types used to instantiate standard library template components), the C++ Standard library depends on components supplied by a C++ program. If these components do not meet their requirements, the Standard places no requirements on the implementation.

2 In particular, the effects are undefined in the following cases:

— for replacement functions (18.4.1), if the installed handler function does not implement the semantics of the applicable **Required behavior** paragraph.

— for handler functions (18.4.2.2, 18.6.2.1, 18.6.1.2), if the installed handler function does not implement the semantics of the applicable **Required behavior** paragraph

— for types used as template arguments when instantiating a template component, if the operations on the type do not implement the semantics of the applicable **Requirements** subclause (20.1, 23.1, 24.1, 26.1).

— if any of these functions or operations throws an exception, unless specifically allowed in the applicable **Required behavior** paragraph.

17.3.3.7 Function arguments [lib.res.on.arguments]

1 Each of the following statements applies to all arguments to functions defined in the C++ Standard library, unless explicitly stated otherwise.

— If an argument to a function has an invalid value (such as a value outside the domain of the function, or a pointer invalid for its intended use), the behavior is undefined.

— If a function argument is described as being an array, the pointer actually passed to the function shall have a value such that all address computations and accesses to objects (that would be valid if the pointer did point to the first element of such an array) are in fact valid.

17.3.4 Conforming implementations [lib.conforming]

1 This subclause describes the constraints upon, and latitude of, implementations of the C++ Standard library. The following subclauses describe an implementation's use of headers (17.3.4.1), macros (17.3.4.2), global functions (17.3.4.3), member functions (17.3.4.4), reentrancy (17.3.4.5), access specifiers (17.3.4.6), class derivation (17.3.4.7), and exceptions (17.3.4.8).

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 (3.2).

2 Header inclusion is limited as follows:

— 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.[127)]

— The C headers (`.h` form, described in Annex D, D.1) shall include only their corresponding C++ header, as described above (17.3.1.2).

— The C++ headers listed in Table 21, C++ Library Headers, shall include the header(s) listed in their respective **Synopsis** subclause (18.4, 18.5, 18.6, 19.1, 20.2, 20.3, 20.4, 21.1, 22.1, 23.2, 24, 25, 26.2, 26.3, 27.3, 27.4, 27.5, 27.6, 27.7, 27.8.1).[128)]

127) 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.

128) C++ headers must include a C++ header that contains any needed definition (3.2).

3 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.[129)]

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

1 The names or global function signatures described in subclause 17.3.1.1 are reserved to the implementation.[130)]

2 All object-like macros defined by the Standard C 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 It is unspecified whether any global functions in the C++ Standard library are defined as `inline` (7.1.2).

2 A call to a global function signature described in Clauses 18 through 27 behaves the same as if the implementation declares no additional global function signatures.[131)]

17.3.4.4 Member functions [lib.member.functions]

1 It is unspecified whether any member functions in the C++ Standard library are defined as `inline` (7.1.2).

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

— by adding arguments with default values to a member function signature;[132)] 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.

3 A call to a member function signature described in the C++ Standard library behaves the same as if the implementation declares no additional member function signatures.[133)]

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 function signature or class described in Clauses 18 through 27 is a `friend` of another class in the C++ Standard Library.

2 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. A member described as public is always public.

129) 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.

130) 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`.

131) 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.

132) Hence, taking the address of a member function has an unspecified type.

133) 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.

17.3.4.7 Derived classes [lib.derivation]

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

— It is unspecified whether a class in the C++ Standard Library 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 the C++ Standard Library 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` is always virtual;

— A base class described as non-`virtual` is never virtual;

— Unless explicitly stated otherwise, types with distinct names are distinct types.[134)]

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 by throwing an exception of the type(s) described in their **Throws:** paragraph and/or their *exception-specification* (15.4).

2 Any of the functions defined in the C++ Standard library that do not have an *exception-specification* prohibiting it, can report a failure to allocate storage by throwing an exception of type `bad_alloc`, or a class derived from `bad_alloc` (18.4.2.1).

134) An implicit exception to this rule are types described as synonyms for basic integral types, such as `size_t` (18.1) and `streamoff` (27.4.1).

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 24:

Table 24—Language support library summary

Subclause	Header(s)
18.1 Types	`<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 `<cstddef>` (Table 25):

Table 25—Header `<cstddef>` synopsis

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

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

4 The macro `NULL` is an implementation-defined C++ null-pointer constant in this International Standard (4.10).[135)]

135) Possible definitions include `0` and `0L`, but not `(void*)0`.

5 The macro `offsetof` accepts a restricted set of *`type`* arguments in this International Standard. *`type`* shall be a POD structure or a POD union (9).

SEE ALSO: subclause 5.3.3, Sizeof, subclause 5.7, Additive operators, subclause 12.5, Free store, and ISO C subclause 7.1.6.

18.2 Implementation properties [lib.support.limits]

1 Characteristics of implementation-dependent fundamental types (3.9.1).

18.2.1 Numeric limits [lib.limits]

1 The `numeric_limits` component provides a C++ program with information about various properties of the implementation's representation of the fundamental types.

2 Specializations shall be provided for each fundamental type, both floating point and integer, including `bool`. The member `is_specialized` shall be `true` for all such specializations of `numeric_limits`.

3 Non-scalar types, such as `complex<T>` (26.2.1), shall not have specializations.

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<signed char>;
  class numeric_limits<unsigned char>;
  class numeric_limits<wchar_t>;

  class numeric_limits<short>;
  class numeric_limits<int>;
  class numeric_limits<long>;
  class numeric_limits<unsigned short>;
  class numeric_limits<unsigned int>;
  class numeric_limits<unsigned long>;

  class numeric_limits<float>;
  class numeric_limits<double>;
  class numeric_limits<long double>;
}
```

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

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

    static const bool has_infinity;
    static const bool has_quiet_NaN;
    static const bool has_signaling_NaN;
    static const bool has_denorm;
    static T infinity();
    static T quiet_NaN();
    static T signaling_NaN();
    static T denorm_min();

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

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

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.[136)]

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

136) Equivalent to CHAR_MIN, SHRT_MIN, FLT_MIN, DBL_MIN, etc.

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

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

4 Maximum finite value.[137)]

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

```
static const int  digits;
```

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

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

8 For floating point types, the number of `radix` digits in the mantissa.[138)]

```
static const int  digits10;
```

9 Number of base 10 digits which can be represented without change.[139)]

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).[140)]

18 For integer types, specifies the base of the representation.[141)]

137) Equivalent to CHAR_MAX, SHRT_MAX, FLT_MAX, DBL_MAX, etc.
138) Equivalent to FLT_MANT_DIG, DBL_MANT_DIG, LDBL_MANT_DIG.
139) Equivalent to FLT_DIG, DBL_DIG, LDBL_DIG.
140) Equivalent to FLT_RADIX.
141) Distinguishes types with bases other than 2 (e.g. BCD).

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.[142)]

21 Meaningful only for floating point types.

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

22 Measure of the maximum rounding error.[143)]

```
static const int  min_exponent;
```

23 Minimum negative integer such that `radix` raised to that power is in range.[144)]

24 Meaningful only for floating point types.

```
static const int  min_exponent10;
```

25 Minimum negative integer such that 10 raised to that power is in range.[145)]

26 Meaningful only for floating point types.

```
static const int  max_exponent;
```

27 Maximum positive integer such that `radix` raised to that power is in range.[146)]

28 Meaningful only for floating point types.

```
static const int  max_exponent10;
```

29 Maximum positive integer such that 10 raised to that power is in range.[147)]

30 Meaningful only for floating point types.

```
static const bool has_infinity;
```

31 True if the type has a representation for positive infinity.

32 Meaningful only for floating point types.

33 Shall be `true` for all specializations in which `is_iec559 == true`.

```
static const bool has_quiet_NaN;
```

142) Equivalent to FLT_EPSILON, DBL_EPSILON, LDBL_EPSILON.
143) This has a precise definition in the Language Independent Arithmetic (LIA-1) standard. Required by LIA-1.
144) Equivalent to FLT_MIN_EXP, DBL_MIN_EXP, LDBL_MIN_EXP.
145) Equivalent to FLT_MIN_10_EXP, DBL_MIN_10_EXP, LDBL_MIN_10_EXP.
146) Equivalent to FLT_MAX_EXP, DBL_MAX_EXP, LDBL_MAX_EXP.
147) Equivalent to FLT_MAX_10_EXP, DBL_MAX_10_EXP, LDBL_MAX_10_EXP.

34 True if the type has a representation for a quiet (non-signaling) ‘‘Not a Number.’’[148)]

35 Meaningful only for floating point types.

36 Shall be `true` for all specializations in which `is_iec559 == true`.

```
static const bool has_signaling_NaN;
```

37 True if the type has a representation for a signaling ‘‘Not a Number.’’[149)]

38 Meaningful only for floating point types.

39 Shall be `true` for all specializations in which `is_iec559 == true`.

```
static const bool has_denorm;
```

40 True if the type allows denormalized values (variable number of exponent bits).[150)]

41 Meaningful only for flotaing point types.

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

42 Representation of positive infinity, if available.[151)]

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

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

44 Representation of a quiet ‘‘Not a Number,’’ if available.[152)]

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.[153)]

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 positive denormalized value.[154)]

49 Meaningful for all floating point types.

148) Required by LIA-1.
149) Required by LIA-1.
150) Required by LIA-1.
151) Required by LIA-1.
152) Required by LIA-1.
153) Required by LIA-1.
154) Required by LIA-1.

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

```
static const bool is_iec559;
```

51 True if and only if the type adheres to IEC 559 standard.[155)]

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.[156)] 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.[157)] 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.

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.[158)]

59 Meaningful for all specializations.

```
static const bool tinyness_before;
```

60 `true` if tinyness is detected before rounding.[159)]

61 Meaningful only for floating point types.

```
static const float_round_style round_style;
```

62 The rounding style for the type.[160)]

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

155) International Electrotechnical Commission standard 559 is the same as IEEE 754.
156) Required by LIA-1.
157) Required by LIA-1.
158) Required by LIA-1.
159) Refer to IEC 559. Required by LIA-1.
160) Equivalent to FLT_ROUNDS. Required by LIA-1.

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 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’’ shall be set to 0 or `false`.

2 [*Example:*

```
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; }

    static const int min_exponent   = -125;
    static const int min_exponent10 = - 37;
    static const int max_exponent   = +128;
    static const int max_exponent10 = + 38;

    static const bool has_infinity      = true;
    static const bool has_quiet_NaN     = true;
    static const bool has_signaling_NaN = true;
    static const bool has_denorm        = false;

    inline static float infinity()      { return ...; }
    inline static float quiet_NaN()     { return ...; }
    inline static float signaling_NaN() { return ...; }
    inline static float denorm_min()    { return min(); }
    static const bool is_iec559  = true;
    static const bool is_bounded = true;
    static const bool is_modulo  = false;
    static const bool traps      = true;
    static const bool tinyness_before = true;

    static const float_round_style round_style = round_to_nearest;
  };
}
```

—end example]

18.2.2 C Library [lib.c.limits]

1 Header `<climits>` (Table 26):

Table 26—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 27):

Table 27—Header `<cfloat>` synopsis

Type	Name(s)		
Values:			
DBL_DIG	DBL_MIN_EXP	FLT_MIN_10_EXP	LDBL_MAX_10_EXP
DBL_EPSILON	FLT_DIG	FLT_MIN_EXP	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_EXP
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 28:

Table 28—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:

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

3 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 (18.6.2.3).

`exit(int `*`status`*`)`

4 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.[161)]

— Next, all static objects are destroyed in the reverse order of their construction. (Automatic objects are not destroyed as a result of calling `exit()`.)[162)]

— 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.[163)]

— 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.[164)]

5 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

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

162) 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()`.

163) Any C streams associated with `cin`, `cout`, etc (27.3) are flushed and closed when static objects are destroyed in the previous phase. The function `tmpfile()` is declared in `<cstdio>`.

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

```
#include <cstdlib>      // for size_t
#include <stdexcept>    // for exception

namespace std {
  void* operator new(size_t size) throw(bad_alloc);
  struct nothrow {};
  void* operator new(size_t size, const nothrow&) throw();
  void  operator delete(void* ptr) throw();
  void* operator new[](size_t size) throw(bad_alloc);
  void* operator new[](size_t size, const nothrow&) throw();
  void  operator delete[](void* ptr) throw();

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

  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.

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

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

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

Effects: 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.

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), or else throw a `bad_alloc` exception.

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, throw `bad_alloc`.

— Otherwise, the function calls the current *new_handler* (18.4.2.2). 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.

`void* operator new(size_t size, const nothrow&) throw();`

Effects: Same as above, except that it is called by a placement version of a *new-expression* when a C++ program prefers a null pointer result as an error indication, instead of a `bad_alloc` exception.

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), or else return a null pointer.
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, return a null pointer.

— Otherwise, the function calls the current *new_handler* (18.4.2.2). 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. If the called *new_handler* function terminates by throwing a `bad_alloc` exception, the function returns a null pointer.

1 [*Example:*

```
T* p1 = new T;              // throws bad_alloc if it fails
T* p2 = new(nothrow()) T;   // returns 0 if it fails
```

—*end example*]

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

Effects: The *deallocation function* (3.7.3.2) called by a *delete-expression* to render the value of `ptr` invalid.
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 the default `operator new(size_t)` or `operator new(size_t,const nothrow&)`.
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 `operator new`.[165] For such a non-null value of `ptr`, reclaims storage allocated by the earlier call to the default `operator new`.

Notes: It is unspecified under what conditions part or all of such reclaimed storage is allocated by a subsequent call to `operator new` or any of `calloc`, `malloc`, or `realloc`, declared in `<cstdlib>`.

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

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

Effects: 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.[166]
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)`.

[165] The value must not have been invalidated by an intervening call to `operator delete(void*)` (17.3.3.7).
[166] 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.

`void* operator new[](size_t size, const nothrow&) throw();`

Effects: Same as above, except that it is called by a placement version of a *new-expression* when a C++ program prefers a null pointer result as an error indication, instead of a `bad_alloc` exception.
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,const nothrow&)`.
Default behavior: Returns `operator new(size,nothrow())`.

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

Effects: The *deallocation function* (3.7.3.2) called by the array form of a *delete-expression* to render the value of `ptr` invalid.
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 `operator new[](size_t)`.[167)] For such a non-null value of `ptr`, reclaims storage allocated by the earlier call to the default `operator new[](size_t)` or `operator new[](size_t,nothrow)`.

1 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>`.

18.4.1.3 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 (17.3.3).

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

Returns: `ptr`.
Notes: Intentionally performs no other action.[168)]

2 [*Example:* This can be useful for constructing an object at a known address:

```
char place[sizeof(Something)];
Something* p = new (place) Something();
```

—*end example*]

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

Returns: `ptr`.
Notes: Intentionally performs no other action.

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

Effects: Intentionally performs no action.

167) The value must not have been invalidated by an intervening call to `operator delete[](void*)` (17.3.3.7).

Notes: Default function called for a placement delete expression. Complements default placement `new`.

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

Effects: Intentionally performs no action.
Notes: Default function called for a placement array delete expression. Complements default placement `new[]`.

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

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

```
bad_alloc() throw();
```

Effects: Constructs an object of class `bad_alloc`.

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

Effects: Copies an object of class `bad_alloc`.
Notes: The result of calling `what()` on the newly constructed object is implementation-defined.

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

Returns: An implementation-defined value.

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[]()` (18.4.1) 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()`;
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`*.
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 exception

namespace std {
  class type_info;
  class bad_cast;
  class bad_typeid;
}
```

SEE ALSO: subclauses 5.2.6, 5.2.7.

18.5.1 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);
  };
}
```

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.

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

Effects: Compares the current object with *`rhs`*.
Returns: `true` if the two values describe the same type.

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

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

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

Effects: Compares the current object with *`rhs`*.

Returns: `true` if `*this` precedes *`rhs`* in the implementation's collation order.

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

Returns: an implementation-defined value.
Notes: The message may be a null-terminated multibyte string (17.2.2.1.3.2), suitable for conversion and display as a `wstring` (21.1.4, 22.2.1.4)

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

Effects: Copies a `type_info` object.
Notes: Since the copy constructor and assignment operator for `type_info` are private to the class, objects of this type cannot be copied.

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

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

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 (5.2.6).

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

Effects: Constructs an object of class `bad_cast`.

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

Effects: Copies an object of class `bad_cast`.
Notes: The result of calling `what()` on the newly constructed object is implementation-defined.

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

Returns: An implementation-defined value.
Notes: The message may be a null-terminated multibyte string (17.2.2.1.3.2), suitable for conversion and display as a `wstring` (21.1.4, 22.2.1.4)

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

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

1 The class `bad_typeid` defines the type of objects thrown as exceptions by the implementation to report a null pointer in a *typeid* expression (5.2.7).

```
bad_typeid() throw();
```

Effects: Constructs an object of class `bad_typeid`.

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

Effects: Copies an object of class `bad_typeid`.
Notes: The result of calling `what()` on the newly constructed object is implementation-defined.

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

Returns: An implementation-defined value.
Notes: The message may be a null-terminated multibyte string (17.2.2.1.3.2), suitable for conversion and display as a `wstring` (21.1.4, 22.2.1.4)

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

```
#include <stdexcept>    // for exception

namespace std {
  class bad_exception;

  typedef void (*unexpected_handler)();
  unexpected_handler set_unexpected(unexpected_handler f);
  void unexpected();

  typedef void (*terminate_handler)();
  terminate_handler set_terminate(terminate_handler f);
  void terminate();
}
```

SEE ALSO: subclause 15.5.

18.6.1 Violating *`exception-specifications`* [lib.exception.unexpected]

18.6.1.1 Class `bad_exception` [lib.bad.exception]

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

1 The class `bad_exception` defines the type of objects thrown as exceptions by the implementation to report a violation of an *exception-specification* (15.5.2).

```
bad_exception() throw();
```

Effects: Constructs an object of class `bad_exception`.

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

Effects: Copies an object of class `bad_exception`.
Notes: The result of calling `what()` on the newly constructed object is implementation-defined.

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

Returns: An implementation-defined value.
Notes: The message may be a null-terminated multibyte string (17.2.2.1.3.2), suitable for conversion and display as a `wstring` (21.1.4, 22.2.1.4)

18.6.1.2 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 a `bad_exception` exception;

— call `terminate()`;

— call either `abort()` or `exit()`;

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

18.6.1.3 `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`*.

18.6.1.4 `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 (18.6.1.2).

18.6.2 Abnormal termination [lib.exception.terminate]

18.6.2.1 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()`.

18.6.2.2 `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`*.

18.6.2.3 `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 (18.6.2.1).

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 28—Header `<cstdarg>` synopsis

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

Table 28—Header `<csetjmp>` synopsis

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

Table 28—Header `<ctime>` synopsis

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

Table 28—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 28—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 29:

Table 29—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() throw();
    exception& exception(const exception&) throw();
    exception& operator=(const exception&) throw();
    virtual ~exception() throw();
    virtual const char* what() const throw();
  };
}
```

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.

```
exception() throw();
```

Effects: Constructs an object of class `exception`.
Notes: Does not throw any exceptions.

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

Effects: Copies an `exception` object.
Notes: The effects of calling `what()` after assignment are implementation-defined.

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

Effects: Destroys an object of class `exception`.
Notes: Does not throw any exceptions.

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

Returns: An implementation-defined NTBS.
Notes: The message may be a null-terminated multibyte string (17.2.2.1.3.2), suitable for conversion and display as a `wstring` (21.1.4, 22.2.1.4)

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

```
namespace std {
  class logic_error : public exception {
  public:
    logic_error(const string& what_arg);
  };
}
```

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

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

Effects: Constructs an object of class `logic_error`.
Postcondition: `what() == what_arg.data()`.

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

```
namespace std {
  class domain_error : public logic_error {
  public:
    domain_error(const string& what_arg);
  };
}
```

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

`domain_error(const string& what_arg);`

Effects: Constructs an object of class `domain_error`.
Postcondition: `what() == what_arg.data()`.

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

```
namespace std {
  class invalid_argument : public logic_error {
  public:
    invalid_argument(const string& what_arg);
  };
}
```

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

`invalid_argument(const string& what_arg);`

Effects: Constructs an object of class `invalid_argument`.
Postcondition: `what() == what_arg.data()`.

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

```
namespace std {
  class length_error : public logic_error {
  public:
    length_error(const string& what_arg);
  };
}
```

1 The class `length_error` defines the type of objects thrown as exceptions to report an attempt to produce an object whose length equals or exceeds its maximum allowable size.

`length_error(const string& what_arg);`

Effects: Constructs an object of class `length_error`.
Postcondition: `what() == what_arg.data()`.

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);
  };
}
```

1 The class `out_of_range` defines the type of objects thrown as exceptions to report an argument value not in its expected range.

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

Effects: Constructs an object of class `out_of_range`.
Postcondition: `what() == what_arg.data()`.

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

```
namespace std {
  class runtime_error : public exception {
  public:
    runtime_error(const string& what_arg);
  };
}
```

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

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

Effects: Constructs an object of class `runtime_error`.
Postcondition: `what() == what_arg.data()`.

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

```
namespae std {
  class range_error : public runtime_error {
  public:
    range_error(const string& what_arg);
  };
}
```

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

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

Effects: Constructs an object of class `range_error`.
Postcondition: `what() == what_arg.data()`.

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

```
namespace std {
  class overflow_error : public runtime_error {
  public:
    overflow_error(const string& what_arg);
  };
}
```

1 The class `overflow_error` defines the type of objects thrown as exceptions to report an arithmetic overflow error.

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

Effects: Constructs an object of class `overflow_error`.

Postcondition: `what() == ` *`what_arg`*`.data()`.

19.2 Assertions [lib.assertions]

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

2 Header `<cassert>` (Table 30):

Table 30—Header `<cassert>` synopsis

Type	Name(s)
Macro:	`assert`

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 31):

Table 31—Header `<cerrno>` synopsis

Type	Name(s)		
Macros:	`EDOM`	`ERANGE`	`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 allocator requirements, utility components, function objects, dynamic memory management utilities, and date/time utilities, as summarized in Table 32:

Table 32—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 The library describes a standard set of requirements for *allocators*, which are objects that 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. All of the containers (23) are parameterized in terms of allocators.

2 In the following Table 33, `X` denotes an allocator class for objects of type `T`, `a` denotes a value of `X`, `n` denotes an instance of type `X::size_type`, `p` denotes an instance of type `X::pointer` which was obtained from `X`.

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

Table 33—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 `reference`.
`X::const_pointer`	pointer to `const T` type	the result of `operator*` of values of `X::const_pointer` is of `const reference`; 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.max_size()`	`X::size_type`	the largest positive value of `X::difference_type`.

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, pairs:
  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&);
}
```

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 Pairs [lib.pairs]

1 The library provides a template for heterogenous pairs of values. The library also provides a matching template function to simplify their construction.

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

2 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)`.

3 [*Example:* Instead of writing,

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

a C++ program may write:

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

—*end example*]

20.3 Function objects [lib.function.objects]

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 (25), 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.

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));
}
```

2 Using function objects together with function templates increases the expressive power of the library as well as making the resulting code much more efficient.

3 [*Example:* If a C++ program wants to have a by-element addition of two vectors `a` and `b` containing `double` and put the result into `a`, it can do:

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

—*end example*]

4 [*Example:* To negate every element of `a`:

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

The corresponding functions will inline the addition and the negation. —*end example*]

5 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 {
    typedef Arg    argument_type;
    typedef Result result_type;
  };

  template <class Arg1, class Arg2, class Result>
  struct binary_function {
    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 (5.6, 5.7).

```
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 (5.9, 5.10).

```
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]

1 The library provides basic function object classes for all of the logical operators in the language (5.14, 5.15, 5.3.1).

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

2 `operator()` returns *x* `&&` *y*.

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

3 `operator()` returns *x* `||` *y*.

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

4 `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 (5.3.1).

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

Returns: `!pred(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> {
  public:
    explicit binary_negate(const Predicate& pred);
    bool operator()(const first_argument_type&  x,
                    const second_argument_type& y) const;
  };
```

2 `operator()` returns `!pred(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 [*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. —*end example*]

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> {
  public:
    explicit pointer_to_unary_function(Result (*f)(Arg));
    Result operator()(const Arg& x) const;
  };
```

2 `operator()` returns *f*(*x*).

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

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

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

3 `operator()` returns *f*(*x*,*y*).

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

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

4 [*Example:*

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

replaces each `C` with `C++` in sequence `v`.[168) —*end example*]

20.4 Memory [lib.memory]

Header `<memory>` synopsis

```
#include <cstddef>      // for size_t, ptrdiff_t
#include <iterator>     // for output_iterator
#include <utility>      // for pair

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;
```

168) Implementations that have multiple pointer to function types shall provide additional `ptr_fun` template functions.

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

// 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;
}
```

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

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

  class allocator::types<void> {  // specialization
  public:
    typedef void* pointer;
    typedef void  value_type;
  };
  void* operator new(size_t N, allocator& a);
}
```

1 The members `allocate()` and `deallocate()` are parameterized to allow them to be specialized for particular types in user allocators.[169)]

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

Returns: `&x`.

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

Returns: `&x`.

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

Notes: Uses `::operator new(size_t)` (18.4.1).
Returns: `new T`, if `n == 1`. Returns `new T[n]`, if `n > 1`.

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

Requires: *p* shall be a pointer value obtained from `allocate()`.
Effects: Deallocates the storage referenced by *p*.
Notes: Uses `::operator delete(void*)` (18.4.1).

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

Returns:

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 [*Example:* 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.

169) In implementation is expected to provide allocators for all supported memory models.

```
#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() {}
};
```

—*end example*]

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 {
  public:
    explicit 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);
  };
}
```

`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)`.[170)]

170) 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.

20.4.3.2 `deallocate` [lib.deallocate]

1 Also, the following functions are provided:

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

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 Temporary buffers [lib.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)`).[171)]

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

Effects: Returns the buffer to which `p` points.
Requires: The buffer shall have been previously allocated by `get_temporary_buffer`.

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]

171) 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*);
```

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

1 Template `auto_ptr` holds onto a pointer obtained via `new` and deletes that object when it itself is destroyed (such as when leaving block scope 6.7).

```
namespace std {
  template<class X> class auto_ptr {
  public:
  // 20.4.5.1 construct/copy/destroy:
    explicit auto_ptr(X* p =0);
    auto_ptr(auto_ptr&);
    void operator=(auto_ptr&);
   ~auto_ptr();

  // 20.4.5.2 members:
    X& operator*() const;
    X* operator->() const;
    X* get() const;
    X* release();
    X* reset(X* p =0);
  };
}
```

2 The `auto_ptr` provides a semantics of strict ownership. An object may be safely pointed to by only one `auto_ptr`, so copying an `auto_ptr` copies the pointer and transfers ownership to the destination.

20.4.5.1 `auto_ptr` constructors [lib.auto.ptr.cons]

```
explicit 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: `get() ==` *`p`*

```
auto_ptr(auto_ptr& a);
```

Effects: copies the argument *`a`* to `*this`.
Calls *`a`*`.release()`.
Postcondition: `get()` `==` the value returned from *`a`*`.release()`.[172)]

```
void operator=(auto_ptr& a);
```

Effects: copies the argument *`a`* to `*this`.
Calls `reset(`*`a`*`.release())`.
Postcondition: `get()` `==` the value returned from *`a`*`.release()`.

```
~auto_ptr();
```

Effects: `delete get()`

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

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

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

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

Returns: The pointer *`p`* specified as the argument to the constructor `auto_ptr(X*` *`p`*`)` or as the argument to the most recent call to `reset(X*` *`p`*`)`.

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

Returns: `get()->m`

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

Postcondition: `get() == 0`

```
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 [lib.c.malloc]

1 Header `<cstdlib>` (Table 34):

172) That is, the value returned by *`a`*`.get()` before clearing it with *`a`*`.release()`.

Table 34—Header `<cstdlib>` synopsis

Type	Name(s)	
Functions:	`calloc`	`malloc`
	`free`	`realloc`

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

3 The functions `calloc()`, `malloc()`, and `realloc()` do not attempt to allocate storage by calling `::operator new()` (18.4).

4 The function `free()` does not attempt to deallocate storage by calling `::operator delete()`.

SEE ALSO: ISO C subclause 7.11.2.

5 Header `<cstring>` (Table 35):

Table 35—Header `<cstring>` synopsis

Type	Name(s)	
Macro:	`NULL`	
Type:	`size_t`	
Functions:	`memchr`	`memcmp`
`memcpy`	`memmove`	`memset`

6 The contents are the same as the Standard C library, with the change to `memchr()` specified in subclause 21.2.

SEE ALSO: ISO C subclause 7.11.2.

20.5 Date and time [lib.date.time]

1 Header `<ctime>` (Table 36):

Table 36—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 37:

Table 37—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 traits = string_char_traits<charT>,
           class Allocator = allocator> class basic_string;

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

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

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

template<class charT, class traits, class Allocator>
  bool operator<=(const basic_string<charT,traits,Allocator>& lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
template<class charT, class traits, class Allocator>
  bool operator<=(const basic_string<charT,traits,Allocator>& lhs,
                  const charT* rhs);
template<class charT, class traits, class Allocator>
  bool operator<=(const charT* lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
template<class charT, class traits, class Allocator>
  bool operator>=(const basic_string<charT,traits,Allocator>& lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
template<class charT, class traits, class Allocator>
  bool operator>=(const basic_string<charT,traits,Allocator>& lhs,
                  const charT* rhs);
template<class charT, class traits, class Allocator>
  bool operator>=(const charT* lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
  template<class charT, class traits, class Allocator>
    basic_istream<charT>&
      operator>>(basic_istream<charT>& is,
                 basic_string<charT,traits,Allocator>& str);
  template<class charT, class traits, class Allocator>
    basic_ostream<charT>&
      operator<<(basic_ostream<charT>& os,
                 const basic_string<charT,traits,Allocator>& str);
  template<class charT, class IS_traits, class STR_traits, class STR_Alloc>
    basic_istream<charT,IS_traits>&
      getline(basic_istream<charT,IS_traits>& is,
              basic_string<charT,STR_traits,STR_Alloc>& str,
              charT delim = IS_traits::newline() );

// 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

    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, char_type& a);
    static basic_ostream<charT>& char_out(basic_ostream<charT>& os, char_type a);
    static bool is_del(char_type a); // characteristic function for delimiters

    // 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.

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

Effects: Assigns *c2* to *c1*.

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

Returns `c1 == c2`

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

Returns: `!(c1 == c2)`

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

Returns: `c1 < c2`

`static char_type eos();`

Returns The null character, `char_type()`

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

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

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

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

`static bool is_del(char_type a);`

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

`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;
```

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

Effects:

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

`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 traits = string_char_traits<charT>,
           class Allocator = allocator>
  class basic_string {
  public:
  // types:
    typedef          traits            traits_type;
    typedef typename traits::char_type value_type;
    typedef typename Allocator::size_type       size_type;
    typedef typename Allocator::difference_type difference_type;

    typedef typename Allocator::types<charT>::reference       reference;
    typedef typename Allocator::types<charT>::const_reference const_reference;
    typedef typename Allocator::types<charT>::pointer         pointer;
    typedef typename Allocator::types<charT>::const_pointer   const_pointer;

    typedef typename Allocator::types<charT>::pointer         iterator;
    typedef typename Allocator::types<charT>::const_pointer   const_iterator;
    typedef reverse_iterator<iterator, value_type,
                    reference, difference_type>               reverse_iterator;
    typedef reverse_iterator<const_iterator, value_type,
                    const_reference, difference_type>         const_reverse_iterator;
    static const size_type npos = -1;

  // 21.1.1.4 construct/copy/destroy:
    explicit basic_string(Allocator& = Allocator());
    basic_string(const basic_string& str, size_type pos = 0,
                 size_type n = npos, Allocator& = Allocator());
    basic_string(const charT* s, size_type n, Allocator& = Allocator());
    basic_string(const charT* s, Allocator& = Allocator());
    basic_string(size_type n, charT c, Allocator& = Allocator());
    template<class InputIterator>
      basic_string(InputIterator begin, InputIterator end,
                   Allocator& = Allocator());
   ~basic_string();
    basic_string& operator=(const basic_string& str);
    basic_string& operator=(const charT* s);
    basic_string& operator=(charT c);
```

```
// 21.1.1.5 iterators:
  iterator       begin();
  const_iterator begin() const;
  iterator       end();
  const_iterator end() const;

  reverse_iterator       rbegin();
  const_reverse_iterator rbegin() const;
  reverse_iterator       rend();
  const_reverse_iterator rend() const;

// 21.1.1.6 capacity:
  size_type size() const;
  size_type length() const;
  size_type max_size() const;
  void resize(size_type n, charT c);
  void resize(size_type n);
  size_type capacity() const;
  void reserve(size_type res_arg);
  bool empty() const;

// 21.1.1.7 element access:
  charT     operator[](size_type pos) const;
  reference operator[](size_type pos);
  const_reference at(size_type n) const;
  reference       at(size_type n);

// 21.1.1.8 modifiers:
  basic_string& operator+=(const basic_string& rhs);
  basic_string& operator+=(const charT* 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 charT* s, size_type n);
  basic_string& append(const charT* s);
  basic_string& append(size_type n, charT c = charT());
  template<class InputIterator>
    basic_string& append(InputIterator first, InputIterator last);

  basic_string& assign(const basic_string& str, size_type pos = 0,
                       size_type n = npos);
  basic_string& assign(const charT* s, size_type n);
  basic_string& assign(const charT* s);
  basic_string& assign(size_type n, charT c = charT());
  template<class InputIterator>
    basic_string& assign(InputIterator first, InputIterator last);

  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 charT* s, size_type n);
  basic_string& insert(size_type pos, const charT* s);
  basic_string& insert(size_type pos, size_type n, charT c = charT() );
  iterator insert(iterator p, charT c = charT());
  iterator insert(iterator p, size_type n, charT c = charT());
  template<class InputIterator>
    void insert(iterator p, InputIterator first, InputIterator last);

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

```
    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 charT* s,
                          size_type n2);
    basic_string& replace(size_type pos, size_type n1, const charT* s);
    basic_string& replace(size_type pos, size_type n, charT c = charT());

    basic_string& replace(iterator i1, iterator i2, const basic_string& str);
    basic_string& replace(iterator i1, iterator i2, const charT* s, size_type n);
    basic_string& replace(iterator i1, iterator i2, const charT* s);
    basic_string& replace(iterator i1, iterator i2,
                          size_type n, charT c = charT());
    template<class InputIterator>
      basic_string& replace(iterator i1, iterator i2,
                            InputIterator j1, InputIterator j2);

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

  // 21.1.1.9 string operations:
    const charT* c_str() const;  // explicit
    const charT* data() const;

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

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

    size_type find_first_not_of(const basic_string& str,
                                size_type pos = 0) const;
    size_type find_first_not_of(const charT* s, size_type pos,
                                size_type n) const;
    size_type find_first_not_of(const charT* s, size_type pos = 0) const;
    size_type find_first_not_of(charT c, size_type pos = 0) const;
    size_type find_last_not_of (const basic_string& str,
                                size_type pos = npos) const;
    size_type find_last_not_of (const charT* s, size_type pos,
                                size_type n) const;
    size_type find_last_not_of (const charT* s, size_type pos = npos) const;
    size_type find_last_not_of (charT c, size_type pos = npos) const;
    basic_string substr(size_type pos = 0, size_type n = npos) const;
    int compare(const basic_string& str, size_type pos = 0,
                size_type n = npos) const;
    int compare(charT* s, size_type pos, size_type n) const;
    int compare(charT* s, size_type pos = 0) 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` (19.1.5);

— an *out-of-range* error is associated with exceptions of type `out_of_range` (19.1.6).

21.1.1.4 `basic_string` constructors [lib.string.cons]

1 In all `basic_string` constructors, a copy of the `Allocator` argument is used for any memory allocation performed by the constructor or member functions during the lifetime of the object.

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

Effects: Constructs an object of class `basic_string`. The postconditions of this function are indicated in Table 38:

Table 38—`basic_string()` effects

Element	Value
`data()`	an unspecified value
`size()`	0
`capacity()`	an unspecified value

```
basic_string(const basic_string<charT,traits,Allocator>& str,
             size_type pos = 0, size_type n = npos,
             Allocator& = Allocator());
```

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 39:

Table 39—`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 `size()`

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

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 40:

Table 40—`basic_string(const charT*,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 `size()`

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

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 41:

Table 41—`basic_string(const charT*)` 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 `size()`

Notes: Uses `traits::length()`.

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

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 42:

Table 42—`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 `size()`

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

Effects: Constructs a string from the values in the range `[begin, end)`, as indicated in Table 43:

Table 43—`basic_string(begin,end)` effects

Element	Value
`data()`	points at the first element of an allocated copy of the elements in the range [*first*, *last*)
`size()`	distance between *first* and *last*
`capacity()`	a value at least as large as `size()`

Notes: see Table ___, subclause _lib.sequence.requirements_.

```
basic_string<charT,traits,Allocator>&
  operator=(const basic_string<charT,traits,Allocator>& str);
```

Returns: `*this = basic_string<charT,traits,Allocator>(`*str*`)`.

```
basic_string<charT,traits,Allocator>&
  operator=(const charT* s);
```

Returns: `*this = basic_string<charT,traits,Allocator>(`*s*`)`.
Notes: Uses `traits::length()`.

```
basic_string<charT,traits,Allocator>& operator=(charT c);
```

Returns: `*this = basic_string<charT,traits,Allocator>(`*c*`)`.

21.1.1.5 `basic_string` iterator support [lib.string.iterators]

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

Returns: an iterator referring to the first character in the string.

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

Returns: an iterator which is the past-the-end value.

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

Returns: an iterator which is semantically equivalent to `reverse_iterator(end())`.

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

Returns: an iterator which is semantically equivalent to `reverse_iterator(begin())`.

21.1.1.6 `basic_string` capacity [lib.string.capacity]

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

Returns: a count of the number of char-like objects currently in the string.
Notes: Uses `traits::length()`.

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

Returns: `size()`.

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

Returns: The maximum size of the string.

```
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()`.

```
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()`.

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

Returns: `size() == 0`.

21.1.1.7 `basic_string` element access [lib.string.access]

```
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.

```
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*`)`.

21.1.1.8 `basic_string` modifiers [lib.string.modifiers]

21.1.1.8.1 `basic_string::operator+=` [lib.string::op+=]

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

Returns: `append(`*rhs*`)`.

```
basic_string<charT,traits,Allocator>& operator+=(const charT* s);
```

Returns: `*this += basic_string<charT,traits,Allocator>(`*s*`)`.
Notes: Uses `traits::length()`.

```
basic_string<charT,traits,Allocator>& operator+=(charT c);
```

Returns: `*this += basic_string<charT,traits,Allocator>(`*c*`)`.

21.1.1.8.2 `basic_string::append` [lib.string::append]

```
basic_string<charT,traits,Allocator>&
  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 `size()` 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,traits,Allocator>&
  append(const charT* s, size_type n);
```

Returns: `append(basic_string<charT,traits,Allocator>(`*s*`,`*n*`))`.

```
basic_string<charT,traits,Allocator>& append(const charT* s);
```

Returns: `append(basic_string<charT,traits,Allocator>(`*s*`))`.
Notes: Uses `traits::length()`.

```
basic_string<charT,traits,Allocator>&
  append(size_type n, charT c = charT());
```

Returns: `append(basic_string<charT,traits,Allocator>(`*c*`,`*n*`))`.

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

Returns: `append(basic_string<charT,traits,Allocator>(`*first*`,` *last*`))`.

21.1.1.8.3 `basic_string::assign` [lib.string::assign]

```
basic_string<charT,traits,Allocator>&
  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,traits,Allocator>&
  assign(const charT* s, size_type n);
```

Returns: `assign(basic_string<charT,traits,Allocator>(s,n))`.

```
basic_string<charT,traits,Allocator>& assign(const charT* s);
```

Returns: `assign(basic_string(s))`.
Notes: Uses `traits::length()`.

```
basic_string<charT,traits,Allocator>&
  assign(size_type n, charT c = charT());
```

Returns: `assign(basic_string<charT,traits,Allocator>(c,n))`.

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

Returns: `assign(basic_string<charT,traits,Allocator>(first, last))`.

21.1.1.8.4 `basic_string::insert` [lib.string::insert]

```
basic_string<charT,traits,Allocator>&
  insert(size_type pos1,
         const basic_string<charT,traits,Allocator>& 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.size() - 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,traits,Allocator>&
  insert(size_type pos, const charT* s, size_type n);
```

Returns: `insert(pos,basic_string<charT,traits,Allocator>(s,n))`.

```
basic_string<charT,traits,Allocator>&
  insert(size_type pos, const charT* s);
```

Returns: `insert(pos,basic_string<charT,traits,Allocator>(s))`.
Notes: Uses `traits::length()`.

```
basic_string<charT,traits,Allocator>&
  insert(size_type pos, size_type n, charT c = charT());
```

Returns: `insert(pos,basic_string<charT,traits,Allocator>(c,n))`.

```
iterator insert(iterator p, charT c = charT());
```

Requires: *p* is a valid iterator on `*this`.
Effects: inserts a copy of *c* before the character referred to by *p*.
Returns: *p*.

```
iterator insert(iterator p, size_type n, charT c = charT());
```

Requires: *p* is a valid iterator on `*this`.
Effects: inserts *n* copies of *c* before the character referrred to by *p*.

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

Requires: *p* is a valid iterator on `*this`. `[`*first*`,` *last*`)` is a valid range.
Effects: inserts copies of the characters in the range `[`*first*`,` *last*`)` before the character referrred to by *p*.

21.1.1.8.5 basic_string::remove [lib.string::remove]

```
basic_string<charT,traits,Allocator>&
  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`.

```
basic_string& remove(iterator p);
```

Requires: *p* is a valid iterator on `*this`.
Effects: removes the character referred to by *p* and calls the character's destructor.
Returns: `*this`.

```
basic_string& remove(iterator first, iterator last);
```

Requires: *first* and *last* are valid iterators on `*this`, defining a range `[`*first*`,` *last*`)`.
Effects: removes the characters in the range `[`*first*`,` *last*`)` and calls the character's destructor.
Complexity: the destructor is called a number of times exactly equal to the size of the range.
Returns: `*this`.

21.1.1.8.6 basic_string::replace [lib.string::replace]

```
basic_string<charT,traits,Allocator>&
  replace(size_type pos1, size_type n1,
          const basic_string<charT,traits,Allocator>& 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* .
Throws length_error if size() - *xlen* >= npos - *rlen*.
Otherwise, the function replaces the string controlled by *this with a string of length size() - *xlen* + *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 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,traits,Allocator>&
  replace(size_type pos, size_type n1, const charT* s, size_type n2);
```

Returns: replace(*pos*,*n1*,basic_string<charT,traits,Allocator>(*s*,*n2*)).

```
basic_string<charT,traits,Allocator>&
  replace(size_type pos, size_type n1, const charT* s);
```

Returns: replace(*pos*,*n1*,basic_string<charT,traits,Allocator>(*s*)).
Notes: Uses traits::length().

```
basic_string<charT,traits,Allocator>&
  replace(size_type pos, size_type n, charT c = charT());
```

Returns: replace(*pos*,*n*,basic_string<charT,traits,Allocator>(*c*,*n*)).

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

Requires: The iterators *i1* and *i2* are valid iterators on *this, defining a range [*i1*,*i2*).
Effects: Replaces the string controlled by *this with a string of length size() - (*i2* - *i1*) + *str*.size() whose first begin() - *i1* elements are a copy of the initial elements of the original string controlled by *this, whose next *str*.size() elements are a copy of the string controlled by *str*, and whose remaining elements are a copy of the elements of the original string controlled by *this beginning at position *i2*.
Returns: *this.
Notes: After the call, the length of the string will be changed by: *str*.size() - (*i2* - *i1*).

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

Returns: replace(*i1*,*i2*,basic_string(*s*,*n*)).
Notes: Length change: *n* - (*i2* - *i1*).

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

Returns: replace(*i1*,*i2*,basic_string(*s*)).
Notes: Length change: traits::length(*s*) - (*i2* - *i1*).
Uses traits::length().

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

Returns: `replace(i1,i2,basic_string(n,c))`.
Notes: Length change: `n - (i2 - i1)`.

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

Returns: `replace(i1,i2,basic_string(j1,j2))`.
Notes: Length change: `j2 - j1 - (i2 - i1)`.

21.1.1.8.7 `basic_string::copy` [lib.string::copy]

```
size_type copy(charT* 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*.
Notes: The function does not append a null object to the string.
Returns: *rlen*.

21.1.1.8.8 `basic_string::swap` [lib.string::swap]

```
void swap(basic_string<charT,traits,Allocator>& 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`.
Complexity: Constant time.

21.1.1.9 `basic_string` string operations [lib.string.ops]

```
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()`.

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

Returns: `c_str()` if `size()` is nonzero, otherwise a null pointer.
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`.

21.1.1.9.1 `basic_string::find` [lib.string::find]

```
size_type find(const basic_string<charT,traits,Allocator>& 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();`

— `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`.
Notes: Uses `traits::eq()`.

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

Returns: `find(basic_string<charT,traits,Allocator>(`*s*`,`*n*`)`*pos*`)`.

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

Returns: `find(basic_string<charT,traits,Allocator>(`*s*`),`*pos*`)`.
Notes: Uses `traits::length()`.

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

Returns: `find(basic_string<charT,traits,Allocator>(`*c*`),`*pos*`)`.

21.1.1.9.2 `basic_string::rfind` [lib.string::rfind]

```
size_type rfind(const basic_string<charT,traits,Allocator>& 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();`

— `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`.
Notes: Uses `traits::eq()`.

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

Returns: `rfind(basic_string<charT,traits,Allocator>(`*s*`,`*n*`),`*pos*`)`.

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

Returns: `rfind(basic_string<charT,traits,Allocator>(`*s*`),`*pos*`)`.
Notes: Uses `traits::length()`.

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

Returns: `rfind(basic_string<charT,traits,Allocator>(`*c*`,`*n*`),`*pos*`)`.

21.1.1.9.3 `basic_string::find_first_of` [lib.string::find.first.of]

```
size_type
  find_first_of(const basic_string<charT,traits,Allocator>& 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();

— 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.

Notes: Uses traits::eq().

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

Returns: find_first_of(basic_string<charT,traits,Allocator>(*s*,*n*),*pos*).

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

Returns: find_first_of(basic_string<charT,traits,Allocator>(*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,traits,Allocator>(*c*),*pos*).

21.1.1.9.4 `basic_string::find_last_of` [lib.string::find.last.of]

```
size_type
  find_last_of(const basic_string<charT,traits,Allocator>& 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();

— 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.

Notes: Uses traits::eq().

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

Returns: find_last_of(basic_string<charT,traits,Allocator>(*s*,*n*),*pos*).

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

Returns: find_last_of(basic_string<charT,traits,Allocator>(*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,traits,Allocator>(c),pos)`.

21.1.1.9.5 `basic_string::find_first_not_of` [lib.string::find.first.not.of]

```
size_type
  find_first_not_of(const basic_string<charT,traits,Allocator>& 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()`;

— `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`.

Notes: Uses `traits::eq()`.

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

Returns: `find_first_not_of(basic_string<charT,traits,Allocator>(s,n),pos)`.

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

Returns: `find_first_not_of(basic_string<charT,traits,Allocator>(s),pos)`.

Notes: Uses `traits::length()`.

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

Returns: `find_first_not_of(basic_string<charT,traits,Allocator>(c),pos)`.

21.1.1.9.6 `basic_string::find_last_not_of` [lib.string::find.last.not.of]

```
size_type
  find_last_not_of(const basic_string<charT,traits,Allocator>& 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()`;

— `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`.

Notes: Uses `traits::eq()`.

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

Returns: `find_last_not_of(basic_string<charT,traits,Allocator>(s,n),pos)`.

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

Returns: `find_last_not_of(basic_string<charT,traits,Allocator>(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,traits,Allocator>(`*c*`),`*pos*`)`.

21.1.1.9.7 `basic_string::substr` [lib.string::substr]

```
basic_string<charT,traits,Allocator>
  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,traits,Allocator>(data()+`*pos*`,`*rlen*`)`.

21.1.1.9.8 `basic_string::compare` [lib.string::compare]

```
int compare(const basic_string<charT,traits,Allocator>& 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 44:

Table 44—`compare()` results

Condition	Return Value
`size()-`*pos* `<` *str*`.size()`	`< 0`
`size()-`*pos* `==` *str*`.size()`	`0`
`size()-`*pos* `>` `str.size()`	`> 0`

Notes: Uses `traits::compare()`.

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

Returns: `compare(basic_string<charT,traits,Allocator>(`*s*`,`*n*`),`*pos*`)`.
Notes: Uses `traits::compare()`.

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

Returns: `compare(basic_string<charT,traits,Allocator>(`*s*`),`*pos*`)`.
Notes: Uses `traits::length()` and `traits::compare()`.

21.1.1.10 `basic_string` non-member functions [lib.string.nonmembers]

21.1.1.10.1 `operator+` [lib.string::op+]

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

Returns: `lhs.append(rhs)`.

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

Returns: `basic_string<charT,traits,Allocator>(lhs) + rhs`.
Notes: Uses `traits::length()`.

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

Returns: `basic_string<charT,traits,Allocator>(lhs) + rhs`.

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

Returns: `lhs + basic_string<charT,traits,Allocator>(rhs)`.
Notes: Uses `traits::length()`.

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

Returns: `lhs + basic_string<charT,traits,Allocator>(rhs)`.

21.1.1.10.2 `operator==` [lib.string::operator==]

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

Returns: `lhs.compare(rhs) == 0`.

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

Returns: `basic_string<charT,traits,Allocator>(`*lhs*`) ==` *rhs*.

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

Returns: *lhs* `== basic_string<charT,traits,Allocator>(`*rhs*`)`.
Notes: Uses `traits::length()`.

21.1.1.10.3 `operator!=` [lib.string::op!=]

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

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

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

Returns: `basic_string<charT,traits,Allocator>(`*lhs*`) !=` *rhs*.

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

Returns: *lhs* `!= basic_string<charT,traits,Allocator>(`*rhs*`)`.
Notes: Uses `traits::length()`.

21.1.1.10.4 `operator<` [lib.string::op<]

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

Returns: *lhs*`.compare(`*rhs*`) < 0`.

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

Returns: `basic_string<charT,traits,Allocator>(`*lhs*`) <` *rhs*.

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

Returns: *lhs* `< basic_string<charT,traits,Allocator>(`*rhs*`)`.

21.1.1.10.5 `operator>` [lib.string::op>]

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

Returns: `lhs.compare(rhs) > 0`.

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

Returns: `basic_string<charT,traits,Allocator>(lhs) > rhs`.

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

Returns: `lhs > basic_string<charT,traits,Allocator>(rhs)`.

21.1.1.10.6 `operator<=` [lib.string::op<=]

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

Returns: `lhs.compare(rhs) <= 0`.

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

Returns: `basic_string<charT,traits,Allocator>(lhs) <= rhs`.

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

Returns: `lhs <= basic_string<charT,traits,Allocator>(rhs)`.

21.1.1.10.7 `operator>=` [lib.string::op>=]

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

Returns: `lhs.compare(rhs) >= 0`.

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

Returns: `basic_string<charT,traits,Allocator>(lhs) >= rhs`.

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

Returns: `lhs <= basic_string<charT,traits,Allocator>(rhs)`.

21.1.1.10.8 Inserters and extractors

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

Notes: Uses `traits::char_in` and `is_del()`.

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

Notes: Uses `traits::char_out()`.

```
template<class charT, class IS_traits, class STR_traits,
         class STR_Alloc>
  basic_istream<charT,IS_traits>&
    getline(basic_istream<charT,IS_traits>& is,
            basic_string<charT,STR_traits,STR_Alloc>& str,
            charT delim = IS_traits::newline() );
```

Effects: An unformatted input function, extracts a line (as delimited by *delim*) from *is* into *str*. The string is initially made empty by calling *str*`.remove(0)`. Each extracted character *c* is appended as if by calling *str*`.append(`*c*`)`. Characters are extracted and appended until any of the following occurs:

— `npos - 1` characters are appended (in which case the function calls *is*`.setstate(failbit)`, which may throw `ios_base::failure` (27.4.4.3)).

— end of file occurs on the input sequence (in which case the function calls *is*`.setstate(eofbit)`, which may throw `ios_base::failure` (27.4.4.3)).

— *c* `==` *delim* for the next available input character *c* (in which case the input character is extracted from *is*, but not appended to `str`).
If the function appends no characters, it calls *is*`.setstate(failbit)`, which may throw `ios_base::failure` (27.4.4.3).

Returns: *is*.

Notes: Uses `STR_traits::char_in()`.

21.1.2 Class `string` [lib.string]

```
namespace std {
  struct string_char_traits<char> {
    typedef char char_type;
    static void assign(char& c1, const char& c2);
    static bool eq(const char& c1, const char& c2);
    static bool ne(const char& c1, const char& c2);
    static bool lt(const char& c1, const char& c2);
    static char eos();

    static basic_istream<char>& char_in (basic_istream<char>& is, char& a);
    static basic_ostream<char>& char_out(basic_ostream<char>& os, char a);
    static bool is_del(char a); // characteristic function for delimiters

    static int compare(const char* s1, const char* s2, size_t n);
    static size_type length(const char* s);
    static char* copy(char* s1, const char* s2, size_t n);
  };

  typedef basic_string<char> string;
}
```

21.1.3 `string_char_traits<char>` members [lib.string.traits.members]

```
static void assign(char& c1, const char& c2);
```

Effects: `c1 = c2`.

```
static bool eq(const char& c1, const char& c2);
```

Returns: `c1 == c2`.

```
static bool ne(const char& c1, const char& c2);
```

Returns: `c1 != c2`.

```
static bool lt(const char& c1, const char& c2);
```

Returns: `c1 < c2`.

```
static char eos();
```

Returns: `0`.

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

Returns: `is >> a` .

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

Returns: `os << a`.

```
bool is_del(char a);
```

Returns: `::isspace(a)`.

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

Returns: `::memcmp(s1,s2,n)`.

```
static size_type length(const char* s);
```

Returns: `::strlen(s)`.

```
static char* copy(char* s1, const char* s2, size_t 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(wchar_t& c1, const wchar_t& c2);
    static bool eq(const wchar_t& c1, const wchar_t& c2);
    static bool ne(const wchar_t& c1, const wchar_t& c2);
    static bool lt(const wchar_t& c1, const wchar_t& c2);
    static wchar_t eos();

    static basic_istream<wchar_t>& char_in (basic_istream<wchar_t>& is, wchar_t& a);
    static basic_ostream<wchar_t>& char_out(basic_ostream<wchar_t>& os, wchar_t a);
    static bool is_del(wchar_t a); // characteristic function for delimiters

    static int compare(const wchar_t* s1, const wchar_t* s2, size_t n);
    static size_type length(const wchar_t* s);
    static wchar_t* copy(wchar_t* s1, const wchar_t* s2, size_t n);
  };

  typedef basic_string<wchar_t> wstring;
}
```

21.1.5 `string_char_traits<wchar_t>` members [lib.wstring.members]

```
static void assign(wchar_t& c1, const wchar_t& c2);
```

Effects: `c1 = c2`.

```
static bool eq(const wchar_t& c1, const wchar_t& c2);
```

Returns: `c1 == c2`.

```
static bool ne(const wchar_t& c1, const wchar_t& c2);
```

Returns: `c1 != c2`.

```
static bool lt(const wchar_t& c1, const wchar_t& c2);
```

Returns: `c1 < c2`.

```
static wchar_t eos();
```

Returns: `0`.

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

Returns: `is >> a` .

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

Returns: `os << a`.

```
bool is_del(wchar_t a);
```

Returns: `::iswspace(a)`.

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

Returns: `::wmemcmp(s1,s2,n)`.

```
static size_type length(const wchar_t* s);
```

Returns: `::wcslen(s)`.

```
static wchar_t* copy(wchar_t* s1, const wchar_t* s2, size_t 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 44—Header `<cctype>` synopsis

Type	Name(s)			
Functions:				
isalnum	isdigit	isprint	isupper	tolower
isalpha	isgraph	ispunct	isxdigit	toupper
iscntrl	islower	isspace		

Table 44—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 44—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 44—Header `<cwchar>` synopsis

Type	Name(s)				
Macros:	`NULL <cwchar>`	`WCHAR_MAX`	`WCHAR_MIN`	`WEOF <cwchar>`	
Types:	`mbstate_t`	`wint_t <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 44—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_t)` is replaced by the two declarations:

```
const void* memchr(const void* s, int c, size_t n);
      void* memchr(      void* s, int c, size_t n);
```

13 both of which have the same behavior as the original declaration.

14 The function signature `wcschr(const wchar_t*, wchar_t)` is replaced by the two declarations:

```
const wchar_t* wcschr(const wchar_t* s, wchar_t c);
      wchar_t* wcschr(      wchar_t* s, wchar_t c);
```

15 both of which have the same behavior as the original declaration.

16 The function signature `wcspbrk(const wchar_t*, const wchar_t*)` is replaced by the two declarations:

```
const wchar_t* wcspbrk(const wchar_t* s1, const wchar_t* s2);
      wchar_t* wcspbrk(      wchar_t* s1, const wchar_t* s2);
```

17 both of which have the same behavior as the original declaration.

18 The function signature `wcsrchr(const wchar_t*, wchar_t)` is replaced by the two declarations:

```
const wchar_t* wcsrchr(const wchar_t* s, wchar_t c);
      wchar_t* wcsrchr(      wchar_t* s, wchar_t c);
```

19 both of which have the same behavior as the original declaration.

20 The function signature `wcswcs(const wchar_t*, const wchar_t*)` is replaced by the two declarations:

```
const wchar_t* wcsstr(const wchar_t* s1, const wchar_t* s2);
      wchar_t* wcsstr(      wchar_t* s1, const wchar_t* s2);
```

21 both of which have the same behavior as the original declaration.

22 The function signature `wmemchr(const wwchar_t_t*, int, size_t)` is replaced by the two declarations:

```
const wchar_t* wmemchr(const wchar_t* s, wchar_t c, size_t n);
      wchar_t* wmemchr(      wchar_t* s, wchar_t c, size_t n);
```

23 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 more portable when confronting) cultural differences. The locale facility includes internationalization support for character classification and string collation, numeric, monetary, and date/time formatting and parsing, and message retrieval.

2 The following subclauses describe components for locales themselves, the standard facets, and facilities from the ISO C library, as summarized in Table 45:

Table 45—Localization library summary

Subclause	Header(s)
22.1 Locales 22.2 Standard `locale` Categories	`<locale>`
22.3 C library locales	`<clocale>`

22.1 Locales [lib.locales]

Header `<locale>` synopsis

```
#include <limits>
#include <string>
#include <iosfwd>
#include <stdexcept>    // for runtime_error
#include <vector>       // for vector<char>

namespace std {
// subclause 22.1.1, locale:
  class locale;
  template <class charT, class Traits>
    basic_ostream<charT,Traits>&
      operator<<(basic_ostream<charT,Traits>& s, const locale& loc);
  template <class charT, class Traits>
    basic_istream<charT,Traits>&
      operator>>(basic_istream<charT,Traits>& 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 22.2.1.3, ctype:
  class ctype_base;
  template <class charT> class ctype;
                         class ctype<char>;        // specialization
  template <class charT> class ctype_byname;
                         class ctype_byname<char>; // specialization
  class codecvt_base;
  template <class fromT, class toT, class stateT> class codecvt;
  template <class fromT, class toT, class stateT> class codecvt_byname;

// subclauses 22.2.2 and 22.2.3, 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 22.2.4, collation:
  template <class charT> class collate;
  template <class charT> class collate_byname;

// subclause 22.2.5, 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 22.2.6, 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 22.2.7, message retrieval:
  class messages_base;
  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.[173)]

173) In this subclause, the type name `struct tm` is an incomplete type that is defined in `<ctime>`.

22.1.1 Class `locale` [lib.locale]

```
namespace std {
  class locale {
  public:
  // types:
    class facet;
    class id;
    typedef unsigned category;
    static const category   // values assigned here are for exposition only
      none     = 0,
      collate  = 0x010, ctype    = 0x020,
      monetary = 0x040, numeric  = 0x080,
      time     = 0x100, messages = 0x200,
      all = collate | ctype | monetary | numeric | time  | messages;

  // construct/copy/destroy:
    locale();
    locale(const locale& other);
    explicit 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);

  // locale operations:
    template <class Facet> const Facet& use()  const;
    template <class Facet> bool         has()  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;

  // global locale objects:
    static       locale  global(const locale&);
    static const locale& classic();
    static       locale  transparent();
  };
}
```

1 Class `locale` implements a type-safe polymorphic set of facets, indexed by facet *type*. In other words, 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.

2 Access to the facets of a `locale` is via two member function templates, `locale::use<facet>()` and `locale::has<facet>()`.

3 [*Example:* An iostream `operator<<` might be implemented (and specialized, for simplicity of exposition) as:

```
ostream& operator<<(ostream& s, double f)
{
  if (s.opfx()) {
    locale loc = s.getloc();
    loc.template use< num_put<char> >().put(s, s, loc, f);
  }
  s.osfx();
  return s;
}
```

—*end example*]

4 In the call to `loc.template use<Facet>()`, the type argument chooses a facet, making available all members of the named type. If `Facet` is not present in a locale (or, failing that, in the `global` locale), it throws the standard exception `bad_cast`. A C++ program can check if a locale implements a particular facet with the member `has<Facet>()`. User-defined facets may be installed in a locale, and used identically as may standard facets (22.2.8).

5 All locale semantics are accessed via `use<>()` and `has<>()`, with two exceptions:

— Convenient global interfaces are provided for traditional `ctype` functions such as `isdigit()` and `isspace()`, so that given a locale object *`loc`* a C++ program can call `isspace(`*`c`*`,` *`loc`*`)`.

— A member operator template `operator()(basic_string<C>&, basic_string<C>&)` is provided so that a locale may be used as a predicate argument to the standard collections, to collate strings.

6 [*Note:* The purpose of this is to ease the conversion of existing extractors (27.6.1.2). —*end note*]

7 A locale which does not implement a facet delegates to the global locale in effect at the time that instantiation of `use<>()` is first called on that facet (22.1.1.5).

8 An instance of `locale` is *immutable*; once a facet reference is obtained from it, that reference remains usable as long as the locale value itself exists. The effect of imbuing on a stream (27.4.3, 27.4.4), or installing as the global locale, the result of static member `locale::transparent()` (or any locale with similar behavior) is unspecified.

9 Caching results from calls to locale facet member functions during calls to iostream inserters and extractors, and in streambufs between calls to `basic_streambuf::imbue`, is explicitly supported (27.5.2).[174)]

10 A `locale` constructed from a name string (such as `"POSIX"`), 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 "`*`".

22.1.1.1 `locale` types [lib.locale.types]

22.1.1.1.1 Type `locale::category` [lib.locale.category]

```
typedef unsigned category;
```

1 *Valid* `category` values include 0 and the `locale` member bitmask elements `collate`, `ctype`, `monetary`, `numeric`, `time`, and `messages`. In addition, `locale` member `all` is defined such that the expression

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

is `true`. Further, the result of applying operators `&` and `|` to any two valid values is itself valid.

174) This implies that member functions of iostream classes cannot safely call `imbue()` themselves, except as specified elsewhere.

2 `locale` member functions expecting a `category` argument require either a valid `category` value or one of the constants `LC_CTYPE` etc., defined in `<cctype>`. Such a `category` value identifies a set of locale categories. Each locale category, in turn, identifies a set of locale facets, as shown in Table 46:

Table 46—Locale Category Facets

Category	Includes Facets
`collate`	`collate<char>, collate<wchar_t>`

`ctype`	`ctype<char>, ctype<wchar_t>`
	`codecvt<char,wchar_t,mbstate_t>,`
	`codecvt<wchar_t,char,mbstate_t>`

`monetary`	`moneypunct<char>, moneypunct<wchar_t>`
	`moneypunct<char,true>, moneypunct<wchar_t,true>,`
	`money_get<char,InputIterator>,`
	`money_get<wchar_t,InputIterator>,`
	`money_put<char,OutputIterator>,`
	`money_put<wchar_t,OutputIterator>`

`numeric`	`numpunct<char>, numpunct<wchar_t>,`
	`num_get<C,InputIterator>, num_put<C,OutputIterator>`

`time`	`time_get<char,InputIterator>,`
	`time_put<wchar_t,OutputIterator>,`
	`time_put<char,OutputIterator>,`
	`time_put<wchar_t,OutputIterator>`

`messages`	`messages<char>, messages<wchar_t>`

3 An implementation is only required to provide instantiations for the facets identified as implementing a `category`. For the facets `num_get<>` and `num_put<>` the implementation provided must depend only on the facets `numpunct<>` and `ctype<>`, instantiated on the same character type. Other facets are allowed to depend on any other facet that is part of a standard category.

4 Each `locale` member function which takes a `category` argument operates on the corresponding set of facets. Those facets represented with a template parameter `InputIterator` or `OutputIterator` indicate the set of all possible instantiations on parameters that satisfy the requirements of an Input Iterator or an Output Iterator, respectively. Those facets represented with a template parameter `C` represent the set of all possible instantiations on a parameter that satisfies the requirements for a character on which any of the iostream components can be instantiated.

5 In declarations of facets, a template formal parameter with name `InputIterator` or `OutputIterator` indicates that instantiations depend only on the semantics specified for an Input Iterator or an Output Iterator as defined in 24.1.

22.1.1.1.2 Class `locale::facet` [lib.locale.facet]

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

1 Class `facet` is the base class for locale feature sets. A class is a *facet* if it is publicly derived from another facet, or if it is a class derived from `locale::facet` and containing a declaration as follows:

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

Template parameters in this Clause which must be facets are those named `Facet` in declarations. A program that passes a type that is *not* a facet, as an (explicit or deduced) template parameter to a locale function expecting a facet, is ill-formed.

2 The *`refs`* argument to the constructor is used for lifetime management.

— If `(`*`refs`*` == 0)` the facet's lifetime is managed by the locale or locales it is incorporated into;

— if `(`*`refs`*` == 1)` its lifetime is until explicitly deleted.

3 Constructors of all facets defined in this Clause take such an argument and pass it along to their `facet` base class constructor. All one-argument constructors defined in this clause are *explicit*, preventing their participation in automatic conversions.

4 For some standard facets a standard "..._byname" class, derived from it, implements the semantics equivalent to that facet of the locale constructed by `locale(const char*)`. Each such facet provides a constructor that takes a `const char*` argument, which names the locale, and a *`refs`* argument, which is passed to the base class constructor. If there is no "..._byname" version of a facet, the base class implements such semantics itself, sometimes with the help of other facets obtained via a `locale` argument.

22.1.1.1.3 Class `locale::id` [lib.locale.id]

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

1 Identification of a locale facet interface, used as an index for lookup and to encapsulate initialization.

2 [*Note:* Because facets are used by iostreams, potentially while static constructors are running, their initialization cannot depend on programmed static initialization.[175] —*end note*]

22.1.1.2 `locale` constructors and destructor [lib.locale.cons]

```
locale();
```

1 Default constructor: a snapshot of the current global locale.
Effects: Constructs a locale instance whose value is a snapshot of the current global locale state as set by `locale::global(locale&)` or the C function `setlocale()`. This constructor is commonly used as the default value for arguments of functions that take a `locale` argument.

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

Effects: Constructs a locale which is a copy of *`other`*.

[175] One way to do this is for `locale` to initialize the `id` member the first time an instance of the facet is installed into a locale. This depends only on static storage being zero before constructors run (3.6.2).

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

Effects: Creates a copy of *`other`*, replacing the current value.
Returns: `*this`

`explicit locale(const char* std_name);`

Effects: Constructs a locale using standard C locale names, e.g. `"POSIX"`. The resulting locale implements semantics defined to be associated with that name.
Requires: The set of valid string argument values is `"C"`, `""`, and any implementation-defined values.

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

Effects: Constructs a locale as a copy of `other` except for the facets identified by the `category` argument, which instead implement the same semantics as `locale(`*`std_name`*`)`.
Notes: The locale has a name if and only if `other` has a name.

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

Effects: Constructs a locale incorporating all facets from the first argument except that of type `Facet`, and installs the second argument as the remaining facet.
Notes: The resulting locale has no name.

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

Effects: Constructs a locale incorporating all facets from the first argument except that identified by `Facet`, and that facet from the second argument instead.
Throws: `runtime_error` if *`one`*`.template has<Facet>()` is `false`.
Notes: The resulting locale has no name.

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

Effects: Constructs a locale incorporating all facets from the first argument except those that implement `cats`, which are instead incorporated from the second argument.
Notes: The resulting locale has a name if and only if the first two arguments have names.

`~locale();`

2 A non-virtual destructor.

22.1.1.3 `locale` members [lib.locale.members]

`template <class Facet> const Facet& use() const;`

1 Get a reference to a facet of a locale.
Effects: If the requested `Facet` is not present in `*this`, but is present in the current global locale, returns the global locale's instance of `Facet`. Because locale objects are *immutable*, subsequent calls to `use<Facet>()` return the same object, regardless of changes to the global locale.[176)]

176) The only exception to this rule is the locale returned by `locale::transparent()`; it always returns the `Facet` found in the global locale at the time of each call.

Throws: `bad_cast` if `(this->template has<Facet>() || locale().template has<Facet>())` is `false`.
Returns: A reference to the requested facet.
Notes: The result is guaranteed by `locale`'s value semantics to last as long as the value of `*this`.

```
template <class Facet> bool has() const;
```

Returns: An indication whether the facet requested is present in `*this`. If `use<Facet>()` has already been called successfully, returns `true`.
Notes: `locale::transparent().template has<Facet>()` always returns `false`.

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

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

22.1.1.4 `locale` operators [lib.locale.operators]

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

Returns: `true` if both arguments are the same locale, or one is a copy of the other, or each has a name and the names are identical; `false` otherwise.

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

Returns: The result of the expression: `!(*this == other)`

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

Effects: Compares two strings according to the `collate<charT>` facet.
Notes: This member operator template (and therefore `locale` itself) satisfies requirements for a comparator predicate template argument (25) applied to strings.
Returns: The result of the following expression:

```
use< collate<charT> >().compare(s1.data(), s1.data()+s1.size(),
                                s2.data(), s2.data()+s2.size()) < 0;
```

1 [*Example:* A vector of strings `v` can be collated according to collation rules in locale `loc` simply by (25.3.1, 23.2.5):

```
  std::sort(v.begin(), v.end(), loc);
```

—*end example*]

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

2 The regular stream output operator for locales (27.6.2.4).
Effects: `s << loc.name() << endl`.
Returns: The output stream argument `s`.

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

3 The regular stream input operator for locales (27.6.1.2).

Effects: Read a line into a string and construct a locale from it. If either operation fails, indicates a failure by calling `s.setstate(ios_base::failbit)` (which may throw `ios_base::failure` (27.4.4.3), otherwise, assigns the constructed `locale` object into the argument `loc`.

Returns: *s*.

22.1.1.5 `locale` static members [lib.locale.statics]

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

1 Replaces `::setlocale()`.

Effects: Sets the global locale to its argument. Subsequent calls to the default constructor, and of other library functions affected by the function `setlocale()`, use the locale *loc* until the next call to this member or `setlocale()`.

Returns: The previous global locale.

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

2 The `"C"` locale.

Returns: A locale that implements the classic `"C"` locale semantics, equivalent to the value `locale("C")`.

Notes: This locale, its facets, and their member functions, do not change with time.

```
static locale transparent();
```

3 The continuously updated global locale.

Returns: A locale which implements semantics that vary dynamically as the global locale is changed.

Notes: The effect of imbuing this locale into an iostreams component is unspecified (_lib.ios.members_).

22.1.2 Convenience interfaces [lib.locale.convenience]

22.1.2.1 Character classification [lib.classification]

```
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;
```

1 Each of these functions `is`***F*** returns the result of the expression:

```
  loc.template use< ctype<charT> >().is(ctype<charT>::F, c)
```

where ***F*** is the `ctype_mask` value corresponding to that function (22.2.1).

22.1.2.2 Character conversions [lib.conversions]

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

Returns: `loc.template use<ctype<charT> >().toupper(c)`.

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

Returns: `loc.template use<ctype<charT> >().tolower(c)`.

22.2 Standard `locale` categories [lib.locale.categories]

1 Each of the standard categories includes a family of facets. Some of these implement formatting or parsing, intended for use by standard or users' operators `<<` and `>>`. Those that take a `basic_ios<charT>&` argument obey all formatting conventions specified for members of that class, including `width()` and `fill()` (27.4.3).

22.2.1 The `ctype` category [lib.category.ctype]

```
namespace std {
  class ctype_base {
  public:
    enum ctype_mask {  // numeric values are for exposition only.
      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=alpha|digit, graph=alnum|punct
    };
  };
}
```

1 The type `ctype_mask` is a bitmask type.

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;
    explicit ctype(size_t refs = 0);

    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* low, const char* high, charT* to) const;
    char         narrow(charT c, char dfault) const;
    const charT* narrow(const charT* low, const charT*, char dfault,
                        char* to) const;

    static locale::id id;

  protected:
   ~ctype();  // virtual
    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* low, const char* high,
                                  charT* dest) const;
    virtual char         do_narrow(charT, char dfault) const;
    virtual const charT* do_narrow(const charT* low, const charT* high,
                                   char dfault, char* dest) const;
  };
```

1 Class `ctype` encapsulates the C library `<cctype>` features. `istream` members are required to use `ctype<>` for character classing during input parsing.

2 The base class implementation implements character classing appropriate to the implementation's native character set.

22.2.1.1.1 `ctype` members [lib.locale.ctype.members]

```
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)`

```
const charT* scan_is(ctype_mask mask,
                          const charT* low, const charT* high) const;
```

Returns: `do_scan_is(mask,low,high)`

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

Returns: `do_scan_not(mask,low,high)`

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

Returns: `do_toupper(`*c*`)` or `do_toupper(`*low*`,`*high*`)`

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

Returns: `do_tolower(`*c*`)` or `do_tolower(`*low*`,`*high*`)`

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

Returns: `do_widen(`*c*`)` or `do_widen(`*low*`,`*high*`,`*to*`)`

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

Returns: `do_narrow(`*c*`,`*dfault*`)` or `do_narrow(`*low*`,`*high*`,`*dfault*`,`*to*`)`

22.2.1.1.2 **ctype** virtual functions [lib.locale.ctype.virtuals]

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

Effects: Classifies a character or sequence of characters. For each argument character, identifies a value *M* of type `ctype_mask` The first form returns the result of the expression `(`*M*` & `*mask*`) != 0`. The second form simply places *M* for all `*`*p* where `(`*low*`<=`*p*` && `*p*`<`*high*`)`, into *vec*`[`*p*`-`*low*`]`.

Returns: The first form returns `true` if the character has the characteristics specified. The second form returns *low*.

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

Effects: Locates a character in a buffer that conforms to a classification *mask*.

Returns: The smallest pointer *p* in the range `[`*low*`, `*high*`)` such that `is(*`*p*`)` would return `true`; otherwise, returns *high*.

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

Effects: Locates a character in a buffer that fails to conform to a classification `mask`.

Returns: The smallest pointer *p*, if any, in the range `[`*low*`, `*high*`)` such that `is(*`*p*`)` would return `false`; otherwise, returns *high*.

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

Effects: Converts a character or characters to upper case.

Effects: The second form replaces each character `*`*p* in the range `[`*low*`, `*high*`)` for which a corresponding upper-case character exists, with that character.

Returns: The first form returns the corresponding upper-case character if it is known to exist, or its argument if not. The second form returns *high*.

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

Effects: Converts a character or characters to upper case.
Effects: The second form replaces each character `*p` in the range `[low, high)` and for which a corresponding lower-case character exists, with that character.
Returns: The first form returns the corresponding lower-case character if it is known to exist, or its argument if not. The second form returns *`high`*.

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

Effects: Applies the simplest reasonable transformation from a `char` value or sequence of `char` values to the corresponding `charT` value or values. The only characters for which unique transformations are required are the digits, alphabetic characters, `'-'`, `'+'`, newline, and space.
For any named `ctype` category with a `ctype<charT>` facet *`ctw`* and valid `ctype_mask` value *`M`*, however, `(is(M, c) || !ctw.is(M, do_widen(c)) )` is `true`.[177)]
The second form transforms each character `*p` in the range `[low, high)`, placing the result in `dest[p-low]`.
Returns: The first form returns the transformed value. The second form returns *`high`*

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

Effects: Applies the simplest reasonable transformation from a `charT` value or sequence of `charT` values to the corresponding `char` value or values. The only characters for which unique transformations are required are the digits, alphabetic characters, `'-'`, `'+'`, newline, and space.
For any named `ctype` category with a `ctype<char>` facet *`ctc`* however, and valid `ctype_mask` value *`M`*, `(is(M, c) || !ctc.is(M, do_narrow(c)) )` is `true`. In addition, for any digit character *`c`*, the expression `(do_narrow(c)-'0')` evaluates to the digit value of the character.
Effects: The second form transforms each character `*p` in the range `[low, high)`, placing the result (or *`dfault`* if no simple transformation is readily available) in `dest[p-low]`.
Returns: The first form returns the transformed value; or *`dfault`* if no mapping is readily available. The second form returns *`high`*.

22.2.1.2 Template class `ctype_byname` [lib.locale.ctype.byname]

```
  template <class charT>
  class ctype_byname : public ctype<charT> {
  public:
    explicit ctype_byname(const char*, size_t refs = 0);
  protected:
   ~ctype_byname();  // virtual
    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;
  };
}
```

177) In other words, the transformed character is not a member of any character classification that *c* is not also a member of.

1 This class is specialized for at least `char` and `wchar_t`.

22.2.1.3 `ctype` specializations [lib.facet.ctype.special]

```
namespace std {
  class ctype<char> : public locale::facet, public ctype_base {
  public:
    typedef char char_type;

    explicit ctype(const ctype_mask* tab = 0, bool del = false,
                   size_t refs = 0);

    bool is(ctype_mask mask, char c) const;
    const char* is(const char* low, const char* high, 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* low, const char* high, char* to) const;
    char  narrow(char c, char /*dfault*/) const;
    const char* narrow(const char* low, const char* high, char /*dfault*/,
                       char* to) const;

    static locale::id id;

  protected:
    const ctype_mask* const table_;
    static const ctype_mask classic_table_[numeric_limits<unsigned char>::max()+1];

   ~ctype();  // virtual
    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;
  };
  private:
    bool delete_it_                  // exposition only
}
```

1 A specialization `ctype<char>` is provided so that the member functions on type `char` can be implemented `inline`.[178)]

22.2.1.3.1 `ctype<char>` destructor [lib.facet.ctype.char.dtor]

`~ctype();`

Effects: `if (`***delete_it_***`) delete[]` ***table_***`;`

[178)] Only the `char` (not `unsigned char` and `signed char`) form is provided. The specialization is specified in the standard, and not left as an implementation detail, because it affects the derivation interface for `ctype<char>`.

22.2.1.3.2 `ctype<char>` members [lib.facet.ctype.char.members]

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

Effects: Passes its *refs* argument to its base class constructor, initializes protected member ***table_*** with the *tab* argument if nonzero, or the static value ***classic_table_*** otherwise, and initializes the protected member ***delete_it_*** to `(tab && del)`.

```
bool        is(ctype_mask mask, char c) const;
const char* is(const char* low, const char* high,
               ctype_mask* vec) const;
```

Effects: The second form, for all `*`*p* in the range `[`*low*, *high*`)`, assigns *vec*`[`*p*-*low*`]` to ***table_***`[(unsigned char)*`*p*`]`.
Returns: The first form returns ***table_***`[(unsigned char)c] & mask`; the second form returns *low*.

```
const char* scan_is(ctype_mask mask,
                    const char* low, const char* high) const;
```

Returns: The smallest *p* in the range `[`*low*, *high*`)` such that `(`***table***`[(unsigned char) *`*p*`] & mask) == true`.

```
const char* scan_not(ctype_mask mask,
                     const char* low, const char* high) const;
```

Returns: The smallest *p* in the range `[`*low*, *high*`)` such that `(`***table***`[(unsigned char) *`*p*`] & mask) == false`.

```
char        toupper(char c) const;
const char* toupper(char* low, const char* high) const;
```

Returns: `do_toupper(`*c*`)` or `do_toupper(`*low*,*high*`)`

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

Returns: `do_tolower(`*c*`)` or `do_tolower(`*low*,*high*`)`

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

Effects: `::memcpy(`*to*, *low*, *high*-*low*`)`
Returns: `c` or `hi`

```
char        narrow(char c, char /*dfault*/) const;
const char* narrow(const char* low, const char* high,
                   char /*dfault*/, char* to) const;
```

Effects: `::memcpy(`*to*, *low*, *high*-*low*`)`
Returns: *c* or *high*.

22.2.1.3.3 ctype<char> overridden virtual functions [lib.facet.ctype.char.virtuals]

22.2.1.4 Template class codecvt [lib.locale.codecvt]

```
namespace std {
  class codecvt_base {
  public:
    enum result { ok, partial, error, noconv };
  };
  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;

    explicit codecvt(size_t refs = 0)

    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;

  protected:
   ~codecvt();  // virtual
    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;
  };
}
```

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 instantiated oppositely.

2 The `stateT` argument selects the pair of codesets being mapped between.

3 Implementations are required to provide instantiations for `<char,wchar_t,mbstate_t>` and `<wchar_t,char,mbstate_t>`.

22.2.1.4.1 codecvt members [lib.locale.codecvt.members]

```
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);`

22.2.1.4.2 codecvt virtual functions [lib.locale.codecvt.virtuals]

```
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;
```

Preconditions: `(from<=from_end && to<=to_end)` well-defined and `true`; *state* initialized, if at the beginning of a sequence, or else equal to the result of converting the preceding characters in the sequence.

Effects: Translates characters in the range [*from*, *from_end*), placing the results starting at *to*. Stops when it runs out of characters to translate or space to put the results, or if it encounters a character it cannot convert. It always leaves the *from_next* and *to_next* pointers pointing one beyond the last character successfully converted.
If no translation is needed (returns noconv), sets *to_next* equal to argument *to*.
Notes: Does not write into **to_limit*. Its operations on *state* are unspecified.
[*Note:* This argument can be used, for example, to maintain shift state, to specify conversion options (such as count only), or to identify a cache of seek offsets. *—end note*]
Returns: An enumeration value, as summarized in Table 47:

Table 47—convert result values

Value	Meaning
ok	completed the conversion
partial	ran out of space in the destination
error	encountered a from_type character it could not convert
noconv	no conversion was needed

22.2.1.5 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> {
  public:
    explicit codecvt_byname(const char*, size_t refs = 0);
  protected:
   ~codecvt_byname();  // virtual
    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;
  };
}
```

22.2.2 The numeric category [lib.category.numeric]

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 extraction of smaller types to extractors for larger types, but are not required to do so.

2 The functions take a locale argument because their base class implementation relies on numpunct<> members to identify all numeric punctuation preferences, and on ctype<> members to perform character classification.

3 Extractor and inserter members of the standard iostreams are required to use num_get<> and num_put<> member functions for formatting and parsing (27.6.1.2.1, 27.6.2.4.1). The ios& argument is used both for format control, and to report errors, as described in subclauses 27.4.4.3 and 27.4.3.2.

22.2.2.1 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;
    explicit num_get(size_t refs = 0);

    iter_type get(iter_type in, iter_type end, ios&,
                  const locale&, bool& v)          const;
    iter_type get(iter_type in, iter_type end, ios& ,
                  const locale&, long& v)          const;
    iter_type get(iter_type in, iter_type end, ios&,
                  const locale&, unsigned long& v) const;
    iter_type get(iter_type in, iter_type end, ios&,
                  const locale&, double& v)        const;
    iter_type get(iter_type in, iter_type end, ios&,
                  const locale&, long double& v)   const;

    static locale::id id;

  protected:
   ~num_get();  // virtual
    virtual iter_type do_get(iter_type, iter_type, ios&, const locale&,
                             bool& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios&, const locale&,
                             long& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios&, const locale&,
                             unsigned long& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios&, const locale&,
                             double& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios&, const locale&,
                             long double& v) const;
  };
}
```

1 The facet `num_get` is used to parse numeric values from an input sequence such as an istream.

22.2.2.1.1 `num_get` members [lib.facet.num.get.members]

```
iter_type get(iter_type in, iter_type end, ios& str
              const locale& loc, bool& val) const;
iter_type get(iter_type in, iter_type end, ios& str
              const locale& loc, long& val) const;
iter_type get(iter_type in, iter_type end, ios& str
              const locale& loc, unsigned long& val) const;
iter_type get(iter_type in, iter_type end, ios& str
              const locale& loc, double& val) const;
iter_type get(iter_type in, iter_type end, ios& str
              const locale& loc, long double& val) const;
```

Returns: `do_get(`*`in`*`, `*`end`*`, `*`str`*`, `*`loc`*`, `*`val`*`)`.

22.2.2.1.2 `num_get` virtual functions [lib.facet.num.get.virtuals]

```
iter_type do_get(iter_type in, iter_type end, ios& str
                 const locale& loc, bool& val) const;
iter_type do_get(iter_type in, iter_type end, ios& str
                 const locale& loc, long& val) const;
iter_type do_get(iter_type in, iter_type end, ios& str
                 const locale& loc, unsigned long& val) const;
iter_type do_get(iter_type in, iter_type end, ios& str
                 const locale& loc, double& val) const;
iter_type do_get(iter_type in, iter_type end, ios& str
                 const locale& loc, long double& val) const;
```

Effects: Reads characters from *in*, interpreting them according to *str*`.flags()`, *loc*`.use template< ctype<charT> >`, and *loc*`.use template< numpunct<charT> >`. `do_get()` ignores the value of *str*`.rdstate()`; however, indicates failure by calling *str*`.setstate(failbit)` (which may throw `ios_base::failure` (27.4.4.3)).
If an error occurs, *val* is unchanged; otherwise it is set to the resulting value.

Notes: Digit group separators are optional; if present, digit grouping is checked after the entire number is read. When reading a non-numeric boolean value, the names are compared exactly.

Returns: An iterator pointing one past the last character consumed as part of the converted field.

22.2.2.2 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;

    explicit num_put(size_t refs = 0);

    iter_type put(iter_type s, ios& f, const locale& loc, bool v)         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, double v) const;
    iter_type put(iter_type s, ios& f, const locale& loc, long double v) const;

    static locale::id id;

  protected:
   ~num_put();  // virtual
    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&, double v) const;
    virtual iter_type do_put(iter_type, ios&, const locale&, long double v) const;
  };
}
```

1 The facet `num_put` is used to format numeric values to a character sequence such as an ostream.

22.2.2.2.1 `num_put` members [lib.facet.num.put.members]

```
iter_type put(iter_type out, ios& str
              const locale& loc, bool val) const;
iter_type put(iter_type out, ios& str
              const locale& loc, long val) const;
iter_type put(iter_type out, ios& str
              const locale& loc, unsigned long val) const;
iter_type put(iter_type out, ios& str
              const locale& loc, double val) const;
iter_type put(iter_type out, ios& str
              const locale& loc, long double val) const;
```

Returns: `do_put(`*out*`, `*str*`, `*loc*`, `*val*`)`.

22.2.2.2.2 `num_put` virtual functions [lib.facet.num.put.virtuals]

```
iter_type do_put(iter_type out, ios& str
                 const locale& loc, bool val) const;
iter_type do_put(iter_type out, ios& str
                 const locale& loc, long val) const;
iter_type do_put(iter_type out, ios& str
                 const locale& loc, unsigned long val) const;
iter_type do_put(iter_type out, ios& str
                 const locale& loc, double val) const;
iter_type do_put(iter_type out, ios& str
                 const locale& loc, long double val) const;
```

Effects: Writes characters to the sequence *out*, formatting *val* according to *str*`.flags()`, *loc*`.use template< ctype<charT> >`, and *loc*`.use template< numpunct<charT> >`. Inserts digit group separators as specified by `numpunct<charT>::do_grouping`.
Notes: `do_put()` ignores and does not change the result of *str*`.rdstate()` (27.4.3).
Returns: An iterator pointing immediately after the last character produced.

22.2.3 The numeric punctuation facet [lib.facet.numpunct]

22.2.3.1 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;

    explicit numpunct(size_t refs = 0);

    string decimal_point()   const;
    string thousands_sep()   const;
    vector<char>  grouping() const;
    string truename()        const;
    string falsename()       const;

    static locale::id id;
  protected:
   ~numpunct();  // virtual
    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
  };
}
```

1 `numpunct<>` specifies numeric punctuation. The base class provides classic "`C`" numeric formats, while the "..._`byname`" version supports named locale (e.g. POSIX, X/Open) numeric formatting semantics.

2 The syntax for number formats is as follows, where `digit` represents the radix set specified by the `fmtflags` argument value, `whitespace` is as determined by the facet `ctype<charT>` (22.2.1.1), and `thousands-sep` and `decimal-point` are the results of corresponding `numpunct<charT>` members. Integer values have the format:

```
  integer   ::= [sign] units
  sign      ::= plusminus [whitespace]
  plusminus ::= '+' | '-'
  units     ::= digits [thousands-sep units]
  digits    ::= digit [digits]
```

and floating-point values have:

```
  floatval ::= [sign] units [decimal-point [digits]] [e [sign] digits] |
               [sign]        decimal-point  digits   [e [sign] digits]
  e        ::= 'e' | 'E'
```

where the number of digits between `thousands-sep`s is as specified by `do_grouping()`. For parsing, if the `digits` portion contains no thousands-separators, no grouping constraint is applied.

22.2.3.1.1 `numpunct` members [lib.facet.numpunct.members]

```
string decimal_point() const;
```

Returns: `do_decimal_point()`

```
string thousands_sep() const;
```

Returns: `thousands_sep()`

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

Returns: `do_grouping()`

```
string truename()  const;
string falsename() const;
```

Returns: `do_truename()` or `do_falsename()`, respectively.

22.2.3.1.2 `numpunct` virtual functions [lib.facet.numpunct.virtuals]

```
string do_decimal_point() const;
```

Returns: A `basic_string<charT>` for use as the decimal radix separator. If this is not a one-character string, `num_get<charT,InputIterator>` is not required to recognize numbers formatted using it.

The base class implementation returns `"."`.

```
string do_thousands_sep() const;
```

Returns: A `basic_string<charT>` for use as the digit group separator. If this is longer than one character, `num_get<charT,InputIterator>` is not required to recognize numbers formatted with it.
The base class implementation returns the empty string.

```
vector<char> do_grouping() const;
```

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

```
string do_truename()  const;
string do_falsename() const;
```

Returns: A string representing the name of the boolean value `true` or `false`, respectively.
In the base class implementation these names are `"true"` and `"false"`.

22.2.3.2 Template class `numpunct_byname` [lib.locale.numpunct.byname]

```
namespace std {
  template <class charT>
  class numpunct_byname : public numpunct<charT> {
    // this class is specialized for char and wchar_t.
  public:
    explicit numpunct_byname(const char*, size_t refs = 0);
  protected:
   ~numpunct_byname();  // virtual
    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
  };
}
```

22.2.4 The collate category [lib.category.collate]

22.2.4.1 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;

    explicit collate(size_t refs = 0);

    int compare(const charT* low1, const charT* high1,
                const charT* low2, const charT* high2) const;
    string transform(const charT* low, const charT* high) const;
    long hash(const charT* low, const charT* high) const;
    static locale::id id;

  protected:
   ~collate();  // virtual
    virtual int    do_compare(const charT* low1, const charT* high1,
                              const charT* low2, const charT* high2) const;
    virtual string do_transform(const charT* low, const charT* high) const;
    virtual long   do_hash     (const charT* low, const charT* high) const;
  };
}
```

1 The class `collate<charT>` provides features for use in the collation (comparison) and hashing of strings. A locale member function template, `operator()`, uses the collate facet to allow a locale to act directly as the predicate argument for standard algorithms (25) and containers operating on strings. The base class implementation applies lexicographic ordering (25.3.8).

2 Each function compares a string of characters `*p` in the range `[low, high)`.

22.2.4.1.1 `collate` members [lib.locale.collate.members]

```
int compare(const charT* low1, const charT* high1,
            const charT* low2, const charT* high2) const;
```

Returns: `do_compare(low1, high1, low2, high2)`

```
string transform(const charT* low, const charT* high) const;
```

Returns: `do_transform(low, high)`

```
long hash(const charT* low, const charT* high) const;
```

Returns: `do_hash(low, high)`

22.2.4.1.2 `collate` virtual functions [lib.locale.collate.virtuals]

```
int do_compare(const charT* low1, const charT* high1,
               const charT* low2, const charT* high2) const;
```

Returns: `1` if the first string is greater than the second, `-1` if less, zero otherwise.

```
string transform(const charT* low, const charT* high) const;
```

Returns: A `basic_string<charT>` value that, compared lexicographically with the result of calling `transform()` on another string, yields the same result as calling `compare()` on the same two strings.[179)]

```
long hash(const charT* low, const charT* high) const;
```

Returns: An integer value equal to the result of calling `hash()` on any other string for which `compare()` returns 0 (equal) when passed the two strings.

Notes: The probability that the result equals that for another string which does not compare equal should be very small, approaching `(2.0/numeric_limits<long>::max())` or less for longer strings.

179) This function is useful when one string is being compared to many other strings.

22.2.4.2 Template class `collate_byname` [lib.locale.collate.byname]

```
namespace std {
  template <class charT>
  class collate_byname : public collate<charT> {
  public:
    explicit collate_byname(const char*, size_t refs = 0);
  protected:
   ~collate_byname();  // virtual
    virtual int    do_compare(const charT* low1, const charT* high1,
                              const charT* low2, const charT* high2) const;
    virtual string do_transform(const charT* low, const charT* high) const;
    virtual long   do_hash(     const charT* low, const charT* high) const;
  };
```

22.2.5 The time category [lib.category.time]

1 The classes `time_get<charT,InputIterator>` and `time_put<charT,OutputIterator>` provide date and time formatting and parsing. The `ios&` argument is used both for format control, and to report errors, as described in subclauses 27.4.3.1.2 and 27.4.3.1.3.

22.2.5.1 Template class `time_get` [lib.locale.time.get]

```
namespace std {
  class time_base {
  public:
    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;

    explicit time_get(size_t refs = 0);

    dateorder date_order()  const { return do_date_order(); }
    iter_type get_time(iter_type s, iter_type end, ios& f,
                       const locale& loc, tm* t)  const;
    iter_type get_date(iter_type s, iter_type end, ios& f,
                       const locale& loc, tm* t)  const;
    iter_type get_weekday(iter_type s, iter_type end, ios& f,
                          const locale& loc, tm* t) const;
    iter_type get_monthname(iter_type s, iter_type end, ios& f,
                            const locale& loc, tm* t) const;
    iter_type get_year(iter_type s, iter_type end, ios& f,
                       const locale& loc, tm* t) const;

    static locale::id id;
  protected:
   ~time_get();  // virtual
    virtual dateorder do_date_order()  const;
    virtual iter_type do_get_time(iter_type s, iter_type end, ios&,
                                  const locale&, tm* t) const;
    virtual iter_type do_get_date(iter_type s, iter_type end, ios&,
                                  const locale&, tm* t) const;
    virtual iter_type do_get_weekday(iter_type s, iter_type end, 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, iter_type end, ios&,
                                  const locale&, tm* t) const;
  };
}
```

1 `time_get` is used to parse a character sequence, extracting components of a time or date into a `struct tm` record. Each `get` member parses a format as produced by a corresponding format specifier to `time_put<>::put`. If the sequence being parsed matches the correct format, the corresponding members of the `struct tm` argument are set to the values used to produce the sequence; otherwise either an error is reported or unspecified values are assigned.[180)]

22.2.5.1.1 `time_get` members [lib.locale.time.get.members]

```
dateorder date_order() const;
```

Returns: `do_date_order()`

```
iter_type get_time(iter_type s, iter_type end, ios& str,
                   const locale& loc, tm* t) const;
```

Returns: `do_get_time(`*s*`,` *end*`,` *str*`,` *loc*`,` *t*`)`

```
iter_type get_date(iter_type s, iter_type end, ios& str,
                   const locale& loc, tm* t) const;
```

Returns: `do_get_date(`*s*`,` *end*`,` *str*`,` *loc*`,` *t*`)`

```
iter_type get_weekday(iter_type s, iter_type end, ios& str,
                      const locale&loc, tm* t) const;
iter_type get_monthname(iter_type s, iter_type end, ios& str,
                        const locale& loc, tm* t) const;
```

Returns: `do_get_weekday(`*s*`,` *end*`,` *str*`,` *loc*`,` *t*`)` or `do_get_monthname(`*s*`,` *end*`,` *str*`,` *loc*`,` *t*

```
iter_type get_year(iter_type s, iter_type end, ios& str,
                   const locale& loc, tm* t) const;
```

180) In other words, user confirmation is required for reliable parsing of user-entered dates and times, but machine-generated formats can be parsed reliably. This allows parsers to be aggressive about interpreting user variations on standard formats.

Returns: `do_get_year(`*s*`,` *end*`,` *str*`,` *loc*`,` *t*`)`

22.2.5.1.2 `time_get` virtual functions [lib.locale.time.get.virtuals]

```
dateorder do_date_order() const;
```

Returns: An enumeration value indicating the preferred order of components for dates composed of day, month, and year.
Returns `no_order` if the date format specified by `'X'` contains other variable components (e.g Julian day, week number, week day).

```
iter_type do_get_time(iter_type s, iter_type end, ios& str, const locale&,
                      tm* t) const;
```

Effects: Reads characters starting at *s* until it has extracted those `struct tm` members, and remaining format characters, used to produce the format specified by `'X'`, or until it encounters an error or end of sequence.
Indicates an error by calling, *str*`.setstate(failbit)`, which may throw `ios_base::failure` (27.4.4.3)).
Returns: An iterator pointing immediately beyond the last character recognized as part of the time, if no error occurred.

```
iter_type do_get_date(iter_type s, iter_type end, ios& str, const locale&,
                      tm* t) const;
```

Effects: Reads characters starting at *s* until it has extracted those `struct tm` members, and remaining format characters, used to produce the format specified by `'x'`, or until it encounters an error.
Indicates failure by calling *str*`.setstate(failbit)` (which may throw `ios_base::failure` (27.4.4.3)).
Returns: An iterator pointing immediately beyond the last character recognized as part of the date, if no error occurred.

```
iter_type do_get_weekday(iter_type s, iter_type end, ios& str,
                         const locale&, tm* t) const;
iter_type do_get_monthname(iter_type s, iter_type end, ios& str,
                           const locale&, tm* t) const;
```

Effects: Reads characters starting at *s* until it has extracted the (perhaps abbreviated) name of a weekday or month. If it finds an abbreviation that is followed by characters that could match a full name, it continues reading until it matches the full name or fails. It sets the appropriate `struct tm` member accordingly.
Indicates failure by calling *str*`.setstate(failbit)` (which may throw `ios_base::failure` (27.4.4.3)).
Returns: An iterator pointing immediately beyond the last character recognized as part of a valid name.

```
iter_type do_get_year(iter_type s, iter_type end, ios& str,
                      const locale&, tm* t) const;
```

Effects: Reads characters starting at *s* until it has extracted an unambiguous year identifier. It is unspecified whether two-digit year numbers are accepted, or what century they are assumed to lie in. Sets the *t*`->tm_year` member accordingly.
Indicates failure by calling *str*`.setstate(failbit)` (which may throw `ios_base::failure` (27.4.4.3)).

Returns: An iterator pointing immediately beyond the last character recognized as part of a valid year identifier.

22.2.5.2 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> {
  public:
    explicit time_get_byname(const char*, size_t refs = 0);
  protected:
   ~time_get_byname();  // virtual
    virtual dateorder do_date_order()  const;
    virtual iter_type do_get_time(iter_type s, iter_type end, ios&,
                                  const locale&, tm* t) const;
    virtual iter_type do_get_date(iter_type s, iter_type end, ios&,
                                  const locale&, tm* t) const;
    virtual iter_type do_get_weekday(iter_type s, iter_type end, ios&,
                                     const locale&, tm* t) const;
    virtual iter_type do_get_monthname(iter_type s, iter_type end, ios&,
                                       const locale&, tm* t) const;
    virtual iter_type do_get_year(iter_type s, iter_type end, ios&,
                                  const locale&, tm* t) const;
  };
}
```

22.2.5.3 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;

    explicit time_put(size_t refs = 0);

      // the following is implemented in terms of other member functions.
    iter_type put(iter_type s, ios& f, const locale& loc, const tm* tmb,
                  const charT* pattern, const charT* pat_end) const;
    iter_type put(iter_type s, ios& f, const locale& loc,
                  const tm* t, char format, char modifier = 0) const;

    static locale::id id;

  protected:
   ~time_put();  // virtual
    virtual iter_type do_put(iter_type s, ios&, const locale&, const tm* t,
                             char format, char modifier) const;
  };
}
```

22.2.5.3.1 time_put members [lib.locale.time.put.members]

```
iter_type put(iter_type s, ios&, const locale&, const tm* t,
              const charT* pattern, const charT* pat_end) const;
iter_type put(iter_type s, ios&, const locale&, const tm* t,
              char format, char modifier = 0) const;
```

Effects: The first form interprets the characters between *pattern* and *pat_end* identically as `strftime()`, (though not treating the null character as a terminator), calling `do_put()` repeatedly as needed.

The second form calls `do_put()` once, simply passing along its arguments.

Returns: An iterator pointing immediately after the last character produced.

22.2.5.3.2 `time_put` virtual functions [lib.locale.time.put.virtuals]

```
iter_type do_put(iter_type s, ios&, const locale&, const tm* t,
                 char format, char modifier) const;
```

Effects: Formats the contents of the parameter *t* into characters placed on the output sequence *s*. Formatting is controlled by the parameters *format* and *modifier*, interpreted identically as the format specifiers in the string argument to the standard library function `strftime()`.[181)]

Returns: An iterator pointing immediately after the last character produced.

22.2.5.4 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>
  {
  public:
    explicit time_put_byname(const char*, size_t refs = 0);
  protected:
   ~time_put_byname();  // virtual
    virtual iter_type do_put(iter_type s, ios&, const locale&, const tm* t,
                             char format, char modifier) const;
  };
}
```

22.2.6 The monetary category [lib.category.monetary]

1 These templates handle monetary formats. A template parameter indicates whether local or international monetary formats are to be used. `money_get<>` and `money_put<>` use `moneypunct<>` members to determine all formatting details. `moneypunct<>` provides basic format information for money processing. The `ios&` argument is used both for format control, and to report errors, as described in subclauses 27.4.3.1.2 and 27.4.3.1.3.

22.2.6.1 Template class `money_get` [lib.locale.money.get]

181) Interpretation of the *modifier* argument is implementation-defined, but should follow POSIX conventions.

```
namespace std {
  template <class charT, bool Intl = false,
            class InputIterator = istreambuf_iterator<charT> >
  class money_get : public locale::facet {
  public:
    typedef charT               char_type;
    typedef InputIterator       iter_type;
    typedef basic_string<charT> string;
    typedef basic_ios<charT>    ios;

    explicit money_get(size_t refs = 0);

    iter_type get(iter_type s, iter_type end, ios& f,
                  const locale& loc, double& units) const;
    iter_type get(iter_type s, iter_type end, ios& f,
                  const locale& loc, string& digits) const;

    static locale::id id;

  protected:
   ~money_get();  // virtual
    virtual iter_type do_get(iter_type, iter_type, ios&, const locale&,
                             double& units) const;
    virtual iter_type do_get(iter_type, iter_type, ios&, const locale&,
                             string& digits) const;
  };
}
```

22.2.6.1.1 money_get members [lib.locale.money.get.members]

```
iter_type get(iter_type s, iter_type end, ios& f,
              const locale& loc, double& quant) const;
iter_type get(s, iter_type end, ios&f,
              const locale& loc, string& quant) const;
```

Returns: `do_get(s, end, f, loc, quant)`

22.2.6.1.2 money_get virtual functions [lib.locale.money.get.virtuals]

```
iter_type do_get(iter_type s, iter_type end, ios& str,
                 const locale& loc, double& units) const;
iter_type do_get(iter_type s, iter_type end, ios& strfP,
                 const locale& loc, string& digits) const;
```

Effects: Reads characters from *s* until it has constructed a monetary value, as specified in *str*`.flags()` and the `moneypunct<charT>` facet of *loc*, or until it encounters an error or runs out of characters. The result is a pure sequence of digits, representing a count of the smallest unit of currency representable.[182] Digit group separators are optional; if present, digit grouping is checked after all syntactic elements have been read. Where `space` or `none` appear in the format pattern, except at the end, optional whitespace is consumed. Sets the argument *units* or *digits* from the sequence of digits found. *units* is negated, or *digits* is preceded by `'-'`, for a negative value.
Indicates failure by calling *str*`.setstate(failbit)` (which may throw `ios_base::failure` (27.4.4.3)).
On error, the *units* or *digits* argument is unchanged.

[182] For example, the sequence `$1,056.23` in a common U.S. locale would yield, for *units*, 105623, or for *digits*, “`105623`”.

Returns: An iterator pointing immediately beyond the last character recognized as part of a valid monetary quantity.

22.2.6.2 Template class `money_put` [lib.locale.money.put]

```
namespace 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;

    explicit money_put(size_t refs = 0);

    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& digits) const;

    static locale::id id;
    static const bool intl = Intl;

  protected:
   ~money_put();  // virtual
    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;
  };
}
```

22.2.6.2.1 `money_put` members [lib.locale.money.put.members]

```
iter_type put(iter_type s, ios& f, const locale& loc,
              double quant) const;
iter_type put(iter_type s, ios& f, const locale& loc,
              const string& quant) const;
```

Returns: `do_put(`*s*`, `*f*`, `*loc*`, `*quant*`)`

22.2.6.2.2 `money_put` virtual functions [lib.locale.money.put.virtuals]

```
iter_type do_put(iter_type s, ios& str, const locale& loc,
                 double units) const;
iter_type do_put(iter_type s, ios& str, const locale& loc,
                 const string& digits) const;
```

Effects: Writes characters to *s*, according to the format specified by the `moneypunct<charT>` facet of *loc*, and *str*`.flags()`. Ignores any fractional part of *units*, or any characters in *digits* beyond the (optional) leading `'-'` and immediately subsequent digits.

Notes: The currency symbol is generated only if `(`*str*`.flags() & ios::showbase)` is true. If `((`*str*`.flags() & ios::adjustfield) == ios::internal)` the fill characters are placed where `none` or `space` appears in the formatting pattern (_lib.money.get.virtuals_).

Returns: An iterator pointing immediately after the last character produced.

22.2.6.3 Template class `moneypunct` [lib.locale.moneypunct]

```
namespace std {
  class money_base {
  public:
    enum part { none, space, symbol, sign, value };
    struct pattern { char field[4]; };
  };

  template <class charT, bool International = false>
  class moneypunct : public locale::facet, public money_base {
  public:
    typedef charT char_type;
    typedef basic_string<charT> string;

    explicit moneypunct(size_t refs = 0);

    charT        decimal_point() const;
    charT        thousands_sep() const;
    vector<char> grouping()      const;
    string       curr_symbol()   const;
    string       positive_sign() const;
    string       negative_sign() const;
    int          frac_digits()   const;
    pattern      pos_format()    const;
    pattern      neg_format()    const;

    static locale::id id;
    static const bool intl = International;

  protected:
   ~moneypunct();  // virtual
    virtual charT        do_decimal_point() const;
    virtual charT        do_thousands_sep() const;
    virtual 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;
  };
}
```

1 This provides money punctuation, similar to `numpunct<>` above (22.2.3.1). In particular, the `value` portion of the format is:

```
  value ::= units [decimal-point [digits]] |
            decimal-point digits
```

if `frac_digits` returns a positive value, or just

```
  value ::= units
```

otherwise. In these forms, the `decimal-point` and `thousands-separator` are as determined below and the number of digits after the decimal point is exactly the value returned by `frac_digits`.

22.2.6.3.1 `moneypunct` members [lib.locale.moneypunct.members]

```
charT        decimal_point() const;
charT        thousands_sep() const;
vector<char> grouping()      const;
string       curr_symbol()   const;
string       positive_sign() const;
string       negative_sign() const;
int          frac_digits()   const;
pattern      pos_format()    const;
pattern      neg_format()    const;
```

1 Each of these functions `F` returns the result of calling the corresponding virtual member function `do_`***F***`()`.

22.2.6.3.2 `moneypunct` virtual functions [lib.locale.moneypunct.virtuals]

```
charT do_decimal_point() const;
```

Returns: The radix separator to use in case `do_frac_digits()` is greater than zero.[183)]

```
charT do_thousands_sep() const;
```

Returns: The digit group separator to use in case `do_grouping()` specifies a digit grouping pattern.[184)]

```
vector<char> do_grouping() const;
```

Returns: A pattern defined identically as the result of `numpunct<charT>::do_grouping()`.[185)]

```
string do_curr_symbol() const;
```

Returns: A string to use as the currency identifier symbol.[186)]

```
string do_positive_sign() const;
```

Returns: The string to use to indicate a positive monetary value.[187)]

```
string do_negative_sign() const;
```

Returns: The string to use to indicate a negative monetary value.
Notes: If it is a one-character string containing `'('`, it is paired with a matching `')'`.

```
int do_frac_digits() const;
```

Returns: The number of digits after the decimal radix separator, if any.[188)]

183) In common U.S. locales this is `'.'`.
184) In common U.S. locales this is `','`.
185) This is most commonly the vector `"{ 3 }"`
186) For international instantiations (second template parameter `true`) this is always four characters long, usually three letters and a space.
187) This is usually the empty string.
188) In common U.S. locales, this is 2.

```
pattern do_pos_format() const;
pattern do_neg_format() const;
```

Returns: A `pattern`, a four-element array specifying the order in which syntactic elements appear in the monetary format.

Notes: In this array each enumeration value `symbol`, `sign`, `value`, and either `space` or `none` appears exactly once. `none`, if present, is not first; `space`, if present, is neither first nor last. Otherwise, the elements may appear in any order. In international instantiations, the result is always `{ symbol, sign, none, value }`.[189)]

22.2.6.4 Template class `moneypunct_byname` [lib.locale.moneypunct.byname]

```
namespace std {
  template <class charT, bool Intl = false>
  class moneypunct_byname : public moneypunct<charT, Intl> {
  public:
    explicit moneypunct_byname(const char*, size_t refs = 0);
  protected:
   ~moneypunct_byname();  // virtual
    virtual charT        do_decimal_point() const;
    virtual charT        do_thousands_sep() const;
    virtual 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;
  };
}
```

22.2.7 The message retrieval category [lib.category.messages]

1 Class `messages<charT>` implements retrieval of strings from message catalogs.

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

```
namespace std {
  class messages_base {
  public:
    typedef THE_POSIX_CATALOG_IDENTIFIER_TYPE catalog;
  };

  template <class charT>
  class messages : public locale::facet, public messages_base {
  public:
    typedef charT char_type;
    typedef int   catalog;
    typedef basic_string<charT> string;

    explicit messages(size_t refs = 0);

    catalog open (const basic_string<char>& fn, const locale&) const;
    string  get  (catalog c, int set, int msgid, const string& dfault) const;
    void    close(catalog c) const;
```

189) Note that the international symbol usually contains a space, itself; for example, `"USD "`.

```
    static locale::id id;

  protected:
   ~messages();  // virtual
    virtual catalog do_open(const basic_string<char>&, const locale&) const;
    virtual string  do_get(catalog, int set, int msgid,
                           const string& dfault) const;
    virtual void    do_close(catalog) const;
  };
}
```

22.2.7.1.1 `messages` members [lib.locale.messages.members]

```
catalog open(const basic_string<char>& name, const locale& loc) const;
```

Returns: `do_open(`*`name`*`, `*`loc`*`)`.

```
string get(catalog cat, int set, int msgid, const string& dfault) const;
```

Returns: `do_get(`*`cat`*`, `*`set`*`, `*`msgid`*`, `*`dfault`*`)`.

```
void  close(catalog cat) const;
```

Effects: Calls `do_close(`*`cat`*`)`.

22.2.7.1.2 `messages` virtual functions [lib.locale.messages.virtuals]

```
catalog do_open(const basic_string<char>& name,
                const locale& loc) const;
```

Returns: A value that may be passed to `get()` to retrieve a message, from the message catalog identified by the string *`name`* according to an implementation-defined mapping. The result can be used until it is passed to `close()`.
Returns a value less than 0 if no such catalog can be opened.
Notes: The locale argument *`loc`* is used for character set code conversion when retrieving messages, if needed.

```
string do_get(catalog cat, int set, int msgid,
              const string& dfault) const;
```

Requires: A catalog *`cat`* obtained from `open()` and not yet closed.
Returns: A message identified by arguments *`set`*, *`msgid`*, and *`dfault`*, according to an implementation-defined mapping. If no such message can be found, returns *`dfault`*.

```
void do_close(catalog cat) const;
```

Requires: A catalog *`cat`* obtained from `open()` and not yet closed.
Effects: Releases unspecified resources associated with *`cat`*.
Notes: The limit on such resources, if any, is implementation-defined.

22.2.7.2 Template class `messages_byname` [lib.locale.messages.byname]

```
namespace std {
  template <class charT>
  class messages_byname : public messages<charT> {
  public:
    explicit messages_byname(const char*, size_t refs = 0);
  protected:
   ~messages_byname();  // virtual
    virtual catalog do_open(const basic_string<char>&, const locale&) const;
    virtual string  do_get(catalog, int set, int msgid,
                           const string& dfault) const;
    virtual void    do_close(catalog) const;
  };
}
```

22.2.8 Program-defined facets [lib.facets.examples]

1 A C++ program may define facets to be added to a locale and used identically as the built-in facets. To create a new facet interface, C++ programs simply derive from `locale::facet` a class containing a static member: `static locale::id id`.

2 [*Note:* The locale member function templates verify its type and storage class. —*end note*]

3 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.

4 [*Example:* Here is a program that just calls C functions:

```
#include <locale>
extern "C" void c_function();
int main()
{
  using namespace std;
  locale::global(locale(""));  // same as setlocale(LC_ALL, "");
  c_function();
  return 0;
}
```

In other words, C library localization is unaffected. —*end example*]

5 [*Example:* 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();
}
```

—*end example*]

6 [*Example:* 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);
}
```

In a European locale, with input `3.456,78`, output is `3456.78`. —*end example*]

7 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 [*Example:* Here is an example of the use of locales in a library interface.

```
// 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);
...
```

This example illustrates two architectural uses of class `locale`.

9 The first is as a default argument in `Date::asString()`, where the default is the global (presumably user-preferred) locale.

10 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.getloc();
  struct tm t;
  loc.template use<time_get<char> >().get_date(s, s, 0, loc, &t);
  if (s) d = Date(t.tm_day, t.tm_mon + 1, t.tm_year + 1900);
  s.isfx();
  return s;
}
```

—*end example*]

11 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 a C++ program must define is the static member `id`, which identifies your class interface as a new facet.

12 [*Example:* Classifying Japanese characters:

```
// file: <jctype>
#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.template use<ctype>().widen('!');
  if (loc.template use<My::JCType>().is_kanji(c))
    cout << "no it isn't!" << endl;
  return 0;
}
```

13 The new facet is used exactly like the built-in facets. —*end example*]

14 [*Example:* 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;
  }
```

—*end example*]

22.3 C Library Locales [lib.c.locales]

1 Header `<clocale>` (Table 48):

Table 48—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 49:

Table 49—Containers library summary

Subclause	Header(s)
23.1 Requirements	
23.2 Sequences	`<bitset>` `<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 50, `X` denotes a container class containing objects of type `T`, `a` and `b` denote values of `X`, `u` denotes an identifier and `r` denotes a value of `X&`.

Table 50—Container requirements

expression	return type	assertion/note pre/post-condition	complexity
`X::value_type`	`T`		compile time
`X::reference`	lvalue of `T`		compile time
`X::const_reference`	const lvalue of `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
`a.swap(b);`	`void`	`swap(a,b)`	constant

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.

5 Constructors for all container types defined in this clause take an `Allocator&` argument. A copy of this argument is used for any memory allocation performed, by these constructors and by all member functions, during the lifetime of each container object.

6 If the iterator type of a container belongs to the bidirectional or random access iterator categories (24.1), the container is called *reversible* and satisfies the additional requirements in the following Table 51:

Table 51—Reversible container requirements

expression	return type	assertion/note pre/post-condition	complexity
`X::reverse_iterator`	iterator type pointing to `T`	`reverse_iterator<iterator, value_type, reference, difference_type>` for random access iterator, `reverse_bidirectional_iterator<iterator, value_type, reference, difference_type>` for bidirectional iterator.	compile time
`X::const_reverse_iterator`	iterator type pointing to `const T`	`reverse_iterator< const_iterator, value_type, const_reference, difference_type>` for random access iterator, `reverse_bidirectional_iterator<const_iterator, value_type, const_reference, difference_type>` for bidirectional iterator.	compile time
`a.rbegin()`	`reverse_iterator`; `const_reverse_iterator` for constant `a`	`reverse_iterator(end())`	constant
`a.rend()`	`reverse_iterator`; `const_reverse_iterator` for constant `a`	`reverse_iterator(begin())`	constant

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 52, `X` denotes a sequence class, `a` denotes value of `X`, `i` and `j` denote iterators satisfying input iterator requirements, `[i, j)` denotes a valid range, `n` denotes a value of `X::size_type`, `p` denotes a valid iterator to `a`, `q`, `q1`, `q2` denote valid dereferenceable iterators to `a`, `[q1, q2)` denotes a valid range, `t` denotes a value of `X::value_type`.

3 The complexities of the expressions are sequence dependent.

Table 52—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 `vector`, `list`, 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 53:

Table 53—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 The phrase ‘‘equality of keys’’ means 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 54, `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 54—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.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.	`Nlog(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 `<bitset>`, `<deque>`, `<list>`, `<queue>`, `<stack>`, and `<vector>`.

Header `<bitset>` 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 bitset;

  // 23.2.1.3 bitset operations:
  template <size_t N> bitset<N> operator&(const bitset<N>&, const bitset<N>&);
  template <size_t N> bitset<N> operator|(const bitset<N>&, const bitset<N>&);
  template <size_t N> bitset<N> operator^(const bitset<N>&, const bitset<N>&);
  template <size_t N> istream&  operator>>(istream& is, bitset<N>& x);
  template <size_t N> ostream&  operator<<(ostream& os, const bitset<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>
    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);
  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 `bitset` [lib.template.bitset]

1 The header `<bitset>` defines a template class and several related functions for representing and manipulating fixed-size sequences of bits.

```
namespace std {
  template<size_t N> class bitset {
  public:
  // bit reference:
    class reference {
    public:
     ~reference();
      reference& operator=(bool x);           // for b[i] = x;
      reference& operator=(const reference&); // for b[i] = b[j];
      bool operator~() const;                 // for x = b[i];
      operator bool() const;                  // for b[i].flip();
      reference& flip();                      // flips the bit
    };

  // 23.2.1.1 constructors:
    bitset();
    bitset(unsigned long val);
    explicit bitset(const string& str, size_t pos = 0, size_t n = size_t(-1));
  // 23.2.1.2 bitset operations:
    bitset<N>& operator&=(const bitset<N>& rhs);
    bitset<N>& operator|=(const bitset<N>& rhs);
    bitset<N>& operator^=(const bitset<N>& rhs);
    bitset<N>& operator<<=(size_t pos);
    bitset<N>& operator>>=(size_t pos);
    bitset<N>& set();
    bitset<N>& set(size_t pos, int val = 1);
    bitset<N>& reset();
    bitset<N>& reset(size_t pos);
    bitset<N>  operator~() const;
    bitset<N>& flip();
    bitset<N>& flip(size_t pos);

  // element access:
    reference operator[](size_t pos);   // for b[i];

    unsigned long  to_ulong() const;
    string to_string() const;
    size_t count() const;
    size_t size()  const;
    bool operator==(const bitset<N>& rhs) const;
    bool operator!=(const bitset<N>& rhs) const;
    bool test(size_t pos) const;
    bool any() const;
    bool none() const;
    bitset<N> operator<<(size_t pos) const;
    bitset<N> operator>>(size_t pos) const;
  private:
//  char array[N];        exposition only
  };
}
```

2 The template class `bitset<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 `bitset<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.

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` (19.1.4);

— an *out-of-range* error is associated with exceptions of type `out_of_range` (19.1.6);

— an *overflow* error is associated with exceptions of type `overflow_error` (19.1.9).

23.2.1.1 `bitset` constructors [lib.bitset.cons]

```
bitset();
```

Effects: Constructs an object of class `bitset<N>`, initializing all bits to zero.

190) An implementation is free to store the bit sequence more efficiently.

```
bitset(unsigned long val);
```

Effects: Constructs an object of class `bitset<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)`.[191)]
If `M < N`, remaining bit positions are initialized to zero.

```
explicit bitset(const string& str, size_t pos = 0, size_t n = size_t(-1));
```

Requires: `pos <= str.size()`.
Throws: `out_of_range` if `pos > str.size()`.
Effects: Determines the effective length *`rlen`* of the initializing string as the smaller of *`n`* and `str.size() - 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 `bitset<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 `bitset` members [lib.bitset.members]

```
bitset<N>& operator&=(const bitset<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`.

```
bitset<N>& operator|=(const bitset<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`.

```
bitset<N>& operator^=(const bitset<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`.

```
bitset<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`.

191) The macro `CHAR_BIT` is defined in `<climits>` (18.2).

Returns: `*this`.

```
bitset<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`.

```
bitset<N>& set();
```

Effects: Sets all bits in `*this`.
Returns: `*this`.

```
bitset<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`.

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

Effects: Resets all bits in `*this`.
Returns: `*this`.

```
bitset<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`.

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

Effects: Constructs an object *`x`* of class `bitset<N>` and initializes it with `*this`.
Returns: *`x`*`.flip()`.

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

Effects: Toggles all bits in `*this`.
Returns: `*this`.

```
bitset<N>& flip(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`.

```
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*.

```
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.

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

Returns: A count of the number of bits set in `*this`.

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

Returns: `N`.

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

Returns: A nonzero value if the value of each bit in `*this` equals the value of the corresponding bit in *rhs*.

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

Returns: A nonzero value if `!(*this ==` *rhs*`)`.

```
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.

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

Returns: `true` if any bit in `*this` is one.

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

Returns: `true` if no bit in `*this` is one.

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

Returns: `bitset<N>(*this) <<=` *pos*.

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

Returns: `bitset<N>(*this) >>=` *pos*.

23.2.1.3 `bitset` operators [lib.bitset.operators]

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

Returns: `bitset<N>(lhs) &= pos`.

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

Returns: `bitset<N>(lhs) |= pos`.

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

Returns: `bitset<N>(lhs) ^= pos`.

```
template <size_t N>
  istream& operator>>(istream& is, bitset<N>& x);
```

1 A formatted input function (27.6.1.2).
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 = bitset<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)` (which may throw `ios_base::failure` (27.4.4.3).
Returns: *is*.

```
template <size_t N> ostream& operator<<(ostream& os, const bitset<N>& x);
```

Returns: `os << x.to_string()` (27.6.2.4).

23.2.2 Template class `deque` [lib.deque]

1 A `deque` is a kind of sequence that, like a `vector` (23.2.5), 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:
  // 23.2.2.1 types:
    typedef typename Allocator::types<T>::reference       reference;
    typedef typename Allocator::types<T>::const_reference const_reference;
    typedef typename Allocator::types<T>::pointer         iterator;
    typedef typename Allocator::types<T>::const_pointer   const_iterator;
    typedef typename Allocator::size_type                 size_type;
    typedef typename Allocator::difference_type           difference_type;
    typedef T value_type;
    typedef reverse_iterator<iterator, value_type,
                 reference, difference_type>              reverse_iterator;
    typedef reverse_iterator<const_iterator, value_type,
                 const_reference, difference_type>        const_reverse_iterator;

  // 23.2.2.2 construct/copy/destroy:
    explicit deque(Allocator& = Allocator());
    explicit deque(size_type n, const T& value = T(), Allocator& = Allocator());
    deque(const deque<T,Allocator>& x, Allocator& = Allocator());
    template <class InputIterator>
      deque(InputIterator first, InputIterator last, Allocator& = Allocator());
   ~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());

  // 23.2.2.3 iterators:
    iterator               begin();
    const_iterator         begin() const;
    iterator               end();
    const_iterator         end() const;
    reverse_iterator       rbegin();
    const_reverse_iterator rbegin() const;
    reverse_iterator       rend();
    const_reverse_iterator rend() const;

  // 23.2.2.4 capacity:
    size_type size() const;
    size_type max_size() const;
    void      resize(size_type sz, T c = T());
    bool      empty() const;

  // 23.2.2.5 element access:
    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;

  // 23.2.2.6 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);
    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);
    void swap(deque<T,Allocator>&);
  };

  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 types [lib.deque.types]

23.2.2.2 deque constructors, copy, and assignment [lib.deque.cons]

```
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);
```

23.2.2.3 deque iterator support [lib.deque.iterators]

23.2.2.4 deque capacity [lib.deque.capacity]

```
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
    ; // do nothing
```

23.2.2.5 deque element access [lib.deque.access]

23.2.2.6 deque modifiers [lib.deque.modifiers]

```
iterator insert(iterator position, const T& x = T());
void     insert(iterator position, size_type n, const T& x);
template <class InputIterator>
  void insert(iterator position,
              InputIterator first, InputIterator last);
```

Effects: Invalidates all the iterators and references to the deque.

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`.

```
void erase(iterator position);
void erase(iterator first, iterator last);
```

Effects: Invalidates all the iterators and references to the deque.

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.

23.2.3 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 (23.2.5) 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:
  // 23.2.3.1 types:
    typedef typename Allocator::types<T>::reference       reference;
    typedef typename Allocator::types<T>::const_reference const_reference;
    typedef typename Allocator::types<T>::pointer         iterator;
    typedef typename Allocator::types<T>::const_pointer   const_iterator;
    typedef typename Allocator::size_type       size_type;
    typedef typename Allocator::difference_type difference_type;
    typedef T value_type;
    typedef reverse_iterator<iterator, value_type,
                 reference, difference_type>             reverse_iterator;
    typedef reverse_iterator<const_iterator, value_type,
                 const_reference, difference_type>       const_reverse_iterator;
  // 23.2.3.2 construct/copy/destroy:
    explicit list(Allocator& = Allocator());
    explicit list(size_type n, const T& value = T(),
                  Allocator& = Allocator());
    template <class InputIterator>
      list(InputIterator first, InputIterator last,
           Allocator& = Allocator());
    list(const list<T,Allocator>& x, Allocator& = Allocator());
   ~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());

  // 23.2.3.3 iterators:
    iterator               begin();
    const_iterator         begin() const;
    iterator               end();
    const_iterator         end() const;
    reverse_iterator       rbegin();
    const_reverse_iterator rbegin() const;
    reverse_iterator       rend();
    const_reverse_iterator rend() const;

  // 23.2.3.4 capacity:
    bool      empty() const;
    size_type size() const;
    size_type max_size() const;
    void      resize(size_type sz, T c = T());

  // element access:
    reference       front();
    const_reference front() const;
    reference       back();
    const_reference back() const;

  // 23.2.3.6 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);
    template <class InputIterator>
      void insert(iterator position, InputIterator first,
                  InputIterator last);

    void erase(iterator position);
    void erase(iterator position, iterator last);
    void swap(list<T,Allocator>&);

  // 23.2.3.7 list operations:
    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` types [lib.list.types]

23.2.3.2 `list` constructors, copy, and assignment [lib.list.cons]

```
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);
```

23.2.3.3 `list` iterator support [lib.list.iterators]

23.2.3.4 `list` capacity [lib.list.capacity]

```
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
    ; // do nothing
```

23.2.3.5 `list` element access [lib.list.access]

23.2.3.6 `list` modifiers [lib.list.modifiers]

```
iterator insert(iterator position, const T& x = T());
void     insert(iterator position, size_type n, const T& x);
template <class InputIterator>
  void insert(iterator position, InputIterator first,
              InputIterator last);
```

Notes: 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.

```
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.

23.2.3.7 `list` operations [lib.list.ops]

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.

```
void splice(iterator position, list<T,Allocator>& x);
```

Requires: `&x != this`.

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`. The result is unchanged is `position == i` or `position == ++i`.

Requires: `i` is a valid dereferenceable 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`. The result is undefined if `position` is an iterator in the range `[first, last)`.

Complexity: Constant time if `&x == this`; otherwise, linear time.

```
                           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.

```
                                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.

```
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.

```
void reverse();
```

Effects: Reverses the order of the elements in the list.
Complexity: Linear time.

```
                         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 Container adapters [lib.container.adapters]

23.2.4.1 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 typename Container::value_type value_type;
    typedef typename 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);
  template <class Container>
    bool operator< (const queue<Container>& x, const queue<Container>& y);
}
```

`operator==`

Returns: `x.c == y.c`.

`operator<`

Returns: `x.c < y.c`.

23.2.4.2 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` (23.2.5) and `deque` (23.2.2) can be used.

```
namespace std {
  template <class Container, class Compare = less<Container::value_type> >
  class priority_queue {
  public:
    typedef typename Container::value_type value_type;
    typedef typename Container::size_type  size_type;
  protected:
    Container c;
    Compare comp;

  public:
    explicit 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
}
```

23.2.4.2.1 `priority_queue` constructors [lib.priqueue.cons]

```
priority_queue(const Compare& x = Compare());
```

Effects: Initializes `comp` with *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);
    }
```

23.2.4.2.2 `priority_queue` members [lib.priqueue.members]

```
void push(const value_type& x);
```

Effects:

```
      c.push_back(x);
      push_heap(c.begin(), c.end(), comp);
```

```
void pop();
```

Effects:

```
      pop_heap(c.begin(), c.end(), comp);
      c.pop_back();
```

23.2.4.3 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` (23.2.5), `list` (23.2.3) and `deque` (23.2.2) can be used.

2 [*Example:* `stack<vector<int> >` is an integer stack made out of `vector`, and `stack<deque<char> >` is a character stack made out of `deque`. —*end example*]

```
namespace std {
  template <class Container>
  class stack {
  public:
    typedef typename Container::value_type value_type;
    typedef typename 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);
  template <class Container>
    bool operator< (const stack<Container>& x, const stack<Container>& y);
}
```

```
operator==
```

Returns: `x.c == y.c`.

23.2.5 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:
  // 23.2.5.1 types:
    typedef typename Allocator::types<T>::reference       reference;
    typedef typename Allocator::types<T>::const_reference const_reference;
    typedef typename Allocator::types<T>::pointer         iterator;
    typedef typename Allocator::types<T>::const_pointer   const_iterator;
    typedef typename Allocator::size_type       size_type;
    typedef typename Allocator::difference_type difference_type;
    typedef T value_type;
    typedef reverse_iterator<iterator, value_type,
                 reference, difference_type>              reverse_iterator;
    typedef reverse_iterator<const_iterator, value_type,
                 const_reference, difference_type>        const_reverse_iterator;

  // 23.2.5.2 construct/copy/destroy:
    explicit vector(Allocator& = Allocator());
    explicit vector(size_type n, const T& value = T(), Allocator& = Allocator());
    vector(const vector<T,Allocator>& x, Allocator& = Allocator());
    template <class InputIterator>
      vector(InputIterator first, InputIterator last, Allocator& = Allocator());
   ~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());

  // 23.2.5.3 iterators:
    iterator               begin();
    const_iterator         begin() const;
    iterator               end();
    const_iterator         end() const;
    reverse_iterator       rbegin();
    const_reverse_iterator rbegin() const;
    reverse_iterator       rend();
    const_reverse_iterator rend() const;

  // 23.2.5.4 capacity:
    size_type size() const;
    size_type max_size() const;
    void      resize(size_type sz, T c = T());
    size_type capacity() const;
    bool      empty() const;
    void      reserve(size_type n);
  // 23.2.5.5 element access:
    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;

  // 23.2.5.6 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);
    template <class InputIterator>
        void insert(iterator position, InputIterator first, InputIterator last);
    void erase(iterator position);
    void erase(iterator first, iterator last);
    void swap(vector<T,Allocator>&);
  };

  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);
}
```

23.2.5.1 `vector` types [lib.vector.types]

23.2.5.2 `vector` constructors, copy, and assignment [lib.vector.cons]

```
vector();
explicit 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.

```
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);
```

23.2.5.3 `vector` iterator support [lib.vector.iterators]

23.2.5.4 `vector` capacity [lib.vector.capacity]

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

Returns: The size of the allocated storage in the vector.

```
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`.

Notes: 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.
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`.

```
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
  ; // do nothing
```

23.2.5.5 `vector` element access [lib.vector.access]

23.2.5.6 `vector` modifiers [lib.vector.modifiers]

```
iterator insert(iterator position, const T& x = T());
void     insert(iterator position, size_type n, const T& x);
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.[192)]

[192)] 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,

```
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.

23.2.6 Class `vector<bool>` [lib.vector.bool]

1 To optimize space allocation, a specialization for `bool` is provided:[193)]

```
namespace std {
  class vector<bool,allocator> {
  public:
  // types:
    typedef const reference const_reference;
    typedef typename Allocator::types<bool>::pointer         iterator;
    typedef typename Allocator::types<bool>::const_pointer   const_iterator;
    typedef typename Allocator::size_type       size_type;
    typedef typename Allocator::difference_type difference_type;
    typedef bool value_type;
    typedef reverse_iterator<iterator, value_type,
                 reference, difference_type>              reverse_iterator;
    typedef reverse_iterator<const_iterator, value_type,
                 const_reference, difference_type>        const_reverse_iterator;

  // bit reference:
    class reference {
    public:
     ~reference();
      operator bool() const;
      reference& operator=(const bool x);
      void flip();      // flips the bit
    };

  // construct/copy/destroy:
    explicit vector(Allocator& = Allocator());
    explicit vector(size_type n, const bool& value = bool(),
                    Allocator& = Allocator());
    vector(const vector<bool,allocator>& x, Allocator& = Allocator());
    template <class InputIterator>
      vector(InputIterator first, InputIterator last, Allocator& = Allocator());
   ~vector();
    vector<bool,allocator>& operator=(const vector<bool,allocator>& x);
    template <class InputIterator>
      void assign(InputIterator first, InputIterator last);
    template <class Size, class T> void assign(Size n, const T& t = T());
```

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.

193) An implementation is expected to provide specializations of `vector<bool>` for all supported memory models.

```
  // iterators:
    iterator               begin();
    const_iterator         begin() const;
    iterator               end();
    const_iterator         end() const;
    reverse_iterator       rbegin();
    const_reverse_iterator rbegin() const;
    reverse_iterator       rend();
    const_reverse_iterator rend() const;

  // capacity:
    size_type size() const;
    size_type max_size() const;
    void      resize(size_type sz, bool c = false);
    size_type capacity() const;
    bool      empty() const;
    void      reserve(size_type n);

  // element access:
    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);
    void swap(vector<bool,Allocator>&);
    void swap(reference x, reference y);
    void flip();        // flips all bits
  };

  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
3 `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:
  // 23.3.1.1 types:
    typedef Key                key_type;
    typedef pair<const Key, T> value_type;
    typedef Compare            key_compare;

    typedef typename Allocator::types<value_type>::reference       reference;
    typedef typename Allocator::types<value_type>::const_reference
                                                             const_reference;
    typedef typename Allocator::types<value_type>::pointer         iterator;
    typedef typename Allocator::types<value_type>::const_pointer
                                                             const_iterator;
    typedef typename Allocator::size_type       size_type;
    typedef typename Allocator::difference_type difference_type;
    typedef reverse_iterator<iterator, value_type,
                 reference, difference_type>              reverse_iterator;
    typedef reverse_iterator<const_iterator, value_type,
                 const_reference, difference_type>        const_reverse_iterator;

    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);
      }
    };

  // 23.3.1.2 construct/copy/destroy:
    explicit map(const Compare& comp = Compare(), Allocator& = Allocator());
    template <class InputIterator>
      map(InputIterator first, InputIterator last,
          const Compare& comp = Compare(), Allocator& = Allocator());
    map(const map<Key,T,Compare,Allocator>& x, Allocator& = Allocator());
   ~map();
    map<Key,T,Compare,Allocator>&
      operator=(const map<Key,T,Compare,Allocator>& x);

  // 23.3.1.3 iterators:
    iterator               begin();
    const_iterator         begin() const;
    iterator               end();
    const_iterator         end() const;
    reverse_iterator       rbegin();
    const_reverse_iterator rbegin() const;
    reverse_iterator       rend();
    const_reverse_iterator rend() const;

  // 23.3.1.4 capacity:
    bool      empty() const;
    size_type size() const;
    size_type max_size() const;
  // 23.3.1.5 element access:
    T&       operator[](const key_type& x);
    const T& operator[](const key_type& x) const;

  // 23.3.1.6 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);
    void swap(map<Key,T,Compare,Allocator>&);

  // 23.3.1.7 observers:
    key_compare   key_comp() const;
    value_compare value_comp() const;

  // 23.3.1.8 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` types [lib.map.types]

23.3.1.2 `map` constructors, copy, and assignment [lib.map.cons]

23.3.1.3 `map` iterator support [lib.map.iterators]

23.3.1.4 `map` capacity [lib.map.capacity]

23.3.1.5 `map` element access [lib.map.access]

```
T& operator[](const key_type& x);
```

Returns: `(*((m.insert(make_pair(`*x*`, T()))).first)).second`.

23.3.1.6 `map` modifiers [lib.map.modifiers]

23.3.1.7 `map` observers [lib.map.observers]

23.3.1.8 `map` operations [lib.map.ops]

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);
      }
    };

    typedef typename Allocator::types<value_type>::reference       reference;
    typedef typename Allocator::types<value_type>::const_reference
                                                             const_reference;
    typedef typename Allocator::types<value_type>::pointer         iterator;
    typedef typename Allocator::types<value_type>::const_pointer
                                                             const_iterator;
    typedef typename Allocator::size_type       size_type;
    typedef typename Allocator::difference_type difference_type;
    typedef reverse_iterator<iterator, value_type,
                 reference, difference_type>              reverse_iterator;
    typedef reverse_iterator<const_iterator, value_type,
                 const_reference, difference_type>        const_reverse_iterator;

  // construct/copy/destroy:
    explicit multimap(const Compare& comp = Compare(),
                      Allocator& = Allocator());
    template <class InputIterator>
      multimap(InputIterator first, InputIterator last,
               const Compare& comp = Compare(), Allocator& = Allocator());
    multimap(const multimap<Key,T,Compare,Allocator>& x, Allocator& = Allocator());
   ~multimap();
    multimap<Key,T,Compare,Allocator>&
      operator=(const multimap<Key,T,Compare,Allocator>& x);
  // iterators:
    iterator                begin();
    const_iterator          begin() const;
    iterator                end();
    const_iterator          end() const;
    reverse_iterator        rbegin();
    const_reverse_iterator  rbegin() const;
    reverse_iterator        rend();
    const_reverse_iterator  rend() const;

  // capacity:
    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);
    void swap(multimap<Key,T,Compare,Allocator>&);

  // observers:
    key_compare    key_comp() const;
    value_compare  value_comp() const;

  // 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 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:
  // 23.3.3.1 types:
    typedef Key     key_type;
    typedef Key     value_type;
    typedef Compare key_compare;
    typedef Compare value_compare;
    typedef typename Allocator::types<Key>::reference       reference;
    typedef typename Allocator::types<Key>::const_reference const_reference;
    typedef typename Allocator::types<Key>::pointer         iterator;
    typedef typename Allocator::types<Key>::const_pointer   const_iterator;
    typedef typename Allocator::size_type       size_type;
    typedef typename Allocator::difference_type difference_type;
    typedef reverse_iterator<iterator, value_type,
                 reference, difference_type>              reverse_iterator;
    typedef reverse_iterator<const_iterator, value_type,
                 const_reference, difference_type>        const_reverse_iterator;

  // 23.3.3.2 construct/copy/destroy:
    explicit set(const Compare& comp = Compare(), Allocator& = Allocator());
    template <class InputIterator>
      set(InputIterator first, InputIterator last,
          const Compare& comp = Compare(), Allocator& = Allocator());
    set(const set<Key,Compare,Allocator>& x, Allocator& = Allocator());
   ~set();
    set<Key,Compare,Allocator>& operator=(const set<Key,Compare,Allocator>& x);

  // 23.3.3.3 iterators:
    iterator               begin();
    const_iterator         begin() const;
    iterator               end();
    const_iterator         end() const;
    reverse_iterator       rbegin();
    const_reverse_iterator rbegin() const;
    reverse_iterator       rend();
    const_reverse_iterator rend() const;

  // 23.3.3.4 capacity:
    bool          empty() const;
    size_type     size() const;
    size_type     max_size() const;

  // 23.3.3.5 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);
    void swap(set<Key,Compare,Allocator>&);

  // 23.3.3.6 observers:
    key_compare   key_comp() const;
    value_compare value_comp() const;
  // 23.3.3.7 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);
}
```

23.3.3.1 `set` types [lib.set.types]

23.3.3.2 `set` constructors, copy, and assignment [lib.set.cons]

23.3.3.3 `set` iterator support [lib.set.iterators]

23.3.3.4 `set` capacity [lib.set.capacity]

23.3.3.5 `set` modifiers [lib.set.modifiers]

23.3.3.6 `set` observers [lib.set.observers]

23.3.3.7 `set` operations [lib.set.ops]

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 typename Allocator::types<Key>::reference       reference;
    typedef typename Allocator::types<Key>::const_reference const_reference;
    typedef typename Allocator::types<Key>::pointer         iterator;
    typedef typename Allocator::types<Key>::const_pointer   const_iterator;
    typedef typename Allocator::size_type       size_type;
    typedef typename Allocator::difference_type difference_type;
    typedef reverse_iterator<iterator, value_type,
                 reference, difference_type>              reverse_iterator;
    typedef reverse_iterator<const_iterator, value_type,
                 const_reference, difference_type>        const_reverse_iterator;
  // construct/copy/destroy:
    explicit multiset(const Compare& comp = Compare(),
                      Allocator& = Allocator());
    template <class InputIterator>
      multiset(InputIterator first, InputIterator last,
               const Compare& comp = Compare(), Allocator& = Allocator());
    multiset(const multiset<Key,Compare,Allocator>& x, Allocator& = Allocator());
   ~multiset();
    multiset<Key,Compare,Allocator>&
        operator=(const multiset<Key,Compare,Allocator>& x);

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

  // capacity:
    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);
    void swap(multiset<Key,Compare,Allocator>&);

  // observers:
    key_compare   key_comp() const;
    value_compare value_comp() const;

  // 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 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 (27.6), and stream buffers (27.5).

2 The following subclauses describe iterator requirements, and components for iterator primitives, predefined iterators, and stream iterators, as summarized in Table 55:

Table 55—Iterators library summary

Subclause	Header(s)
24.1 Requirements	
24.2 Iterator primitives 24.3 Predefined iterators 24.4 Stream iterators	`<iterator>`

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, the library formalizes not just the interfaces but also the semantics and complexity assumptions of iterators. All iterators `i` support the expression `*i`, resulting in a value of some class, enumeration, or built-in type `T`, called the *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 an abstraction of pointers, their semantics is a generalization of most of the semantics of pointers in C++. This ensures that every template function that takes iterators works as well with regular pointers. This Standard defines five categories of iterators, according to the operations defined on them: *input iterators*, *output iterators*, *forward iterators*, *bidirectional iterators* and *random access iterators*, as shown in Table 56.

Table 56—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 also satisfy all the requirements of the forward iterators and can be used whenever a forward iterator is specified; Random access iterators also satisfy all the requirements of bidirectional iterators and can be used whenever a bidirectional iterator is specified.

4 Besides its category, a forward, bidirectional, or random access iterator can also be *mutable* or *constant* depending on whether the result of the expression `*i` behaves as a reference or as a reference to a constant. Constant iterators do not satisfy the requirements for output iterators, and the result of the expression `*i` (for constant iterator `i`) cannot be used in an expression where an lvalue is required.

5 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 an iterator `i` for which the expression `*i` is defined are called *dereferenceable.* The library never assumes that past-the-end values are dereferenceable. Iterators can 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` must always be assumed to have a singular value of a pointer. Results of most expressions are undefined for singular values; the only exception is an assignment of a non-singular value to an iterator that holds a singular value. In this case the singular value is overwritten the same way as any other value. Dereferenceable and past-the-end values are always non-singular.

6 An iterator `j` is called *reachable* from an iterator `i` if and only if there is a finite sequence of applications of the expression `++i` that makes `i == j`. If `j` is reachable from `i`, they refer to the same container.

7 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.

8 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.

9 In the following sections, `a` and `b` denote values of `X`, `n` denotes a value of the distance type `Distance`, `u`, `tmp`, and `m` denote identifiers, `r` denotes a value of `X&`, `t` denotes 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 following expressions are valid, as shown in Table 57:

Table 57—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`	`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++`	convertible to `const X&`	`{ X tmp = r;`
		`  ++r;`
		`  return tmp; }`	

`*r++`	`T`		

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. —*end note*]

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 58:

Table 58—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		
`++r`	`X&`		`&r == &++r`.

`r++`	convertible to `const X&`	`{ X tmp = r;`
		`  ++r;`
		`  return tmp; }`	

`*r++ = t`	result is not used		

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. —*end note*]

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 59:

Table 59—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`	`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 dereference-able implies `++r == ++s`. `&r == &++r`.

`r++`	convertible to `const X&`	`{ X tmp = r;`
		`  ++r;`
		`  return tmp; }`	

`*r++`	`T&`		

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. —*end note*]

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, in addition to satisfying the requirements for forward iterators, the following expressions are valid as shown in Table 60:

Table 60—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--`	convertible to `const X&`	`{ X tmp = r;`
		`  --r;`
		`  return tmp; }`	

`*r--`	convertible to `T`		

2 [*Note:* Bidirectional iterators allow algorithms to move iterators backward as well as forward. —*end note*]

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, in addition to satisfying the requirements for bidirectional iterators, the following expressions are valid as shown in Table 61:

Table 61—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`	`{` ***TBS*** `}`	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)`	

24.1.6 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.

2 [*Note:* For all the regular pointer types, `value_type()` and `distance_type()` can be defined 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); }
```

—*end note*]

3 [*Example:* To implement a generic `reverse` function, a C++ program can do the following:

```
template <class BidirectionalIterator>
inline void reverse(BidirectionalIterator first, BidirectionalIterator last) {
  __reverse(first, last, value_type(first), distance_type(first));
}
```

4 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;
  }
}
```

—*end example*]

5 [*Note:* If there is an additional pointer type `far` such that the difference of two `far` pointers is of the type `long`, an implementation may 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); }
```

—*end note*]

6 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.

7 [*Example:* If the pointer types are defined to be in the random access iterator category by:

```
template <class T>
inline random_access_iterator_tag
  iterator_category(const T*)
    { return random_access_iterator_tag(); }
```

8 For a program-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();
}
```

—*end example*]

9 [*Example:* 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
  }
```

—*end example*]

10 [*Example:* If a C++ program wants to define a bidirectional iterator for some data structure containing `double` and such that it works on a large memory model of the implementation, it can do so with:

```
  class MyIterator : public bidirectional_iterator<double, long> {
  // code implementing ++, etc.
  };
```

11 Then there is no need to define `iterator_category`, `value_type`, and `distance_type` on `MyIterator`. —*end example*]

Header `<iterator>` synopsis

```
#include <cstddef>      // for ptrdiff_t
#include <iosfwd>       // for istream, ostream
#include <ios>          // for ios_traits
#include <streambuf>    // for streambuf

namespace std {
// subclause _lib.library.primitives_, 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*);

// subclause 24.2.6, 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 Reference, class Distance = ptrdiff_t>
      class reverse_bidirectional_iterator;
  template <class BidirectionalIterator, class T, class Reference, class Distance>
    bool operator==(
      const reverse_bidirectional_iterator
        <BidirectionalIterator,T,Reference,Distance>& x,
      const reverse_bidirectional_iterator
        <BidirectionalIterator,T,Reference,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,Reference,Distance>& x,
      const reverse_iterator<RandomAccessIterator,T,Reference,Distance>& y);
  template <class RandomAccessIterator, class T, class Distance>
    bool operator<(
      const reverse_iterator<RandomAccessIterator,T,Reference,Distance>& x,
      const reverse_iterator<RandomAccessIterator,T,Reference,Distance>& y);
  template <class RandomAccessIterator, class T, class Distance>
    Distance operator-(
      const reverse_iterator<RandomAccessIterator,T,Reference,Distance>& x,
      const reverse_iterator<RandomAccessIterator,T,Reference,Distance>& y);
  template <class RandomAccessIterator, class T, class Distance>
    reverse_iterator<RandomAccessIterator,T,Reference,Distance> operator+
      (Distance n,
       const reverse_iterator<RandomAccessIterator,T,Reference,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);
}
```

24.2 Iterator primitives [lib.iterator.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:

24.2.1 Standard iterator tags [lib.std.iterator.tags]

```
namespace std {
  struct input_iterator_tag {};
  struct output_iterator_tag {};
  struct forward_iterator_tag {};
  struct bidirectional_iterator_tag {};
  struct random_access_iterator_tag {};
}
```

24.2.2 Basic iterators [lib.basic.iterators]

```
namespace std {
  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 {};
}
```

1 [*Note:* `output_iterator` is not a template because output iterators do not have either value type or distance type defined. —*end note*]

24.2.3 `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()`.

24.2.4 `value_type` [lib.value.type]

```
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*);
```

Returns: `(T*)(0)`.

24.2.5 `distance_type` [lib.distance.type]

```
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>&);
```

Returns: `(Distance*)(0)`.

```
template <class T> ptrdiff_t* distance_type(const T*);
```

Returns: `(ptrdiff_t*)(0)`.

24.2.6 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 `last`.[194)]

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: `&*(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 Reference = T&, class Distance = ptrdiff_t>
  class reverse_bidirectional_iterator
    : public bidirectional_iterator<T,Distance> {
  protected:
    BidirectionalIterator current;
  public:
    reverse_bidirectional_iterator();
    explicit reverse_bidirectional_iterator(BidirectionalIterator x);
    BidirectionalIterator base();       // explicit
    Reference operator*();
    reverse_bidirectional_iterator<BidirectionalIterator,T,Reference,Distance>&
      operator++();
    reverse_bidirectional_iterator<BidirectionalIterator,T,Reference,Distance>
      operator++(int);
    reverse_bidirectional_iterator<BidirectionalIterator,T,Reference,Distance>&
      operator--();
    reverse_bidirectional_iterator<BidirectionalIterator,T,Reference,Distance>
      operator--(int);
  };

  template <class BidirectionalIterator, class T, class Distance>
    bool operator==(
      const reverse_bidirectional_iterator
        <BidirectionalIterator,T,Reference,Distance>& x,
      const reverse_bidirectional_iterator
        <BidirectionalIterator,T,Reference,Distance>& y);
}
```

194) `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*`.

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]

```
explicit reverse_bidirectional_iterator(BidirectionalIterator x);
```

Effects: Initializes `current` with *x*.

24.3.1.2.2 Conversion [lib.reverse.bidir.iter.conv]

```
BidirectionalIterator base();   // explicit
```

Returns: `current`

24.3.1.2.3 `operator*` [lib.reverse.bidir.iter.op.star]

```
Reference operator*();
```

Effects:

```
BidirectionalIterator tmp = current;
return *--tmp;
```

24.3.1.2.4 `operator++` [lib.reverse.bidir.iter.op++]

```
reverse_bidirectional_iterator<BidirectionalIterator,T,Reference,Distance>&
  operator++();
```

Effects: `--current;`
Returns: `*this`

```
reverse_bidirectional_iterator<BidirectionalIterator,T,Reference,Distance>
  operator++(int);
```

Effects:

```
reverse_bidirectional_iterator<BidirectionalIterator,T,Reference,Distance>
  tmp = *this;
--current;
return tmp;
```

24.3.1.2.5 `operator--` [lib.reverse.bidir.iter.op--]

```
reverse_bidirectional_iterator
  <BidirectionalIterator,T,Reference,Distance>&
    operator--();
```

Effects: `++current`
Returns:

```
reverse_bidirectional_iterator<BidirectionalIterator,T,Reference,Distance>
  operator--(int);
```

Effects:

```
reverse_bidirectional_iterator
  <BidirectionalIterator,T,Reference,Distance> tmp = *this;
++current;
return tmp;
```

24.3.1.2.6 `operator==` [lib.reverse.bidir.iter.op==]

```
template <class BidirectionalIterator, class T, class Reference, class Distance>
  bool operator==(
    const reverse_bidirectional_iterator
      <BidirectionalIterator,T,Reference,Distance>& x,
    const reverse_bidirectional_iterator
      <BidirectionalIterator,T,Reference,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 Reference = T&, class Distance = ptrdiff_t>
  class reverse_iterator : public random_access_iterator<T,Distance> {
  protected:
    RandomAccessIterator current;
  public:
    reverse_iterator();
    explicit reverse_iterator(RandomAccessIterator x);

    RandomAccessIterator base();        // explicit
    Reference operator*();

    reverse_iterator<RandomAccessIterator,T,Reference,Distance>& operator++();
    reverse_iterator<RandomAccessIterator,T,Reference,Distance>  operator++(int);
    reverse_iterator<RandomAccessIterator,T,Reference,Distance>& operator--();
    reverse_iterator<RandomAccessIterator,T,Reference,Distance>  operator--(int);

    reverse_iterator<RandomAccessIterator,T,Reference,Distance>
      operator+ (Distance n) const;
    reverse_iterator<RandomAccessIterator,T,Reference,Distance>&
      operator+=(Distance n);
    reverse_iterator<RandomAccessIterator,T,Reference,Distance>
      operator- (Distance n) const;
    reverse_iterator<RandomAccessIterator,T,Reference,Distance>&
      operator-=(Distance n);
    Reference operator[](Distance n);

    template <class RandomAccessIterator, class T,
              class Reference, class Distance>
      bool operator==(
        const reverse_iterator<RandomAccessIterator,T,Reference,Distance>& x,
        const reverse_iterator<RandomAccessIterator,T,Reference,Distance>& y);

    template <class RandomAccessIterator, class T,
              class Reference, class Distance>
      bool operator<(
        const reverse_iterator<RandomAccessIterator,T,Reference,Distance>& x,
        const reverse_iterator<RandomAccessIterator,T,Reference,Distance>& y);
    template <class RandomAccessIterator, class T,
              class Reference, class Distance>
      Distance operator-(
        const reverse_iterator<RandomAccessIterator,T,Reference,Distance>& x,
        const reverse_iterator<RandomAccessIterator,T,Reference,Distance>& y);

    template <class RandomAccessIterator, class T,
              class Reference, class Distance>
      reverse_iterator<RandomAccessIterator,T,Reference,Distance> operator+(
             Distance n,
             const reverse_iterator
               <RandomAccessIterator,T,Reference,Distance>& x);
  };
}
```

1 [*Note:* 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, it would have been written as:

```
  template <class BidirectionalIterator,
    class T = typename BidirectionalIterator::reference_type,
    class Distance = typename BidirectionalIterator::difference_type>
  class reverse_bidirectional_iterator: bidirectional_iterator<T,Distance> {
  /* ... */
  };
```

—*end note*]

24.3.1.4 `reverse_iterator` operations [lib.reverse.iter.ops]

24.3.1.4.1 `reverse_iterator` constructor [lib.reverse.iter.cons]

```
explicit reverse_iterator(RandomAccessIterator x);
```

Effects: Initializes `current` with *x*.

24.3.1.4.2 Conversion [lib.reverse.iter.conv]

```
RandomAccessIterator base();    // explicit
```

Returns: `current`

24.3.1.4.3 `operator*` [lib.reverse.iter.op.star]

```
Reference operator*();
```

Effects:

```
RandomAccessIterator tmp = current;
return *--tmp;
```

24.3.1.4.4 `operator++` [lib.reverse.iter.op++]

```
reverse_iterator<RandomAccessIterator,T,Reference,Distance>&
  operator++();
```

Effects: `--current;`
Returns: `*this`

```
reverse_iterator<RandomAccessIterator,T,Reference,Distance>
  operator++(int);
```

Effects:

```
reverse_iterator<RandomAccessIterator,T,Reference,Distance> tmp = *this;
--current;
return tmp;
```

24.3.1.4.5 `operator--` [lib.reverse.iter.op--]

```
reverse_iterator<RandomAccessIterator,T,Reference,Distance>&
  operator--();
```

Effects: `++current`
Returns:

```
reverse_iterator<RandomAccessIterator,T,Reference,Distance>
  operator--(int);
```

Effects:

```
reverse_iterator<RandomAccessIterator,T,Reference,Distance> tmp = *this;
++current;
return tmp;
```

24.3.1.4.6 `operator==` [lib.reverse.iter.op==]

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

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:
    explicit back_insert_iterator(Container& x);
    back_insert_iterator<Container>&
      operator=(const typename 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]

```
explicit back_insert_iterator(Container& x);
```

Effects: Initializes `container` with *x*.

24.3.2.2.2 `back_insert_iterator::operator=` [lib.back.insert.iter.op=]

```
back_insert_iterator<Container>&
  operator=(const typename 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:
    explicit front_insert_iterator(Container& x);
    front_insert_iterator<Container>&
      operator=(const typename 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]

```
explicit front_insert_iterator(Container& x);
```

Effects: Initializes `container` with *x*.

24.3.2.4.2 `front_insert_iterator::operator=` [lib.front.insert.iter.op=]

```
front_insert_iterator<Container>&
  operator=(const typename 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;
    typename Container::iterator iter;

  public:
    insert_iterator(Container& x, typename Container::iterator i);
    insert_iterator<Container>&
      operator=(const typename 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, Iterator i);
```

Effects: Initializes `container` with *x* and `iter` with *i*.

24.3.2.6.2 `insert_iterator::operator=` [lib.insert.iter.op=]

```
insert_iterator<Container>&
  operator=(const typename Container::value_type& value);
```

Effects:

```
  iter = container.insert(iter, value);
  ++iter;
```

insert_iterator::operator=

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,typename 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.

2 [*Example:*

```
partial_sum_copy(istream_iterator<double>(cin),  istream_iterator<double>(),
  ostream_iterator<double>(cout, "\n"));
```

reads a file containing floating point numbers from `cin`, and prints the partial sums onto `cout`. —*end example*]

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 : public 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(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 typename 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. `operator*` provides access to the current input character, if any. Each time `operator++` is evaluated, the iterator advances to the next input character. If the end of stream is reached (streambuf::sgetc() returns `traits::eof()`), the iterator becomes equal to the *end of stream* iterator value. The default constructor `istreambuf_iterator()` and the constructor `istreambuf_iterator(0)` both construct an end of stream iterator object suitable for use as an end-of-range.

2 The result of `operator*()` on an end of stream is undefined. For any other iterator value a `char_type` is returned. It is impossible to assign a character via an input iterator.

3 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 evaluated a new value is used.

4 The practical consequence of this fact is that an `istreambuf_iterator` object can be used only for *one-pass algorithms*. Two end of stream iterators are always equal. An end of stream iterator is not equal to a non-end of stream iterator.

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<charT, traits>::proxy {
    charT keep_;
    basic_streambuf<charT,traits>* sbuf_;
    proxy(charT c,
          basic_streambuf<charT,traits>* sbuf);
      : keep_(c), sbuf_(sbuf) {}
  public:
    charT operator*() { return keep_; }
  };
}
```

1 Class `istream_iterator<charT,traits>::proxy` provides a temporary placeholder as the return value of the post-increment operator (`operator++`). It keeps the character pointed to by the previous value of the iterator for some possible future access to get the character.

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*()
```

Effects: 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 and only if both iterators are either at end-of-stream, or are the end-of-stream value, regardless of what `streambuf` they iterator over.

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 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) {}`

```
ostreambuf_iterator<charT,traits>&
  operator=(charT c);
```

Effects:

sbuf_`->sputc(c);`

Returns: `*this`.

24.4.4.2 `ostreambuf_iterator` operations [lib.ostreambuf.iter.ops]

```
ostreambuf_iterator<charT,traits>& operator*();
```

Returns: `*this`.

```
ostreambuf_iterator<charT,traits>& operator++();
ostreambuf_iterator<charT,traits>  operator++(int);
```

Returns: `*this`.

```
bool equal(ostreambuf_iterator& b);
```

Returns: ***sbuf_*** `== b.sbuf`.

24.4.4.3 `ostreambuf_iterator` non-member operations [lib.ostreambuf.iterator.nonmembers]

```
output_iterator iterator_category (const ostreambuf_iterator&);
```

Returns: `output_iterator()`.

```
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)`.

```
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-modifying sequence operation, modifying sequence operations, sorting and related operations, and algorithms from the ISO C library, as summarized in Table 62:

Table 62—Algorithms library summary

Subclause	Header(s)
25.1 Non-modifying 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-modifying sequence operations:
  template<class InputIterator, class Function>
    Function 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,
               BinaryPredicate pred);
```

```
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);

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, modifying 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 ForwardIterator1, class ForwardIterator2>
    void iter_swap(ForwardIterator1 a, ForwardIterator2 b);

  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 ForwardIterator>
    void rotate(ForwardIterator first, ForwardIterator middle,
                ForwardIterator 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>
    BidirectionalIterator 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> const T& min(const T& a, const T& b);
  template<class T, class Compare>
    const T& min(const T& a, const T& b, Compare comp);
  template<class T> const T& max(const T& a, const T& b);
  template<class T, class Compare>
    const T& max(const T& a, const T& b, Compare comp);

  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 InputIterator>
    InputIterator max_element(InputIterator first, InputIterator last);
  template<class InputIterator, class Compare>
    InputIterator max_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.[195] 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 correctly in the construct `if (`*`pred`*`(*`*`first`*`)){...}`. The function object `pred` is assumed not to apply any non-constant function through the dereferenced iterator.

[195] 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`.

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)){...}`. *binary_pred* shall 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;
}
```

25.1 Non-modifying sequence operations [lib.alg.nonmodifying]

25.1.1 For each [lib.alg.foreach]

```
template<class InputIterator, class Function>
  Function 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.
Returns: *f*.
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: Exactly `find(`*`first`*`, `*`last`*`, `*`value`*`) - `*`first`* 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 *last*`2 - first2` the following corresponding conditions hold: `*(i + n) == *(`*first2* `+ n)`, *pred*`(*(i + n), *(`*first2* `+ n)) == true`. Returns *last1* if no such iterator is found.[196)]

Complexity: At most `(`*last1* `-` *first1*`) * (`*last2* `-` *first2*`)` applications of the corresponding predicate.

```
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.

Complexity: At most `(`*last1* `-` *first1*`) *` *count* applications of the corresponding predicate.

25.2 Mutating sequence operations [lib.alg.modifying.operations]

196) 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. Most implementations will provide a specialization:

```
char* search(char* first1, char* last1, char* first2, char* last2);
```

that will use a variation of the Boyer-Moore algorithm for fast string searching.

25.2.1 Copy [lib.alg.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.

```
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 *`first`*. [197] 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]

```
template<class T> void swap(T& a, T& b);
```

Effects: Exchanges values stored in two locations.

```
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: *`first`*`2 + (`*`last1`*` - `*`first1`*`)`.
Complexity: Exactly *`last1 - first1`* swaps.

```
template<class ForwardIterator1, class ForwardIterator2>
  void iter_swap(ForwardIterator1 a, ForwardIterator2 b);
```

Effects: Exchanges the values pointed to by the two iterators *a* and *b*.

[197] `copy_backward` (_lib.copy.backward_) 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]

```
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.

```
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]

```
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.

```
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]

```
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.

```
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]

```
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.

```
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]

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

Effects: For each non-negative integer `i < (last - first)`, places the element from the position `first + i` into position `first + (i + (last - middle)) % (last - first)`.
Notes: This is a left rotate.
Requires: `[first, middle)` and `[middle, last)` are valid ranges.
Complexity: At most `last - first` swaps.

```
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*`(`*n*`)` (where *n* is a positive argument of type `RandomAccessIterator::distance`) returns a randomly chosen value of type `RandomAccessIterator::distance`) in the interval `[0,` *n*`)`.

25.2.12 Partitions [lib.alg.partitions]

```
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.

```
template<class BidirectionalIterator, class Predicate>
  BidirectionalIterator
    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.

3 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.

4 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`.

5 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.[198)]

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 `N(logN)`^{2} (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]

198) If the worst case behavior is important `stable_sort()` (25.3.1.2) 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 the `j` in 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) + 1` 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) + 2` comparisons.

25.3.4 Merge [lib.alg.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.

```
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: When enough additional memory is available, `(`*`last`*` - `*`first`*`) - 1` comparisons. If no additional memory is available, an algorithm with complexity `NlogN` (where `N` is equal to *`last`*` - `*`first`*) may be used.

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 (23.3.4) containing multiple copies of equal elements. The semantics of the set operations are generalized to `multiset`s 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 `2 * ((`*`last1`*` - `*`first1`*`) + (`*`last2`*` - `*`first2`*`)) - 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 `2 * ((`*`last1`*` - `*`first1`*`) + (`*`last2`*` - `*`first2`*`)) - 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 `2 * ((`*`last1`*` - `*`first1`*`) + (`*`last2`*` - `*`first2`*`)) - 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 `2 * ((`*`last1`*` - `*`first1`*`) + (`*`last2`*` - `*`first2`*`)) - 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 `2 * ((`*`last1`*` - `*`first1`*`) + (`*`last2`*` - `*`first2`*`)) - 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.

2 These properties make heaps useful as priority queues.

3 `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]

```
template<class T> const T& min(const T& a, const T& b);
template<class T, class Compare>
  const T& min(const T& a, const T& b, Compare comp);
```

Returns: The smaller value.
Notes: Returns the first argument when their arguments are equal.

```
template<class T> const T& max(const T& a, const T& b);
template<class T, class Compare>
  const T& max(const T& a, const T& b, Compare comp);
```

Returns: The larger value.
Notes: Returns the first argument when their arguments are equal.

```
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.

```
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.

25.3.8 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]

```
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.

```
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 63):

Table 63—Header `<cstdlib>` synopsis

Type	Name(s)	
Functions:	`bsearch`	`qsort`

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

[*Note:* For the Standard C library function:

```
void qsort(void* base, size_t nmemb, size_t size,
           int (*compar)(const void*, const void*));
```

the function argument *`compar`* shall have `extern "C"` linkage (7.5). Also, since *`compar()`* may throw an exception, `qsort()` is allowed to propagate the exception (17.3.4.8). —*end note*]

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 64:

Table 64—Numerics library summary

Subclause	Header(s)
26.1 Requirements	
26.2 Complex numbers	`<complex>`
26.3 Numeric arrays	`<valarray>`
26.4 Generalized numeric operations	`<numeric>`
26.5 C library	`<cmath>` `<cstdlib>`

26.1 Numeric type requirements [lib.numeric.requirements]

1 The `complex` and `valarray` components are parameterized by the type of information they contain and manipulate. A C++ program shall instantiate these components with types that satisfy the following requirements:[199)]

— *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.
[*Note:* 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.
[*Example:* An implementation is allowed to initialize a `valarray` by allocating storage using the `new`

199) In other words, value types. These include built-in arithmetic types, pointers, the library class `complex`, and instantiations of `valarray` for value types.

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. —*end example*]
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 `vector` (23.2.5) instead of `valarray` for that class; —*end note*]

— If *T* is a class, it does not overload unary `operator&`.

2 In addition, many member and related functions of `valarray<`*T*`>` can be successfully instantiated and will exhibit well-defined behavior if and only if *T* satisfies additional requirements specified for each such member or related function.

3 [*Example:* It is valid to instantiate `valarray<`*complex*`>`, but `operator>()` will not be successfully instantiated for `valarray<`*complex*`>` operands, since `complex` does not have any ordering operators. —*end example*]

26.2 Complex numbers [lib.complex.numbers]

1 The header `<complex>` defines a template class, and numerous functions for representing and manipulating complex numbers.

Header `<complex>` synopsis

```
namespace std {
  template<class T> class complex;
  class complex<float>;
  class complex<double>;
  class complex<long double>;

  // 26.2.5 operators:
  template<class T>
    complex<T> operator+(const complex<T>&, const complex<T>&);
  template<class T> complex<T> operator+(const complex<T>&, T);
  template<class T> complex<T> operator+(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>&, T);
  template<class T> complex<T> operator-(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>&, T);
  template<class T> complex<T> operator*(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/(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>&, T);
  template<class T> complex<T> operator==(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>&, T);
  template<class T> complex<T> operator!=(T, const complex<T>&);
  template<class T> istream& operator>>(istream&, complex<T>&);
  template<class T> ostream& operator<<(ostream&, const complex<T>&);

  // 26.2.6 values:

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

  template<class T> T abs(const complex<T>&);
  template<class T> T arg(const complex<T>&);
  template<class T> T norm(const complex<T>&);

  template<class T> complex<T> conj(const complex<T>&);
  template<class T> complex<T> polar(T, T);

  // 26.2.7 transcendentals:
  template<class T> complex<T> acos (const complex<T>&);
  template<class T> complex<T> asin (const complex<T>&);
  template<class T> complex<T> atan (const complex<T>&);
  template<class T> complex<T> atan2(const complex<T>&, const complex<T>&);
  template<class T> complex<T> atan2(const complex<T>&, T);
  template<class T> complex<T> atan2(T, 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> log10(const complex<T>&);

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

  template<class T> complex<T> sin  (const complex<T>&);
  template<class T> complex<T> sinh (const complex<T>&);
  template<class T> complex<T> sqrt (const complex<T>&);
  template<class T> complex<T> tan  (const complex<T>&);
  template<class T> complex<T> tanh (const complex<T>&);
}
```

26.2.1 Template class `complex` [lib.complex]

```
namespace std {
  template<class T>
  class complex {
  public:
    complex();
    complex(T re);
    complex(T re, T im);
    template<class X> complex(const complex<X>&);

    T real() const;
    T imag() const;
    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>&);
  };
```

1 The class `complex` describes an object that can store the Cartesian components, `real()` and `imag()`, of a complex number.

26.2.2 `complex` specializations [lib.complex.special]

```
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<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<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<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>&);
};
```

26.2.3 `complex` member functions [lib.complex.members]

`template<class T> complex(T` *`re`* `= T(), T` *`im`* `= T());`

Effects: Constructs an object of class `complex`.

1 Postcondition: `real() ==` *`re`* `&& imag() ==` *`im`*.

26.2.4 `complex` member operators [lib.complex.member.ops]

```
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`.

```
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`.

```
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`.

```
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`.

26.2.5 `complex` non-member operations [lib.complex.ops]

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

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

Returns: `complex<T>(lhs) += rhs`.

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

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

Returns: `complex<T>(lhs) -= rhs`.

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

Returns: `complex<T>(`*lhs*`) *=` *rhs*.

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

Returns: `complex<T>(`*lhs*`) /=` *rhs*.

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

Returns: *lhsP.real() == rhs.real() && lhs.imag() == rhs.imag()*.
Notes: The imaginary part is assumed to be `T()`, or 0.0, for the `T` arguments.

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

Returns: *rhs*)`!(`*lhs*==

```
template<class T> istream& operator>>(istream& is, complex<T>& x);
```

Effects: Extracts a complex number *x* of the form: `u`, `(u)`, or `(u,v)`, where `u` is the real part and `v` is the imaginary part (27.6.1.2).
Requires: The input values be convertible to `T`.
If bad input is encountered, calls *is*`.setstate(ios::failbit)` (which may throw `ios::failure` (27.4.4.3).
Returns: *is*.

```
template<class T>
  ostream& operator<<(ostream& os, complex x);
```

Returns: *os* `<< '(' <<` *x*`.real() << ',' <<` *x*`.imag() << ')'`.

26.2.6 `complex` value operations [lib.complex.value.ops]

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

Returns: *x*`.real()`.

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

Returns: *x*`.imag()`.

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

Returns: the ***TBS*** of *x*.

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

Returns: the squared magnitude of *x*.

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

Returns: the ***TBS*** of *x*.

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

Returns: the `complex` value corresponding to a complex number whose magnitude is *rho* and whose phase angle is *theta*.

26.2.7 `complex` transcendentals [lib.complex.transcendentals]

```
template<class T> complex<T> acos (const complex<T>& x);
template<class T> complex<T> asin (const complex<T>& x);
template<class T> complex<T> atan (const complex<T>& x);
template<class T> complex<T> atan2(const complex<T>& x);
template<class T> complex<T> atan2(const complex<T>& x, T y);
template<class T> complex<T> atan2(T x, const complex<T>& y);
template<class T> complex<T> cos  (const complex<T>& x);
template<class T> complex<T> cosh (const complex<T>& x);
template<class T> complex<T> exp  (const complex<T>& x);
template<class T> complex<T> log  (const complex<T>& x);
template<class T> complex<T> log10(const complex<T>& x);
template<class T>
  complex<T> pow(const complex<T>& x, const complex<T>& y);
template<class T> complex<T> pow  (const complex<T>& x, T y);
template<class T> complex<T> pow  (T x, const complex<T>& y);
template<class T> complex<T> pow  (const complex<T>& x, int y);
template<class T> complex<T> sin  (const complex<T>& x);
template<class T> complex<T> sinh (const complex<T>& x);
template<class T> complex<T> sqrt (const complex<T>& x);
template<class T> complex<T> tan  (const complex<T>& x);
template<class T> complex<T> tanh (const complex<T>& x);
```

1 For each of these functions ***F***, returns a `complex` value corresponding to the mathematical function (26.5) computed for `complex` arguments.

26.3 Numeric arrays [lib.numarray]

Header `<valarray>` synopsis

```
#include <cstddef>    // for size_t
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> valarray<T> operator*
  (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator* (const valarray<T>&, const T&);
template<class T> valarray<T> operator* (const T&, const valarray<T>&);

template<class T> valarray<T> operator/
  (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator/ (const valarray<T>&, const T&);
template<class T> valarray<T> operator/ (const T&, const valarray<T>&);

template<class T> valarray<T> operator%
  (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator% (const valarray<T>&, const T&);
template<class T> valarray<T> operator% (const T&, const valarray<T>&);

template<class T> valarray<T> operator+
  (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator+ (const valarray<T>&, const T&);
template<class T> valarray<T> operator+ (const T&, const valarray<T>&);

template<class T> valarray<T> operator-
  (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator- (const valarray<T>&, const T&);
template<class T> valarray<T> operator- (const T&, const valarray<T>&);

template<class T> valarray<T> operator^
  (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator^ (const valarray<T>&, const T&);
template<class T> valarray<T> operator^ (const T&, const valarray<T>&);

template<class T> valarray<T> operator&
  (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator& (const valarray<T>&, const T&);
template<class T> valarray<T> operator& (const T&, const valarray<T>&);

template<class T> valarray<T> operator|
  (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator| (const valarray<T>&, const T&);
template<class T> valarray<T> operator| (const T&, const valarray<T>&);

template<class T> valarray<T> operator<<
  (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator<<(const valarray<T>&, const T&);
template<class T> valarray<T> operator<<(const T&, const valarray<T>&);

template<class T> valarray<T> operator>>
  (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator>>(const valarray<T>&, const T&);
template<class T> valarray<T> operator>>(const T&, const valarray<T>&);

template<class T> valarray<T> operator&&
  (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator&&(const valarray<T>&, const T&);
template<class T> valarray<T> operator&&(const T&, const valarray<T>&);

template<class T> valarray<T> operator||
  (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator||(const valarray<T>&, const T&);
template<class T> valarray<T> operator||(const T&, const valarray<T>&);
```

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

  template<class T>
    valarray<bool> operator< (const valarray<T>&, const valarray<T>&);
  template<class T> valarray<bool> operator< (const valarray<T>&, const T&);
  template<class T> valarray<bool> operator< (const T&, const valarray<T>&);
  template<class T>
    valarray<bool> operator> (const valarray<T>&, const valarray<T>&);
  template<class T> valarray<bool> operator> (const valarray<T>&, const T&);
  template<class T> valarray<bool> operator> (const T&, const valarray<T>&);
  template<class T>
    valarray<bool> operator<=(const valarray<T>&, const valarray<T>&);
  template<class T> valarray<bool> operator<=(const valarray<T>&, const T&);
  template<class T> valarray<bool> operator<=(const T&, const valarray<T>&);
  template<class T>
    valarray<bool> operator>=(const valarray<T>&, const valarray<T>&);
  template<class T> valarray<bool> operator>=(const valarray<T>&, const T&);
  template<class T> valarray<bool> operator>=(const T&, const valarray<T>&);

  template<class T> T min(const valarray<T>&);
  template<class T> T max(const valarray<T>&);

  template<class T> valarray<T> abs  (const valarray<T>&);
  template<class T> valarray<T> acos (const valarray<T>&);
  template<class T> valarray<T> asin (const valarray<T>&);
  template<class T> valarray<T> atan (const valarray<T>&);

  template<class T> valarray<T> atan2(const valarray<T>&, const valarray<T>&);
  template<class T> valarray<T> atan2(const valarray<T>&, const T&);
  template<class T> valarray<T> atan2(const T&, const valarray<T>&);

  template<class T> valarray<T> cos  (const valarray<T>&);
  template<class T> valarray<T> cosh (const valarray<T>&);
  template<class T> valarray<T> exp  (const valarray<T>&);
  template<class T> valarray<T> log  (const valarray<T>&);
  template<class T> valarray<T> log10(const valarray<T>&);

  template<class T> valarray<T> pow  (const valarray<T>&, const valarray<T>&);
  template<class T> valarray<T> pow  (const valarray<T>&, const T&);
  template<class T> valarray<T> pow  (const T&, const valarray<T>&);

  template<class T> valarray<T> sin  (const valarray<T>&);
  template<class T> valarray<T> sinh (const valarray<T>&);
  template<class T> valarray<T> sqrt (const valarray<T>&);
  template<class T> valarray<T> tan  (const valarray<T>&);
  template<class T> valarray<T> tanh (const valarray<T>&);
}
```

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.

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` (18.4.2.1) exception if there are not sufficient resources available to carry out the operation. Note that the exception is not mandated.

26.3.1 Template class `valarray` [lib.template.valarray]

```
namespace std {
  template<class T> class valarray {
  public:
    // 26.3.1.1 construct/destroy:
    valarray();
    explicit valarray(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();

  // 26.3.1.2 assignment:
    valarray<T>& operator=(const valarray<T>&);
    valarray<T>& operator=(const slice_array<T>&);
    valarray<T>& operator=(const gslice_array<T>&);
    valarray<T>& operator=(const mask_array<T>&);
    valarray<T>& operator=(const indirect_array<T>&);

  // 26.3.1.3 element access:
    T                 operator[](size_t) const;
    T&                operator[](size_t);

  // _lib.valarray.subset_ subset operations:
    valarray<T>       operator[](slice) const;
    slice_array<T>    operator[](slice);
    valarray<T>       operator[](const gslice&) const;
    gslice_array<T>   operator[](const gslice&);
    valarray<T>       operator[](const valarray<bool>&) const;
    mask_array<T>     operator[](const valarray<bool>&);
    valarray<T>       operator[](const valarray<size_t>&) const;
    indirect_array<T> operator[](const valarray<size_t>&);

  // 26.3.1.5 unary operators:
    valarray<T> operator+() const;
    valarray<T> operator-() const;
    valarray<T> operator~() const;
    valarray<T> operator!() const;

  // 26.3.1.6 computed assignment:
    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>&);
    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>&);
    valarray<T>& operator|= (const valarray<T>&);
    valarray<T>& operator&= (const valarray<T>&);
    valarray<T>& operator<<=(const valarray<T>&);
    valarray<T>& operator>>=(const valarray<T>&);

  // 26.3.1.7 member functions:
    size_t length() const;
    operator T*();
    operator const T*() const;

    T    sum() const;
    void fill(const T&);
    T    min() const;
    T    max() const;

    valarray<T> shift (int) const;
    valarray<T> cshift(int) const;
    valarray<T> apply(T func(T)) 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.[200)]

2 An implementation is permitted to qualify any of the functions declared in `<valarray>` as `inline`.

26.3.1.1 `valarray` constructors [lib.valarray.cons]

```
valarray();
```

Effects: Constructs an object of class `valarray<`*T*`>`,[201)] 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.

```
explicit valarray(size_t);
```

1 The array created by this constructor has a length equal to the value of the argument. The elements of the array are constructed using the default constructor for the instantiating type *T*.

200) 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.
201) For convenience, such objects are referred to as ‘‘arrays’’ throughout the remainder of subclause 26.3.

```
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<T>&);
```

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`.

```
~valarray();
```

26.3.1.2 `valarray` assignment [lib.valarray.assign]

```
valarray<T>& operator=(const valarra<T>y&);
```

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<T>& operator=(const slice_array<T>&);
valarray<T>& operator=(const gslice_array<T>&);
valarray<T>& operator=(const mask_array<T>&);
valarray<T>& operator=(const indirect_array<T>&);
```

2 These operators allow the results of a generalized subscripting operation to be assigned directly to a `valarray`.

26.3.1.3 `valarray` element access [lib.valarray.access]

```
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 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.[202)]

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.

26.3.1.4 `valarray` subset operations [lib.valarray.sub]

```
valarray<T>         operator[](slice) const;
slice_array<T>      operator[](slice);
valarray<T>         operator[](const gslice&) const;
gslice_array<T>     operator[](const gslice&);
valarray<T>         operator[](const valarray<bool>&) const;
mask_array<T>       operator[](const valarray<bool>&);
valarray<T>         operator[](const valarray<size_t>&) const;
indirect_array<T>   operator[](const valarray<size_t>&);
```

1 Each of these operations returns a subset of the array. The `const`-qualified versions return this subset as a new `valarray`. The non-`const` versions return a class template object which has reference semantics to the original array.

26.3.1.5 `valarray` unary operators [lib.valarray.unary]

```
valarray<T> operator+() const;
valarray<T> operator-() const;
valarray<T> operator~() const;
valarray<T> operator!() const;
```

[202)] 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.

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 array. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding element of the array.

26.3.1.6 `valarray` computed assignment [lib.valarray.cassign]

```
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>&);
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>&);
```

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 array is then returned by reference.

3 If the array and the argument array do not have the same length, the behavior is undefined. The appearance of an array on the left hand side of a computed assignment does *not* invalidate references or pointers.

```
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&);
```

4 Each of these operators may only be instantiated for a type *T* to which the indicated operator can be applied.

5 Each of these operators applies the indicated operation to each element of the array and the scalar argument.

6 The array is then returned by reference.

7 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.

26.3.1.7 `valarray` member functions [lib.valarray.members]

```
size_t length() const;
```

1 This function returns the number of elements in the array.

```
operator T*();
operator const T*() const;
```

2 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`.

3 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.[203)]

```
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.

4 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.

```
void fill(const T&);
```

This function assigns the value of the argument to all the elements of the array. The length of the array is not changed, nor are any pointers or references to the elements of the array invalidated.

```
valarray<T> shift(int) const;
```

5 This function returns an array whose length is identical to the array, but whose element values are shifted the number of places indicated by the argument.

6 A positive argument value results in a left shift, a negative value in a right shift, and a zero value in no shift.

7 [*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. —*end example*]

```
valarray<T> cshift(int) const;
```

8 This function returns an array whose length is identical to the array, but whose element values are shifted in a circular fashion the number of places indicated by the argument.

9 A positive argument value results in a left shift, a negative value in a right shift, and a zero value in no shift.

```
valarray<T> apply(T func(T)) const;
valarray<T> apply(T func(const T&)) const;
```

10 These functions return an array whose length is equal to the array. Each element of the returned array is assigned the value returned by applying the argument function to the corresponding element of the array.

[203)] This form of access is essential for reusability and cross-language programming.

```
void free();
```

11 This function sets the length of an array to zero.[204)]

26.3.2 `valarray` non-member operations [lib.valarray.nonmembers]

26.3.2.1 `valarray` binary operators [lib.valarray.binary]

```
template<class T> valarray<T> operator* (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator/ (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator% (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator+ (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator- (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator^ (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator& (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator| (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator<<(const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator>>(const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator&&(const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator||(const valarray<T>&, const valarray<T>&);
```

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.

204) An implementation may reclaim the storage used by the array when this function is called.

```
template<class T> valarray<T> operator* (const valarray<T>&, const T&);
template<class T> valarray<T> operator* (const T&, const valarray<T>&);
template<class T> valarray<T> operator/ (const valarray<T>&, const T&);
template<class T> valarray<T> operator/ (const T&, const valarray<T>&);
template<class T> valarray<T> operator% (const valarray<T>&, const T&);
template<class T> valarray<T> operator% (const T&, const valarray<T>&);
template<class T> valarray<T> operator+ (const valarray<T>&, const T&);
template<class T> valarray<T> operator+ (const T&, const valarray<T>&);
template<class T> valarray<T> operator- (const valarray<T>&, const T&);
template<class T> valarray<T> operator- (const T&, const valarray<T>&);
template<class T> valarray<T> operator^ (const valarray<T>&, const T&);
template<class T> valarray<T> operator^ (const T&, const valarray<T>&);
template<class T> valarray<T> operator& (const valarray<T>&, const T&);
template<class T> valarray<T> operator& (const T&, const valarray<T>&);
template<class T> valarray<T> operator| (const valarray<T>&, const T&);
template<class T> valarray<T> operator| (const T&, const valarray<T>&);
template<class T> valarray<T> operator<<(const valarray<T>&, const T&);
template<class T> valarray<T> operator<<(const T&, const valarray<T>&);
template<class T> valarray<T> operator>>(const valarray<T>&, const T&);
template<class T> valarray<T> operator>>(const T&, const valarray<T>&);
template<class T> valarray<T> operator&&(const valarray<T>&, const T&);
template<class T> valarray<T> operator&&(const T&, const valarray<T>&);
template<class T> valarray<T> operator||(const valarray<T>&, const T&);
template<class T> valarray<T> operator||(const T&, const valarray<T>&);
```

4 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*.

5 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.

26.3.2.2 `valarray` comparison operators [lib.valarray.comparison]

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

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.

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

4 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*.

5 Each of these operators returns a *bool* array whose length is equal to the length of the array argument. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding element of the array and the scalar argument.

26.3.2.3 `valarray` min and max functions [lib.valarray.min.max]

```
template<class T> T min(const valarray<T>& a);
template<class T> T max(const valarray<T>& a);
```

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 (`a.min()`) or maximum (`a..max()`) value found in the argument array *a*.

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.

26.3.2.4 `valarray` transcendentals [lib.valarray.transcend]

```
template<class T> valarray<T> abs  (const valarray<T>&);
template<class T> valarray<T> acos (const valarray<T>&);
template<class T> valarray<T> asin (const valarray<T>&);
template<class T> valarray<T> atan (const valarray<T>&);
template<class T> valarray<T> atan2(const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> atan2(const valarray<T>&, const T&);
template<class T> valarray<T> atan2(const T&, const valarray<T>&);
template<class T> valarray<T> cos  (const valarray<T>&);
template<class T> valarray<T> cosh (const valarray<T>&);
template<class T> valarray<T> exp  (const valarray<T>&);
template<class T> valarray<T> log  (const valarray<T>&);
template<class T> valarray<T> log10(const valarray<T>&);
template<class T> valarray<T> pow  (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> pow  (const valarray<T>&, const T&);
template<class T> valarray<T> pow  (const T&, const valarray<T>&);
template<class T> valarray<T> sin  (const valarray<T>&);
template<class T> valarray<T> sinh (const valarray<T>&);
template<class T> valarray<T> sqrt (const valarray<T>&);
template<class T> valarray<T> tan  (const valarray<T>&);
template<class T> valarray<T> tanh (const valarray<T>&);
```

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*.

26.3.3 Class `slice` [lib.class.slice]

```
namespace std {
  class slice {
  public:
    slice();
    slice(size_t, size_t, size_t);

    size_t start() const;
    size_t length() const;
    size_t 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.[205)]

26.3.3.1 `slice` constructors [lib.cons.slice]

```
slice();
slice(size_t start, size_t length, size_t 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.

205) C++ programs may instantiate this class.

2 [*Example:* `slice(3, 8, 2)` constructs a slice which selects elements 3, 5, 7, ... 17 from an array. —*end example*]

26.3.3.2 `slice` access functions [lib.slice.access]

```
size_t start() const;
size_t length() const;
size_t stride() const;
```

1 These functions return the start, length, or stride specified by a `slice` object.

26.3.4 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&);
   ~slice_array();
  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 [*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` are 1, 4, ..., 13. —*end example*]

3 [*Note:* C++ programs 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. —*end note*]

26.3.4.1 `slice_array` constructors [lib.cons.slice.arr]

```
slice_array();
slice_array(const slice_array&);
```

1 The `slice_array` template has no public constructors. These constructors are declared to be private. These constructors need not be defined.

26.3.4.2 `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.

26.3.4.3 `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.

26.3.4.4 `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.

26.3.5 The `gslice` class [lib.class.gslice]

```
namespace std {
  class gslice {
  public:
    gslice();
    gslice(size_t s, const valarray<size_t>& l, const valarray<size_t>& d);

    size_t           start() const;
    valarray<size_t> length() const;
    valarray<size_t> 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 $k = s + \sum_j i_j d_j$ where the multidimensional indices i_j range in value from 0 to $l_{ij} - 1$.

3 [*Example:* The `gslice` specification

```
start  = 3
length = {2, 4, 3}
stride = {19, 4, 1}
```

yields the sequence of one-dimensional indices

$$k = 3 + (0,1) \times 19 = (0,1,2,3) \times 4 + (0,1,2) \times 1$$

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. —*end example*]

4 It is possible to have degenerate generalized slices in which an address is repeated.

5 [*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),
            ...
```

—*end example*]

6 If a degenerate slice is used as the argument to the non-`const` version of `operator[](const gslice&)`, the resulting behavior is undefined.

26.3.5.1 `gslice constructors` [lib.gslice.cons]

```
gslice();
gslice(size_t start, const valarray<size_t>& lengths,
                           const valarray<size_t>& 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.

26.3.5.2 `gslice` access functions [lib.gslice.access]

```
size_t           start()  const;
valarray<size_t> length() const;
valarray<size_t> stride() const;
```

These access functions return the representation of the start, lengths, or strides specified for the `gslice`.

26.3.6 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&);
   ~gslice_array();
  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:* C++ programs 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. —*end note*]

26.3.6.1 `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.

26.3.6.2 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.

26.3.6.3 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.

26.3.6.4 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.

26.3.7 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&);
   ~mask_array();
  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:* 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. —*end note*]

26.3.7.1 `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.

26.3.7.2 `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.

26.3.7.3 `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.

26.3.7.4 `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.

26.3.8 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&);
   ~indirect_array();
  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:* 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. —*end note*]

26.3.8.1 `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.

26.3.8.2 `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 [*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. —*end example*]

26.3.8.3 `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.

26.3.8.4 `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.

26.4 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);
}
```

26.4.1 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.[206)]

Requires: `binary_op` shall not cause side effects.

26.4.2 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

[206)] `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.

the range `[first2, first2 + (last - first))` in order.
Requires: `binary_op1` and `binary_op2` shall not cause side effects.

26.4.3 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`.

26.4.4 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`.

26.5 C Library [lib.c.math]

1 Headers `<cmath>` and `<cstdlib>` (`abs()`, `div()`, `rand()`, `srand()`).

Table 64—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 64—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, with the following additions:

3 In addition to the `int` versions of certain math functions in `<cstdlib>`, C++ adds `long` overloaded versions of these functions, with the same semantics.

4 The added signatures are:

```
long   abs(long);          // labs()
ldiv_t div(long, long);    // ldiv()
```

5 In addition to the `double` versions of the math functions in `<cmath>`, C++ adds `float` and `long double` overloaded versions of these functions, with the same semantics.

6 The added signatures are:

```
float abs  (float);
float acos (float);
float asin (float);
float atan (float);
float atan2(float, float);
float ceil (float);
float cos  (float);
float cosh (float);
float exp  (float);
float fabs (float);
float floor(float);
float fmod (float, float);
float frexp(float, int*);
float modf (float, float*);
float ldexp(float, int);
float log  (float);
float log10(float);
float pow  (float, float);
float pow  (float, int);
float sin  (float);
float sinh (float);
float sqrt (float);
float tan  (float);
float tanh (float);


double abs(double);         // fabs()
double pow(double, int);
```

```
long double abs  (long double);
long double acos (long double);
long double asin (long double);
long double atan (long double);
long double atan2(long double, long double);
long double ceil (long double);
long double cos  (long double);
long double cosh (long double);
long double exp  (long double);
long double fabs (long double);
long double floor(long double);
long double frexp(long double, int*);
long double fmod (long double, long double);
long double frexp(long double, int*);
long double log  (long double);
long double log10(long double);
long double modf (long double, long double*);
long double pow  (long double, long double);
long double pow  (long double, int);
long double sin  (long double);
long double sinh (long double);
long double sqrt (long double);
long double tan  (long double);
long double tanh (long double);
```

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 65:

Table 65—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	`<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 any unit element which, treated sequentially, can represent text. The term does not only mean `char` and `wchar_t` type objects, but any value which can be represented by a type which provides the definitions specified in (21.1.1.1).

— **character container type** Character container type is a class or a type used to represent a *character*. It is used for one of the template parameter of the iostream class templates.

— **iostream class templates** The iostream class templates are templates defined in this clause that take two template arguments: `charT` and `traits`. The argument `charT` is a character container class, and the argument `traits` is a structure which defines additional characteristics and functions of the character type represented by `charT` necessary to implement the iostream class templates.

— **narrow-oriented iostream classes** The narrow-oriented iostream classes are the instantiations of the iostream class templates on the character container class `char` and the default value of the `traits` parameter. 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 instantiations of the iostream class templates on the character container class `wchar_t` and the default value of the `traits` parameter. (27.3.2).

— **repositional streams and arbitrary-positional streams** A *repositional stream*, can seek to only the position where we previously encountered. On the other hand, an *arbitrary-positional* stream can seek to any integral position within the length of the stream. Every arbitrary-positional stream is repositional.

27.1.2 Type requirements [lib.iostreams.type.reqmts]

1 There are several types and functions needed for implementing the iostream class templates. 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 iostream class templates expects to the character container class.

27.1.2.1 Type *CHAR_T* [lib.iostreams.char.t]

1 Those C++ programs that 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 string class templates and the iostream class templates.

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.[207] 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 The effect of passing to any function defined in this clause an ***OFF_T*** value not obtained from a function defined in this clause (for example, assigned an arbitrary integer), is undefined, except where otherwise noted.

4 Convertible to type ***POS_T***.[208] 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*.[209]

[207] It is usually a synonym for one if the signed basic integral types whose representation at least as many bits as type `long`.
[208] An implementation may use the same type for both ***OFF_T*** and ***POS_T***.
[209] 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.

3 With a stream buffer for a repositional stream (but not an arbitrary-positional stream), a C++ program can either obtain the current position of the stream buffer or specify the previous position previously obtained

4 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 66.

5 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`.

Table 66—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`

6 The behavior of the stream after restoring the position with a ***POS_T*** value modified using any other arithmetic operations is undefined.

7 The stream operations whose return type is ***POS_T*** may return ***POS_T***(***OFF_T***(–1)) as an *invalid* ***POS_T*** *value* to signal an error.

8 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.

27.2 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 istream cin;
  extern ostream cout;
  extern ostream cerr;
  extern ostream clog;

  extern wistream win;
  extern wostream wout;
  extern wostream werr;
  extern wostream wlog;
}
```

1 The header `<iostream>` declares objects that associate objects with the standard C streams provided for by the functions declared in `<cstdio>` (27.8.2).

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. 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.[210)]

27.3.1 Narrow stream objects [lib.narrow.stream.objects]

```
istream cin;
```

210) Constructors and destructors for static objects can access these objects to read input from `stdin` or write output to `stdout` or `stderr`.

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`.

```
ostream cout;
```

3 The object `cout` controls output to an unbuffered stream buffer associated with the object `stdout`, declared in `<cstdio>` (27.8.2).

```
ostream cerr;
```

4 The object `cerr` controls output to an unbuffered stream buffer associated with the object `stderr`, declared in `<cstdio>` (27.8.2).

5 After the object `cerr` is initialized, `cerr.flags() & unitbuf` is nonzero.

```
ostream clog;
```

6 The object `clog` controls output to a stream buffer associated with the object `stderr`, declared in `<cstdio>` (27.8.2).

27.3.2 Wide stream objects [lib.wide.stream.objects]

```
wistream 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`.

```
wostream wout;
```

3 The object `wout` controls output to an unbuffered stream buffer associated with the object `stdout`, declared in `<cstdio>` (27.8.2).

```
wostream werr;
```

4 The object `werr` controls output to an unbuffered stream buffer associated with the object `stderr`, declared in `<cstdio>` (27.8.2).

5 After the object `werr` is initialized, `werr.flags() & unitbuf` is nonzero.

```
wostream wlog;
```

6 The object `wlog` controls output to a stream buffer associated with the object `stderr`, declared in `<cstdio>` (27.8.2).

27.4 Iostreams base classes [lib.iostreams.base]

Header `<ios>` synopsis

```
#include <stdexcept>    // for exception

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);

// 27.4.5.2 adjustfield:
  ios_base& internal   (ios_base& str);
  ios_base& left       (ios_base& str);
  ios_base& right      (ios_base& str);

// 27.4.5.3 basefield:
  ios_base& dec        (ios_base& str);
  ios_base& hex        (ios_base& str);
  ios_base& oct        (ios_base& str);

// 27.4.5.4 floatfield:
  ios_base& fixed      (ios_base& str);
  ios_base& scientific (ios_base& str);
}
```

27.4.1 Types [lib.stream.types]

```
typedef OFF_T streamoff;
```

1 The type `streamoff` is an implementation-defined type that satisfies the requirements of type ***OFF_T*** (27.1.2.3).

```
typedef OFF_T wstreamoff;
```

2 The type `wstreamoff` is an implementation-defined type that satisfies the requirements of type ***OFF_T*** (27.1.2.3).

```
typedef POS_T streampos;
```

3 The type `streampos` is an implementation-defined type that satisfies the requirements of type ***POS_T*** (27.1.2.4).

```
typedef POS_T wstreampos;
```

4 The type `wstreampos` is an implementation-defined type that satisfies the requirements of type ***POS_T*** (27.1.2.4).

```
typedef INT_T streamsize;
```

5 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.[211]

27.4.2 Template struct `ios_traits` [lib.ios.traits]

```
namespace std {
  template <class charT> struct ios_traits<charT> {
  // 27.4.2.1 Types:
    typedef charT char_type;
    typedef INT_T int_type;
    typedef POS_T pos_type;
    typedef OFF_T off_type;
    typedef To be specified state_type;

  // 27.4.2.2 values:
    static char_type  eos();
    static int_type   eof();
    static int_type   not_eof(char_type c);
    static char_type  newline();
    static size_t     length(const char_type* s);

  // 27.4.2.3 tests:
    static bool       eq_char_type(char_type, char_type);
    static bool       eq_int_type (int_type, int_type);
    static bool       is_eof(int_type);
    static bool       is_whitespace(const ctype<char_type> ctype&, char_type c);
```

[211] `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`).

```
    // 27.4.2.4 conversions:
      static char_type  to_char_type(int_type);
      static int_type   to_int_type (char_type);
      static char_type* copy(char_type* dst, const char_type* src, size_t n) ;

      static state_type get_state(pos_type pos);
      static pos_type   get_pos  (streampos fpos, state_type state);
    };
  }
```

1 The template struct `ios_traits<charT>` is a traits class which maintains the definitions of the types and functions necessary to implement the iostream class templates. 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 shall provide the following two instantiations of `ios_traits`:

```
struct ios_traits<char>;
struct ios_traits<wchar_t>;
```

27.4.2.1 `ios_traits` types [lib.ios.traits.types]

`state_type` is an implementation-defined value-oriented type. It holds the *conversion state*, and is compatible with the function `locale::codecvt()`.

27.4.2.2 `ios_traits` value functions [lib.ios.traits.values]

```
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.

```
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.

```
int_type not_eof(char_type c);
```

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()`.

```
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()`.

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

Effects: Determines the length of a null terminated character string pointed to by *s*.

27.4.2.3 `ios_traits` test functions [lib.ios.traits.tests]

`bool eq_char_type(char_type` *`c1`*`, char_type` *`c2`*`);`

Returns: `true` if *`c1`* and *`c2`* represent the same character.

`bool eq_int_type(int_type` *`c1`*`, int_type` *`c2`*`);`

Returns: `true` if *`c1`* and *`c2`* represent the same character.

`bool is_eof(int_type` *`c`*`);`

Returns: `true` if *`c`* represents the end-of-file.

`bool is_whitespace(char_type` *`c`*`, const ctype<char_type>&` *`ctype`*`);`

Returns: `true` if *`c`* represents a whitespace character. The default definition is as if it returns *`ctype`*`.isspace(`*`c`*`)`. (See also 27.6.1.1.2)

1 An implementation of the iostream class templates 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>`.

27.4.2.4 `ios_traits` conversion functions [lib.ios.traits.convert]

`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.

`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.

`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.

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

Returns: `0`.

`pos_type get_pos(streampos` *`fpos`*`, state_type` *`state`*`);`

Returns: `pos_type(`*`pos`*`)`.

27.4.3 Class `ios_base` [lib.ios.base]

```
namespace std {
  class ios_base {
  public:
    class failure;
    typedef T1 fmtflags;
    static const fmtflags boolalpha;
    static const fmtflags dec;
    static const fmtflags fixed;
    static const fmtflags hex;
    static const fmtflags internal;
    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;

  // 27.4.4.3 iostate flags:

    iostate exceptions() const;
    void exceptions(iostate except);

  // 27.4.3.2 fmtflags state:
    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);

  // 27.4.3.3 locales:
    locale imbue(const locale& loc);
    locale getloc() const;

  // 27.4.3.4 storage:
    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
  };
}
```

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.

3 [*Note:* 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. —*end note*]

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:
    explicit failure(const string& msg);
    virtual ~failure();
    virtual const char* what() const;
  };
}
```

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.

```
explicit failure(const string& msg);
```

Effects: Constructs an object of class `failure`, initializing the base class with `exception(`*`msg`*`)`.
Postcondition: `what() ==` *`msg`*`.str()`

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

Returns: The message *`msg`* with which the exception was created.

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 67:

Table 67—`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 right (final positions) of certain generated output;
`oct`	converts integer input or generates integer output in octal base;
`right`	adds fill characters on the left (initial 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 68:

Table 68—`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 69:

Table 69—`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 70:

Table 70—`openmode` effects

Element	Effect(s) if set
`app`	seek to end before each write
`ate`	open and seek to end immediately after opening
`binary`	perform input and output in binary mode (as opposed to text mode)
`in`	open for input
`out`	open for output
`trunc`	truncate an existing stream when opening

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 71:

Table 71—seekdir effects

Element	Meaning
`beg`	request a seek (for subsequent input or output) 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 eight objects declared in `<iostream>` (27.3) that associate file stream buffers with the standard C streams provided for by the functions declared in `<cstdio>` (27.8.2).

```
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`, `cout`, `cerr`, `clog` (27.3.1), `win`, `wout`, `werr`, and `wlog` (27.3.2). In any case, the function then adds one to the value stored in *`init_cnt`*.

```
~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 `ios_base fmtflags` state functions [lib.fmtflags.state]

```
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()`.

```
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()`.

```
void unsetf(fmtflags mask);
```

Effects: Clears *`mask`* in `flags()`.

```
int_type fill() const;
```

Returns: The character to use to pad (fill) an output conversion to the specified field width (27.6.2.4).

```
int_type fill(int_type fillch);
```

Postcondition: `&`*`fillch`* `== fill()`.
Returns: The previous value of `fill()`.

```
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()`.

```
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()`.

27.4.3.3 `ios_base locale` functions [lib.ios.base.locales]

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

Postcondition: *`loc`* `== getloc()`.
Returns: The previous value of `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.

27.4.3.4 `ios_base` storage functions [lib.ios.base.storage]

```
static int xalloc();
```

Returns: *`index`* `++`.

```
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.[212)]

```
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`*`]`.
Notes: After a subsequent call to `pword(int)` for the same object, the earlier return value may no longer be valid.

27.4.3.5 `ios_base` constructors [lib.ios.base.cons]

```
ios_base();
```

Effects: Constructs an object of class `ios_base`, assigning initial values to its member objects. The postconditions of this function are indicated in Table 72:

Table 72—`ios_base()` 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()`	`locale::classic()`
`index`	***???***
`iarray`	a null pointer
`parray`	a null pointer

212) 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.

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 {
  public:
  // Types:
    typedef charT                     char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;

    operator bool() const
    bool operator!() const
    iostate rdstate() const;
    void clear(iostate state = goodbit);
    void setstate(iostate state);
    bool good() const;
    bool eof()  const;
    bool fail() const;
    bool bad()  const;

  // 27.4.4.1 Constructor/destructor:
    explicit basic_ios(basic_streambuf<charT,traits>* sb);
    virtual ~basic_ios();

  // 27.4.4.2 Members:
    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);

    basic_ios& copyfmt(const basic_ios& rhs);

  // 27.4.3.3 locales:
    locale imbue(const locale& loc);

  protected:
    basic_ios();
    void init(basic_streambuf<charT,traits>* sb);
  };
}
```

27.4.4.1 basic_ios constructors [lib.basic.ios.cons]

`explicit 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` (27.4.3.5),

`void init(basic_streambuf<charT,traits>* sb);`

27.4.4.2 Member functions [lib.basic.ios.members]

```
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()`.

```
basic_streambuf<charT,traits>* rdbuf() const;
```

Returns: A pointer to the `streambuf` associated with the stream.

```
basic_streambuf<charT,traits>* rdbuf(basic_streambuf<charT,traits>* sb);
```

Postcondition: *`sb`* `== rdbuf()`.
Effects: Calls `clear()`.
Returns: The previous value of `rdbuf()`.

```
  // 27.4.3.3 locales:
locale imbue(const locale& loc);
```

Effects: Calls `ios_base::imbue(`*`loc`*`)` (27.4.3.3) and `rdbuf()->pubimbue(`*`loc`*`)` (27.5.2.2.1).

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

Effects: Assigns to the member objects of `*this` the corresponding member objects of *`rhs`*, except that:

— `rdstate()` is 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.[213)]

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`.

27.4.4.3 `basic_ios iostate` flags functions [lib.iostate.flags]

```
operator bool() const
```

Returns: `!fail()`.

```
bool operator!() const
```

Returns: `fail()`.

213) This suggests an infinite amount of copying, but the implementation can keep track of the maximum element of the arrays that is non-zero.

`basic_ios iostate` flags functions

```
iostate rdstate() const;
```

Returns: The control state of the stream buffer.

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

Postcondition: `state == rdstate()`.
Effects: If `(rdstate() & exceptions()) == 0`, returns. Otherwise, the function throws an object *`fail`* of class `basic_ios::failure` (27.4.3.1.1), constructed with implementation-defined argument values.

```
void setstate(iostate state) throw(failure);
```

Effects: Calls `clear(rdstate() | state)` (which may throw `basic_ios::failure` (27.4.3.1.1)).

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

Returns: `rdstate() == 0`

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

Returns: `true` if `eofbit` is set in `rdstate()`.

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

Returns: `true` if `failbit` or `badbit` is set in `rdstate()`.[214)]

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

Returns: `true` if `badbit` is set in `rdstate()`.

```
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()`.
Effects: Calls `clear(rdstate())`.

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`*.[215)]

214) Checking `badbit` also for `fail()` is historical practice.

`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*.
Notes: Does not affect any extractors.

`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);
```

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`*.

[215] 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`.

27.5 Stream buffers [lib.stream.buffers]

Header `<streambuf>` synopsis

```
#include <ios>   // for ios_traits

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]

1 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.

2 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).

3 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 Template class basic_streambuf<charT,traits> [lib.streambuf]

```
namespace std {
  template<class charT, class traits = ios_traits<charT> >
  class basic_streambuf {
  public:
  // Types:
    typedef charT                     char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;

    virtual ~basic_streambuf();

  // 27.5.2.2.1 locales:
    locale   pubimbue(const locale &loc);
    locale   getloc() const;

  // 27.5.2.2.2 buffer and positioning:
    basic_streambuf<char_type,traits>*
             pubsetbuf(char_type* s, streamsize n);
    pos_type pubseekoff(off_type off, ios_base::seekdir way,
                        ios_base::openmode which = ios_base::in | ios_base::out);
    pos_type pubseekpos(pos_type sp,
                        ios_base::openmode which = ios_base::in | ios_base::out);
    int      pubsync();

  // Get and put areas:
  // 27.5.2.2.3 Get area:
    int      in_avail();
    int_type snextc();
    int_type sbumpc();
    int_type sgetc();
    int      sgetn(char_type* s, streamsize n);

  // 27.5.2.2.4 Putback:
    int_type sputbackc(char_type c);
    int      sungetc();

  // 27.5.2.2.5 Put area:
    int      sputc(char_type c);
    int_type sputn(const char_type* s, streamsize n);

  protected:
    basic_streambuf();

  // 27.5.2.3.1 Get area:
    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);

  // 27.5.2.3.2 Put area:
    char_type* pbase() const;
    char_type* pptr() const;
    char_type* epptr() const;
    void       pbump(int n);
    void       setp(char_type* pbeg, char_type* pend);
  // 27.5.2.4 virtual functions:
  // 27.5.2.4.1 Locales:
    virtual void imbue(const locale &loc);

  // 27.5.2.4.2 Buffer management and positioning:
    virtual basic_streambuf<char_type,traits>*
                     setbuf(char_type* s, streamsize n);
    virtual pos_type seekoff(off_type off, ios_base::seekdir way,
              ios_base::openmode which = ios_base::in | ios_base::out);
    virtual pos_type seekpos(pos_type sp,
              ios_base::openmode which = ios_base::in | ios_base::out);
    virtual int      sync();

  // 27.5.2.4.3 Get area:
    virtual int        showmanyc();
    virtual streamsize xsgetn(char_type* s, streamsize n);
    virtual int_type   underflow();
    virtual int_type   uflow();

  // 27.5.2.4.4 Putback:
    virtual int_type   pbackfail(int_type c = traits::eof());

  // 27.5.2.4.5 Put area:
    virtual streamsize xsputn(const char_type* s, streamsize n);
    virtual int_type   overflow (int_type c = traits::eof());
  };
}
```

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`.

27.5.2.1 `basic_streambuf` constructors [lib.streambuf.cons]

```
basic_streambuf();
```

Effects: Constructs an object of class `basic_streambuf<charT,traits>` and initializes:[216)]

— all its pointer member objects to null pointers,

— the `getloc()` member to the return value of `locale::classic()`.

Notes: Once the `getloc()` member is initialized, results of calling locale member functions, and of members of facets so obtained, can safely be cached until the next time the member `imbue` is called.

216) 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 `basic_streambuf` public member functions [lib.streambuf.members]

27.5.2.2.1 Locales [lib.streambuf.locales]

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

Postcondition: *`loc`* `== getloc()`.
Effects: Calls `imbue(`*`loc`*`)`.
Returns: Previous value of `getloc()`.

```
locale getloc() const;
```

Returns: If `pubimbue()` has ever been called, then the last value of *`loc`* supplied, otherwise classic `"C"` locale `locale::classic()`. If called after `pubimbue()` has been called but before `pubimbue` has returned (i.e. from within the call of `imbue()`) then it returns the previous value.

27.5.2.2.2 Buffer management and positioning [lib.streambuf.buffer]

```
basic_streambuf<char_type,traits>* pubsetbuf(char_type* s, streamsize n);
```

Returns: `setbuf(`*`s`*`,`*`n`*`)`.

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

Returns: `seekoff(`*`off`*`,`*`way`*`,`*`which`*`)`.

```
pos_type pubseekpos(pos_type sp,
               ios_base::openmode which = ios_base::in | ios_base::out);
```

Returns: `seekpos(`*`sp`*`,`*`which`*`)`.

```
int pubsync();
```

Returns: `sync()`.

27.5.2.2.3 Get area [lib.streambuf.pub.get]

```
int in_avail();
```

Returns: If a read position is available, returns `gend() - gnext()`. Otherwise returns `showmanyc()` (27.5.2.4.3).

```
int_type snextc();
```

Effects: Calls `sbumpc()`.
Returns: if that function returns `traits::eof()`, returns `traits::eof()`. Otherwise, returns `sgetc()`.
Notes: Uses `traits::eof()`.

```
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.

```
int_type sgetc();
```

Returns: If the input sequence read position is not available, returns `underflow()`. Otherwise, returns `char_type(*gptr())`.

```
int sgetn(char_type* s, streamsize n);
```

Returns: `xsgetn(s,n)`.

27.5.2.2.4 Putback [lib.streambuf.pub.pback]

```
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()`.

```
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()`.

27.5.2.2.5 Put area [lib.streambuf.pub.put]

```
int sputc(char_type 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()`.

```
int_type sputn(const char_type* s, streamsize n);
```

Returns: `xsputn(s,n)`.

27.5.2.3 `basic_streambuf` protected member functions [lib.streambuf.protected]

27.5.2.3.1 Get area access [lib.streambuf.get.area]

```
char_type* eback() const;
```

Returns: The beginning pointer for the input sequence.

```
char_type* gptr() const;
```

Returns: The next pointer for the input sequence.

```
char_type* egptr() const;
```

Returns: The end pointer for the output sequence.

```
void gbump(int n);
```

Effects: Advances the next pointer for the input sequence by *n*.

```
void setg(char_type* gbeg, char_type* gnext, char_type* gend);
```

Postconditions: *gbeg* `== eback()`, *gnext* `== gptr()`, and *gend* `== egptr()`.

27.5.2.3.2 Put area access [lib.streambuf.put.area]

```
char_type* pbase() const;
```

Returns: The beginning pointer for the output sequence.

```
char_type* pptr() const;
```

Returns: The next pointer for the output sequence.

```
char_type* epptr() const;
```

Returns: The end pointer for the output sequence.

```
void pbump(int n);
```

Effects: Advances the next pointer for the output sequence by *n*.

```
void setp(char_type* pbeg, char_type* pend);
```

Postconditions: *pbeg* `== pbase()`, *pbeg* `== pptr()`, and *pend* `== epptr()`.

27.5.2.4 `basic_streambuf` virtual functions [lib.streambuf.virtuals]

27.5.2.4.1 Locales [lib.streambuf.virt.locales]

```
void imbue(const locale&)
```

Effects: Change any translations based on locale.
Note: Allows the derived class to be informed of changes in locale at the time they occur. Between invocations of this function a class derived from streambuf can safely cache results of calls to locale functions and to members of facets so obtained.
Default behavior: Does nothing.

27.5.2.4.2 Buffer management and positioning [lib.streambuf.virt.buffer]

```
basic_streambuf* setbuf(char_type* s, streamsize n);
```

Effects: Performs an operation that is defined separately for each class derived from `basic_streambuf` in this clause (27.7.1.3, 27.8.1.4).
Default behavior: Returns `this`.

```
pos_type seekoff(off_type off, ios_base::seekdir way,
                 ios_base::openmode which
                  = ios_base::in | ios_base::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 (27.7.1.3, 27.8.1.4).

Default behavior: Returns an object of class `pos_type` that stores an *invalid stream position* (27.1.1).

```
pos_type seekpos(pos_type sp,
                             ios_base::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 (_lib.stringbuf::seekpos_, _lib.filebuf::seekpos_).
Default behavior: Returns an object of class `pos_type` that stores an *invalid stream position*.

```
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 (27.8.1.4).
Default behavior: Returns zero.

27.5.2.4.3 Get area [lib.streambuf.virt.get]

```
int showmanyc();
```
[217)]

Returns: a guaranteed lower bound on the number of characters that can be read from the input sequence before a call to `uflow()` or `underflow()` returns `traits::eof()`. A positive return value of indicates that the next such call will not return `traits::eof()`.[218)]
Default behavior: Returns zero.
Notes: Uses `traits::eof()`.

```
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 `traits::eof()`.
Returns: The number of characters assigned.[219)]
Notes: Uses `traits::eof()`.

```
int_type underflow();
```

Notes: The public members of `basic_streambuf` call this virtual function 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.

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.

217) The morphemes of `showmany` are "es-how-many-see", not "show-manic".
218) The next such call might fail by throwing an exception. The intention is that the next call will return ‘‘immediately.’’
219) Classes derived from `basic_streambuf` can provide more efficient ways to implement `xsgetn()` and `xsputn()` by overriding these definitions from the base class.

b) Some sequence (possibly empty) of characters read from the input sequence.

2 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.

3 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()`.

Effects: 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.

4 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)

Returns: `traits::eof()` to indicate failure.
Default behavior: Returns `traits::eof()`.

```
int_type uflow();
```

Requires: The constraints are the same as for `underflow()`, 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(traits::eof())`. If `underflow()` returns `traits::eof()`, returns `traits::eof()`. Otherwise, does `gbump(-1)` and returns `*gptr()`.
Returns: `traits::not_eof(c)`.
Notes: Uses `traits::eof()`.

27.5.2.4.4 Putback [lib.streambuf.virt.pback]

```
int_type pbackfail(int c = traits::eof());
```

Notes: The public functions of `basic_streambuf` call this virtual function only when `gptr()` is null, `gptr() == eback()`, or `*gptr() != c`. Other calls shall also satisfy that constraint.
The *pending sequence* is defined as for `underflow()`, with the modifications that

— If `c == traits::eof()` then the input sequence is backed up one character before the pending sequence is determined.

— If `c != traits::eof()` then *c* is prepended. Whether the input sequence is backed up or modified in any other way is unspecified.

Postcondition: On return, the constraints of `gptr()`, `eback()`, and `pptr()` are the same as for `underflow()`.
Returns: `traits::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.

Returns some value other than `traits::eof()` to indicate success.

Default behavior: Returns `traits::eof()`.

27.5.2.4.5 Put area [lib.streambuf.virt.put]

```
streamsize xsputn(const char_type* s, streamsize n);
```

Effects: 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 `traits::eof()`.

Returns: The number of characters written.

```
int_type overflow(int_type c = traits::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 == traits::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.

Requires: 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[220)]

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: `traits::eof()` or throws an exception if the function fails.
Otherwise, returns some value other than `traits::eof()` to indicate success.[221)]

Default behavior: Returns `traits::eof()`.

27.6 Formatting and manipulators [lib.iostream.format]

Header `<istream>` synopsis

220) That is, for each class derived from an instance of `basic_streambuf` in this clause (27.7.1, 27.8.1.1), a specification of how consuming a character effects the associated output sequence is given. There is no requirement on a program-defined class.

221) Typically, `overflow` returns *c* to indicate success.

```
#include <ios>  // for ios_traits

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

```
#include <ios>  // for ios_traits

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);
}
```

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:
  // Types:
    typedef charT                     char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;

  // _lib.istream.cons_ Constructor/destructor:
    explicit basic_istream(basic_streambuf<charT,traits>* sb);
    virtual ~basic_istream();

  // 27.6.1.1.2 Prefix/suffix:
    bool ipfx(bool noskipws = false);
    void isfx();

  // 27.6.1.2 Formatted input:
    basic_istream<charT,traits>& operator>>
        (basic_istream<charT,traits>& (*pf)(basic_istream<charT,traits>&))
    basic_istream<charT,traits>& operator>>
        (basic_ios<charT,traits>& (*pf)(basic_ios<charT,traits>&))
    basic_istream<charT,traits>& operator>>(char_type* s);

    basic_istream<charT,traits>& operator>>(char_type& c);
    basic_istream<charT,traits>& operator>>(bool& n);
    basic_istream<charT,traits>& operator>>(short& n);
    basic_istream<charT,traits>& operator>>(unsigned short& n);
    basic_istream<charT,traits>& operator>>(int& n);
    basic_istream<charT,traits>& operator>>(unsigned int& n);
    basic_istream<charT,traits>& operator>>(long& n);
    basic_istream<charT,traits>& operator>>(unsigned long& n);
    basic_istream<charT,traits>& operator>>(float& f);
    basic_istream<charT,traits>& operator>>(double& f);
    basic_istream<charT,traits>& operator>>(long double& f);

    basic_istream<charT,traits>& operator>>(void*& p);
    basic_istream<charT,traits>& operator>>
        (basic_streambuf<char_type,traits>* sb);

  // 27.6.1.3 Unformatted input:
    streamsize gcount() const;
    int_type get();
    basic_istream<charT,traits>& get(char_type& c);
    basic_istream<charT,traits>& get(char_type* s, streamsize n,
                      char_type delim = traits::newline());
    basic_istream<charT,traits>& get(basic_streambuf<char_type,traits>& sb,
                      char_type delim = traits::newline());

    basic_istream<charT,traits>& getline(char_type* s, streamsize n,
                          char_type delim = traits::newline());

    basic_istream<charT,traits>& ignore
        (streamsize n = 1, int_type delim = traits::eof());
    int_type                     peek();
    basic_istream<charT,traits>& read    (char_type* s, streamsize n);
    streamsize                   readsome(char_type* s, streamsize n);
    basic_istream<charT,traits>& putback(char_type c);
    basic_istream<charT,traits>& unget();
    int sync();

    pos_type tellg();
    basic_istream<charT,traits>& seekg(pos_type&);
    basic_istream<charT,traits>& seekg(off_type&, ios_base::seekdir);
  };
}
```

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 `rdbuf()->sbumpc()` or `rdbuf()->sgetc()`. They may use other public members of `istream` except that they do not invoke any virtual members of `rdbuf()` except `uflow()`.

3 If `rdbuf()->sbumpc()` or `rdbuf()->sgetc()` returns `traits::eof()`, then the input function, except as explicitly noted otherwise, completes its actions and does `setstate(eofbit)`, which may throw `ios_base::failure` (27.4.4.3), 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 `exception()` rethrows the exception without completing its actions.

27.6.1.1.1 `basic_istream` constructors [lib.basic.istream.cons]

```
explicit 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)` (27.4.4.1).
Postcondition: `gcount() == 0`

```
virtual ~basic_istream();
```

Effects: Destroys an object of class `basic_istream`.
Notes: Does not perform any operations of `rdbuf()`.

27.6.1.1.2 `basic_istream` prefix and suffix [lib.istream.prefix]

```
bool ipfx(bool noskipws = false);
```

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.[222] 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.

[222] The call `tie()->flush()` does not necessarily occur if the function can determine that no synchronization is necessary.

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)` (which may throw `ios_base::failure` (27.4.4.3)) and returns `false`.[223)]

2 [*Example:* 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()) != traits::eof()) {
      if (!traits::is_whitespace (c,ctype)==0) {
        rdbuf()->sputbackc (c);
        break;
      }
    }
    ...
 }
```

—*end example*]

3 When using `ios_traits<char>` or `ios_traits<wchar_t>`, the behavior of the function `traits::is_whitespace()` is ‘‘as if’’ it invokes:

```
ctype = getloc().use<ctype<charT> >().is(ctype<charT>::space, c);
```

(see 27.4.2.3); otherwise, the behavior of the function `traits::is_whitespace()` is unspecified.

4 [*Example:* Those C++ programs that 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)...
    }
};
```

—*end example*]

`void isfx();`

Effects: None.

223) The functions `ipfx(int)` and `isfx()` can also perform additional implementation-dependent operations.

27.6.1.2 Formatted input functions [lib.istream.formatted]

27.6.1.2.1 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<>` (22.2.2.1) 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 (*this, 0, *this, loc, tmp);
if ((TYPE)tmp != tmp) { // set fail bit...
} else val = (TYPE)tmp;
```

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 ‘‘as if’’ for an equivalent call to the function `::fscanf()`[224] operating with the global locale set to `getloc()`, with the

[224] The signature `fscanf(FILE*, const char*, ...)` is declared in `<cstdio>` (27.8.2)

following alterations:

— The formatted input function extracts characters from a stream buffer, rather than reading them from an input file.[225)]

— If `(flags() & skipws) == 0`, 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.

7 [*Note:* For conversion to an integral type other than a character type, the function determines the integral conversion specifier as indicated in Table 73:

Table 73—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`

—*end note*]

8 If the scan fails for any reason, the formatted input function calls `setstate(failbit)`, which may throw `ios_base::failure` (27.4.4.3).

27.6.1.2.2 `basic_istream::operator>>` [lib.istream::extractors]

```
basic_istream<charT,traits>& operator>>
    (basic_istream<charT,traits>& (*pf)(basic_istream<charT,traits>&))
```

Returns: `pf(*this)`.[226)]

```
basic_istream<charT,traits>& operator>>
    (basic_ios<charT,traits>& (*pf)(basic_ios<charT,traits>&));
```

Effects: Calls `pf(*this)`, then returns `*this`.[227)]

```
basic_istream<charT,traits>& operator>>(char_type* s);
```

Effects: Extracts characters and stores them into successive locations of an array whose first element is designated by *s*.[228)] 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;

225) The stream buffer can, of course, be associated with an input file, but it need not be.
226) See, for example, the function signature `ws(basic_istream&)` (27.6.1.4).
227) See, for example, the function signature `dec(basic_ios<charT,traits>&)` (27.4.5.3).
228) Note that this function is not overloaded on types `signed char` and `unsigned char`.

— end-of-file occurs on the input sequence;

— `traits::is_whitespace(`*`c,ctype`*`)` is true 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)`, which may throw `ios_base::failure` (27.4.4.3). In any case, it then stores a null character into the next successive location of the array and calls `width(0)`.
Returns: `*this`.
Notes: Uses `traits::eos()`.

```
basic_istream<charT,traits>& 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`.

```
basic_istream<charT,traits>& operator>>(bool& n);
```

Effects: Converts a boolean value, if one is available, and stores it in *`x`*.
Returns: `*this`.
Notes: Behaves as if:

```
getloc().use<num_get<charT,istreambuf_iterator<charT,traits> >().
  get(*this, 0, *this, getloc(), n);
```

[*Note:* `num_get<>::get()` just sets the `iostate` flags, without checking whether `failure()` should be thrown; so `operator>>()` needs to check that. —*end note*]

3 If *`flags`*`.flag() & ios_base::boolalpha` is `false`, `num_get<>::get()` (22.2.2) tries to read an integer value, which if found must be 0 or 1; if the `boolalpha` flag is `true`, it reads characters until it determines whether the `numpunct<>::truename()` or `falsename()` sequence[229] is present. In either case if an exact match is not found calls `setstate(failbit)`.

```
basic_istream<charT,traits>& operator>>(short& n);
```

Effects: Converts a signed short integer, if one is available, and stores it in *`n`*.
Returns: `*this`.

```
basic_istream<charT,traits>& operator>>(unsigned short& n);
```

Effects: Converts an unsigned short integer, if one is available, and stores it in *`n`*.
Returns: `*this`.

```
basic_istream<charT,traits>& operator>>(int& n);
```

Effects: Converts a signed integer, if one is available, and stores it in *`n`*.
Returns: `*this`.

```
basic_istream<charT,traits>& operator>>(unsigned int& n);
```

Effects: Converts an unsigned integer, if one is available, and stores it in *`n`*.

[229] The boolean value names for the default classic "`C`" locale are "`false`" and "`true`".

Returns: `*this`.

```
basic_istream<charT,traits>& operator>>(long& n);
```

Effects: Converts a signed long integer, if one is available, and stores it in *n*.
Returns: `*this`.

```
basic_istream<charT,traits>& operator>>(unsigned long& n);
```

Effects: Converts an unsigned long integer, if one is available, and stores it in *n*.
Returns: `*this`.

```
basic_istream<charT,traits>& operator>>(float& f);
```

Effects: Converts a `float`, if one is available, and stores it in *f*.
Returns: `*this`.

```
basic_istream<charT,traits>& operator>>(double& f);
```

Effects: Converts a `double`, if one is available, and stores it in *f*.
Returns: `*this`.

```
basic_istream<charT,traits>& operator>>(long double& f);
```

Effects: Converts a `long double`, if one is available, and stores it in *f*.
Returns: `*this`.

```
basic_istream<charT,traits>& operator>>(void*& p);
```

Effects: Converts a pointer to `void`, if one is available, and stores it in *p*.
Returns: `*this`.

```
basic_istream<charT,traits>& operator>>
    (basic_streambuf<charT,traits>* sb);
```

Requires: *sb* shall be non-null.
Effects: If *sb* is null, calls `setstate(badbit)`, which may throw `ios_base::failure` (27.4.4.3). Extracts characters from `*this` and inserts them in the output sequence controlled by *sb*. 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

4 If the function inserts no characters, it calls `setstate(failbit)`, which may throw `ios_base::failure` (27.4.4.3). If failure was due to catching an exception thrown while extracting characters from *sb* and `failbit` is on in `exceptions()` (27.4.4.3), then the caught exception is rethrown.
Returns: `*this`.

27.6.1.3 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.

```
streamsize gcount() const;
```

Returns: The number of characters extracted by the last unformatted input member function called for the object.

```
int_type get();
```

Effects: Extracts a character *c*, if one is available. Otherwise, the function calls `setstate(failbit)`, which may throw `ios_base::failure` (27.4.4.3),
Returns: *c* if available, otherwise `traits::eof()`.

```
basic_istream<charT,traits>& get(char_type& c);
```

Effects: Extracts a character, if one is available, and assigns it to *c*.[230] Otherwise, the function calls `setstate(failbit)` (which may throw `ios_base::failure` (27.4.4.3)).
Returns: `*this`.

```
basic_istream<charT,traits>& get(char_type*  s, streamsize n,
                  char_type delim = traits::newline());
```

Effects: Extracts characters and stores them into successive locations of an array whose first element is designated by *s*.[231] 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).

2 If the function stores no characters, it calls `setstate(failbit)` (which may throw `ios_base::failure` (27.4.4.3)). In any case, it then stores a null character into the next successive location of the array.
Returns: `*this`.

```
basic_istream<charT,traits>& get(basic_streambuf<char_type,traits>& sb,
                  char_type delim = traits::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).

[230] Note that this function is not overloaded on types `signed char`and `unsigned char`.
[231] Note that this function is not overloaded on types `signed char` and `unsigned char`.

3 If the function inserts no characters, it calls `setstate(failbit)`, which may throw `ios_base::failure` (27.4.4.3).
Returns: `*this`.

```
basic_istream<charT,traits>& getline(char_type* s, streamsize n,
                        char_type delim = traits::newline());
```

Effects: Extracts characters and stores them into successive locations of an array whose first element is designated by *s*.[232] 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);[233]

3) *n* - 1 characters are stored (in which case the function calls `setstate(failbit)`).

4 These conditions are tested in the order shown.[234]

5 If the function extracts no characters, it calls `setstate(failbit)` (which may throw `ios_base::failure` (27.4.4.3)).[235]

6 In any case, it then stores a null character (using `traits::eos()`) into the next successive location of the array.
Returns: `*this`.

7 [*Example:*

```
#include <iostream>

int main()
{
  using namespace std;
  const int line_buffer_size = 100;

  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;
  }
}
```

—*end example*]

232) Note that this function is not overloaded on types `signed char` and `unsigned char`.
233) Since the final input character is ‘‘extracted,’’ it is counted in the `gcount()`, even though it is not stored.
234) This allows an input line which exactly fills the buffer, without setting `failbit`. This is different behavior than the historical AT&T implementation.
235) This implies an empty input line will not cause `failbit` to be set.

```
basic_istream<charT,traits>&
    ignore(int n = 1, int_type delim = traits::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)`, which may throw `ios_base::failure` (27.4.4.3));

— *c* `==` *delim* for the next available input character *c* (in which case *c* is extracted).

Notes: The last condition will never occur if *delim* `== traits::eof()`.
Returns: `*this`.

```
int_type peek();
```

Returns: `traits::eof()` if `good()` is false. Otherwise, returns `rdbuf()->sgetc()`.

```
basic_istream<charT,traits>& 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*.[236] 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)`, which may throw `ios_base::failure` (27.4.4.3)).

Returns: `*this`.

```
streamsize readsome(char_type* s, streamsize n);
```

Effects: Extracts characters and stores them into successive locations of an array whose first element is designated by *s*.
Returns: A value based on `in_avail()`:

— If `in_avail() < 0`, calls `setstate(eofbit)` (which may throw `ios_base::failure` (27.4.4.3)), and returns zero;

— If `in_avail() == 0`, returns zero;

— If `in_avail() > 0`, returns `read(`*s*`, min(in_avail(),`*n*`))`.

```
basic_istream<charT,traits>& putback(char_type c);
```

Effects: Calls `rdbuf->sputbackc(`*c*`)`. If that function returns `traits::eof()`, calls `setstate(badbit)` (which may throw `ios_base::failure` (27.4.4.3)).
Returns: `*this`.

```
basic_istream<charT,traits>& unget();
```

Effects: Calls `rdbuf->sungetc()`. If that function returns `traits::eof()`, calls `setstate(badbit)` (which may throw `ios_base::failure` (27.4.4.3)).
Returns: `*this`.

[236] Note that this function is not overloaded on types `signed char` and `unsigned char`.

```
int sync();
```

Effects: If `rdbuf()` is a null pointer, returns `traits::eof()`. Otherwise, calls `rdbuf()->pubsync()` and, if that function returns `traits::eof()`, calls `setstate(badbit)` (which may throw `ios_base::failure` (27.4.4.3), and returns `traits::eof()`. Otherwise, returns zero.
Notes: Uses `traits::eof()`.

```
pos_type tellg();
```

Returns: if `fail() == true`, returns `streampos(-1)` to indicate failure. Otherwise, returns `rdbuf()->pubseekoff(0, cur, in)`.

```
basic_istream<charT,traits>& seekg(pos_type& pos);
```

Effects: If `fail() != true`, executes `rdbuf()->pubseekpos(`*pos*`)`.
Returns: `*this`.

```
basic_istream<charT,traits>& seekg(off_type& off, ios_base::seekdir dir);
```

Effects: If `fail() != true`, executes `rdbuf()->pubseekoff(`*off*`,` *dir*`)`.
Returns: `*this`.

27.6.1.4 Standard `basic_istream` manipulators [lib.istream.manip]

```
namespace std {
  template<class charT, class traits>
    basic_istream<charT,traits>& ws(basic_istream<charT,traits>& is);
}
```

Effects: Skips any whitespace in the input sequence: saves a copy of *is*`.fmtflags`, then clears *is*`.skipws` in *is*`.flags()`. Then calls *is*`.ipfx()`, then *is*`.isfx()`, then restores *is*`.flags()` to its saved value.
Returns: *is*.

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:
  // Types:
    typedef charT                      char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;

  // 27.6.2.2 Constructor/destructor:
    explicit basic_ostream(basic_streambuf<char_type,traits>* sb);
    virtual ~basic_ostream();
  // 27.6.2.3 Prefix/suffix:
    bool opfx();
    void osfx();

  // 27.6.2.4 Formatted output:
    basic_ostream<charT,traits>& operator<<
        (basic_ostream<charT,traits>& (*pf)(basic_ostream<charT,traits>&));
    basic_ostream<charT,traits>& operator<<
        (basic_ios<charT,traits>& (*pf)(basic_ios<charT,traits>&));
    basic_ostream<charT,traits>& operator<<(const char_type* s);

    basic_ostream<charT,traits>& operator<<(char_type c);
    basic_ostream<charT,traits>& operator<<(bool n);
    basic_ostream<charT,traits>& operator<<(short n);
    basic_ostream<charT,traits>& operator<<(unsigned short n);
    basic_ostream<charT,traits>& operator<<(int n);
    basic_ostream<charT,traits>& operator<<(unsigned int n);
    basic_ostream<charT,traits>& operator<<(long n);
    basic_ostream<charT,traits>& operator<<(unsigned long n);
    basic_ostream<charT,traits>& operator<<(float f);
    basic_ostream<charT,traits>& operator<<(double f);
    basic_ostream<charT,traits>& operator<<(long double f);

    basic_ostream<charT,traits>& operator<<(void* p);
    basic_ostream<charT,traits>& operator<<
        (basic_streambuf<char_type,traits>* sb);

  // 27.6.2.5 Unformatted output:
    basic_ostream<charT,traits>& put(char_type c);
    basic_ostream<charT,traits>& write(const char_type* s, streamsize n);

    basic_ostream<charT,traits>& flush();

    pos_type tellp();
    basic_ostream<charT,traits>& seekp(pos_type&);
    basic_ostream<charT,traits>& seekp(off_type&, ios_base::seekdir);
  };
}
```

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 virtual members of `rdbuf()` except `overflow()`. If the called function throws an exception, the output function calls `setstate(badbit)`, which may throw `ios_base::failure` (27.4.4.3), and if `badbit` is on in `exceptions()` rethrows the exception.

27.6.2.2 `basic_ostream` constructors [lib.ostream.cons]

```
explicit 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*`)` (27.4.4.1).
Postcondition: `rdbuf() ==` *sb*.

```
virtual ~basic_ostream();
```

Effects: Destroys an object of class `basic_ostream`.
Notes: Does not perform any operations on `rdbuf()`.

27.6.2.3 `basic_ostream` prefix and suffix functions [lib.ostream.prefix]

```
bool opfx();
```

1 If `good()` is nonzero, prepares for formatted or unformatted output. If `tie()` is not a null pointer, calls `tie()->flush()`.[237)]
Returns: `good()`.[238)]

```
void osfx();
```

2 If `(flags() & unitbuf) != 0`, calls `flush()`.

```
pos_type tellp();
```

Returns: if `fail() == true`, returns `streampos(-1)` to indicate failure. Otherwise, returns `rdbuf()->pubseekoff(0, cur, out)`.

```
basic_ostream<charT,traits>& seekp(pos_type& pos);
```

Effects: If `fail() != true`, executes `rdbuf()->pubseekpos(`*`pos`*`)`.
Returns: `*this`.

```
basic_ostream<charT,traits>& seekp(off_type& off, ios_base::seekdir dir);
```

If `fail() != true`, executes `rdbuf()->pubseekoff(`*`off`*`, `*`dir`*`)`.
Returns:

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 (22.2.2):

```
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);
```

[237)] The call `tie()->flush()` does not necessarily occur if the function can determine that no synchronization is necessary.
[238)] The function signatures `opfx()` and `osfx()` can also perform additional implementation-dependent operations.

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`.

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. The behavior of such functions is described in terms of the conversion specification ‘‘as if’’ for an equivalent call to the function `::fprintf()`,[239] 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.[240]

— 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)`, which may throw `ios_base::failure` (27.4.4.3).

6 [*Note:* For conversion from an integral type other than a character type, the function determines the integral conversion specifier as indicated in Table 74:

Table 74—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`

—*end note*]

7 [*Note:* For conversion from a floating-point type, the function determines the floating-point conversion specifier as indicated in Table 75:

239) The signature `fprintf(FILE*, const char_type*, ...)` is declared in `<cstdio>` (27.8.2).
240) The stream buffer can, of course, be associated with an output file, but it need not be.

Table 75—Floating-point conversions

State	`stdio` equivalent
`(flags() & floatfield) == fixed`	`%f`
`(flags() & floatfield) == scientific`	`%e`
`(flags() & uppercase) != 0`	`%E`
`Otherwise,`	
	`%g`
`(flags() & uppercase) != 0`	`%G`

—*end note*]

8 [*Note:* The conversion specifier has the following additional qualifiers prepended as indicated in Table 76:

Table 76—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`	`#`

—*end note*]

9 [*Note:* For any conversion, if `width()` is nonzero, then a field width is specified in the conversion specification. The value is `width()`. —*end note*]

10 For conversion from a floating-point type, if `(flags() & fixed) != 0` or if `precision() > 0`, then `precision()` is specified in the conversion specification.

11 [*Note:* Moreover, for any conversion, padding with the fill character returned by `fill()` behaves as indicated in Table 77:

Table 77—Fill padding

State	Justification	`fprintf` flag,padding
`(flags() & adjustfield)==left`	left (pad after text)	(none), space padding
`(flags() & adjustfield)==internal`	internal	0, zero padding[241)]
Otherwise	right (pad before text)	-, space padding

—*end note*]

12 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]

```
basic_ostream<charT,traits>& operator<<
    (basic_ostream<charT,traits>& (*pf)(basic_ostream<charT,traits>&))
```

241) The conversion specification `#o` generates a leading `0` which is *not* a padding character.

Returns: `pf(*this)`.[242)]

```
basic_ostream<charT,traits>& operator<<
    (basic_ios<charT,traits>& (*pf)(basic_ios<charT,traits>&))
```

Effects: Calls `pf(*this)`.
Returns: `*this`.[243)]

```
basic_ostream<charT,traits>& operator<<(const char_type* s);
```

Requires: *s* shall be a null-terminated byte string.
Effects: Converts the NTBS *s* with the conversion specifier `s`.
Returns: `*this`.

```
basic_ostream<charT,traits>& operator<<(char_type c);
```

Effects: Converts the `char_type` *c* with the conversion specifier `c` and a field width of zero.[244)]
Notes: The stored field width (`basic_ios<charT,traits>::width()`) is *not* set to zero.
Returns: `*this`.

```
basic_ostream<charT,traits>& operator<<(bool n);
```

1 Behaves as if:

```
    getloc().use<num_put<charT,istreambuf_iterator<charT,traits> >()
      .put(*this, *this, getloc(), n);
```

which writes out a 0 or 1, or the results of `getloc().use<numpunct<charT> >().truename()` or `falsename()` (22.2.2), according as whether the `boolalpha` flag is set.
Returns: `*this`.

```
basic_ostream<charT,traits>& operator<<(short n);
```

Effects: Converts the signed short integer *n* with the integral conversion specifier preceded by `h`.
Returns: `*this`.

```
basic_ostream<charT,traits>& operator<<(unsigned short n);
```

Effects: Converts the unsigned short integer *n* with the integral conversion specifier preceded by `h`.
Returns: `*this`.

```
basic_ostream<charT,traits>& operator<<(int n);
```

Effects: Converts the signed integer *n* with the integral conversion specifier.
Returns: `*this`.

```
basic_ostream<charT,traits>& operator<<(unsigned int n);
```

242) See, for example, the function signature `endl(basic_ostream&)` (27.6.2.6) .
243) See, for example, the function signature `dec(ios_base&)` (27.4.5.3).
244) Note that this function is not overloaded on types `signed char` and `unsigned char`.

Effects: Converts the unsigned integer *n* with the integral conversion specifier.
Returns: `*this`.

```
basic_ostream<charT,traits>& operator<<(long n);
```

Effects: Converts the signed long integer *n* with the integral conversion specifier preceded by `l`.
Returns: `*this`.

```
basic_ostream<charT,traits>& operator<<(unsigned long n);
```

Effects: Converts the unsigned long integer *n* with the integral conversion specifier preceded by `l`.
Returns: `*this`.

```
basic_ostream<charT,traits>& operator<<(float f);
```

Effects: Converts the `float` *f* with the floating-point conversion specifier.
Returns: `*this`.

```
basic_ostream<charT,traits>& operator<<(double f);
```

Effects: Converts the `double` *f* with the floating-point conversion specifier.
Returns: `*this`.

```
basic_ostream<charT,traits>& operator<<(long double f);
```

Effects: Converts the `long double` *f* with the floating-point conversion specifier preceded by `L`.
Returns: `*this`.

```
basic_ostream<charT,traits>& operator<<(void* p);
```

Effects: Converts the pointer to `void` *p* with the conversion specifier `p`.
Returns: `*this`.

```
basic_ostream<charT,traits>& operator<<
    (basic_streambuf<charT,traits>* sb);
```

Effects: Gets characters from *sb* and inserts them in `*this`. Characters are read from *sb* 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 while getting a character from *sb* (in which case, the exception is rethrown).

2 If the function inserts no characters or if it stopped because an exception was thrown while extracting a character, it calls `setstate(failbit)` (which may throw `ios_base::failure` (27.4.4.3)). 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.

```
basic_ostream<charT,traits>& put(char_type c);
```

Effects: Inserts the character *c*, if possible.[245)]

2 Otherwise, calls `setstate(badbit)` (which may throw `ios_base::failure` (27.4.4.3)).
Returns: `*this`.

```
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*.[246)] 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)`, which may throw `ios_base::failure` (27.4.4.3)).
Returns: `*this`.

```
basic_ostream& flush();
```

3 If `rdbuf()` is not a null pointer, calls `rdbuf()->pubsync()`. If that function returns `traits::eof()`, calls `setstate(badbit)` (which may throw `ios_base::failure` (27.4.4.3)).
Returns: `*this`.

27.6.2.6 Standard `basic_ostream` manipulators [lib.ostream.manip]

```
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*.[247)]

```
namespce std {
  template<class charT, class traits>
    basic_ostream<charT,traits>& ends(basic_ostream<charT,traits>& os);
}
```

Effects: Inserts a null character into the output sequence: calls *os*`.put(traits::eos())`.
Returns: *os*.

245) Note that this function is not overloaded on types `signed char` and `unsigned char`.
246) Note that this function is not overloaded on types `signed char` and `unsigned char`.
247) 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.

```
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]

1 The header `<iomanip>` defines a type and several related functions that use this type to provide extractors and inserters that alter information maintained by class `ios_base` and its derived classes.

2 The type ***smanip*** is an implementation-defined function type (8.3.5) returned by the standard manipulators.

smanip `resetiosflags(ios_base::fmtflags` *`mask`*`);`

Returns: ***smanip***(*`f`*, *`mask`*), where *`f`* can be defined as:[248)]

```
  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;
    }
```

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;
    }
```

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;
    }
```

smanip `setfill(int` *`c`*`);`

248) 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`).

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;
}
```

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;
}
```

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]

1 The header `<sstream>` defines three template classes, and six types, that associate stream buffers with objects of class `basic_string`, as described in subclause 21.1.

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 77—Header `<cstdlib>` synopsis

Type	Name(s)	
Functions:		
	`atoi`	`strtod`
	`atol`	`strtol`

2 *SEE ALSO:* ISO C subclause 7.10.1.

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:
  // Types:
    typedef charT                     char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;

  // 27.7.1.1 Constructors:
    explicit basic_stringbuf(ios_base::openmode which
                              = ios_base::in | ios_base::out);
    explicit basic_stringbuf(const basic_string<char_type>& str,
                             ios_base::openmode which
                              = ios_base::in | ios_base::out);

  // 27.7.1.2 Get and set:
    basic_string<char_type> str() const;
    void                    str(const basic_string<char_type>& s);

  protected:
  // 27.7.1.3 Overridden virtual functions:
    virtual int_type   underflow();
    virtual int_type   pbackfail(int_type c = traits::eof());
    virtual int_type   overflow (int_type c = traits::eof());

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

  private:
//  ios_base::openmode mode;        exposition only
  };
}
```

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`.

27.7.1.1 `basic_stringbuf` constructors [lib.stringbuf.cons]

```
explicit 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()` (27.5.2.1), and initializing ***mode*** with *`which`*.

Notes: The function allocates no array object.

```
explicit 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()`, initializing the base class with `basic_streambuf()` (27.5.2.1), and initializing ***mode*** with *`which`*.

Postconditions: `str() ==` *`str`*. If *`str`*`.size() > 0`, sets the get and/or put pointers as indicated in Table 78:

Table 78—`str` get/set areas

Condition	Setting
`(`*`which`*` & ios_base::in) != 0`	`setg(str(),str(),str()+`*`str`*`.size())`
`(`*`which`*` & ios_base::out) != 0`	`setp(str(),str(),str()+`*`str`*`.size())`

27.7.1.2 Member functions [lib.stringbuf.members]

```
basic_string<char_type> str() const;
```

Returns: The return value of this function are indicated in Table 79:

Table 79—`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);
```

Postcondition: `str() ==` *`s`*. If *`str`*`.size() > 0`, sets the get and/or put pointers as indicated in Table 80:

Table 80—`str` get/set areas

Condition	Setting
`(`*`which`*` & ios_base::in) != 0`	`setg(str(),str(),str()+`*`str`*`.size())`
`(`*`which`*` & ios_base::out) != 0`	`setp(str(),str(),str()+`*`str`*`.size())`

27.7.1.3 Overridden virtual functions [lib.stringbuf.virtuals]

```
int_type underflow();
```

Returns: If the input sequence has a read position available, returns `char_type(*gptr())`. Otherwise, returns `traits::eof()`.

```
int_type pbackfail(int_type c = traits::eof());
```

Effects: Puts back the character designated by *`c`* to the input sequence, if possible, in one of three ways:

— If *`c`* `!= traits::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`* `!= traits::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`* `== traits::eof()` and if the input sequence has a putback position available, assigns `gptr() - 1` to `gptr()`.
Returns: `char_type(`*`c`*`)`.

Returns: `traits::eof()` to indicate failure.
Notes: If the function can succeed in more than one of these ways, it is unspecified which way is chosen.

```
int_type overflow(int_type c = traits::eof());
```

Effects: Appends the character designated by *`c`* to the output sequence, if possible, in one of two ways:

— If *`c`* `!= traits::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`* `== traits::eof()`, there is no character to append.
Signals success by returning a value other than `traits::eof()`.

Notes: The function can alter the number of write positions available as a result of any call.
Returns: `traits::eof()` to indicate failure.

1 [*Note:* 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()`). —*end note*]

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

Effects: Alters the stream position within one of the controlled sequences, if possible, as indicated in Table 81:

Table 81—`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.

2 For a sequence to be positioned, if its next pointer (either `gptr()` or `pptr()`) is a null pointer, the positioning operation fails. Otherwise, the function determines *`newoff`* as indicated in Table 82:

Table 82—`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

3 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.

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

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.

4 For a sequence to be positionedif its next pointer (either `gptr()` or `pptr()`) 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.

27.7.2 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:
  // Types:
    typedef charT                      char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;

  // 27.7.2.1 Constructors:
    explicit basic_istringstream(ios_base::openmode which = ios_base::in);
    explicit basic_istringstream(const basic_string<charT>& str,
                                 ios_base::openmode which = ios_base::in);

  // 27.7.2.2 Members:
    basic_stringbuf<charT,traits>* rdbuf() const;

    basic_string<charT> str() const;
    void                str(const basic_string<charT>& s);
  private:
//  basic_stringbuf<charT,traits> sb;     exposition only
  };
}
```

1 The class `basic_istringstream<charT,traits>` supports reading objects of class `basic_string<charT,traits>`. It uses a `basic_stringbuf` object to control the associated storage. For the sake of exposition, the maintained data is presented here as:

— `sb`, the `stringbuf` object.

27.7.2.1 `basic_istringstream` constructors [lib.istringstream.cons]

```
explicit 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(&sb)` and initializing `sb` with `basic_stringbuf<charT,traits>(which))` (27.7.1.1).

```
explicit 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(&sb)` and initializing `sb` with `basic_stringbuf<charT,traits>(str, which))` (27.7.1.1).

27.7.2.2 Member functions [lib.istringstream.members]

```
basic_stringbuf<charT,traits>* rdbuf() const;
```

Returns: `(basic_stringbuf<charT,traits>*)&`*`sb`*.

```
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]

```
namespace std {
  template <class charT, class traits = ios_traits<charT> >
  class basic_ostringstream : public basic_ostream<charT,traits> {
  public:
  // Types:
    typedef charT            char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;

  // 27.7.2.4 Constructors/destructor:
    explicit basic_ostringstream(ios_base::openmode which = ios_base::out);
    explicit basic_ostringstream(const basic_string<charT>& str,
                             ios_base::openmode which = ios_base::out);
    virtual ~basic_ostringstream();

  // 27.7.2.5 Members:
    basic_stringbuf<charT,traits>* rdbuf() const;

    basic_string<charT> str() const;
    void                str(const basic_string<charT>& s);
  private:
//  basic_stringbuf<charT,traits> sb;     exposition only
  };
}
```

1 The class `basic_ostringstream<charT,traits>` supports writing objects of class `basic_string<charT,traits>`. It uses a `basic_stringbuf` object to control the associated storage. For the sake of exposition, the maintained data is presented here as:

— *`sb`*, the `stringbuf` object.

27.7.2.4 `basic_ostringstream` constructors [lib.ostringstream.cons]

```
explicit 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`*`)` and initializing *`sb`* with `basic_stringbuf<charT,traits>(`*`which`*`)` (27.7.1.1).

```
explicit 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(&sb)` and initializing *sb* with `basic_stringbuf<charT,traits>(str, which))` (27.7.1.1).

27.7.2.5 Member functions [lib.ostringstream.members]

```
basic_stringbuf<charT,traits>* rdbuf() const;
```

Returns: `(basic_stringbuf<charT,traits>*)&sb`.

```
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]

27.8.1 File streams [lib.fstreams]

1 The header `<fstream>` defines three class templates, and six types, that associate stream buffers with files and assist reading and writing files.

Header `<fstream>` synopsis

```
#include <streambuf>
#include <istream>
#include <ostream>

namespace std {
  template <class charT, class traits = ios_traits<charT> >
    class basic_filebuf;
  typedef basic_filebuf<char>    filebuf;
  typedef basic_filebuf<wchar_t> wfilebuf;

  template <class charT, class traits = ios_traits<charT> >
    class basic_ifstream;
  typedef basic_ifstream<char>    ifstream;
  typedef basic_ifstream<wchar_t> wifstream;

  template <class charT, class traits = ios_traits<charT> >
    class basic_ofstream;
  typedef basic_ofstream<char>    ofstream;
  typedef basic_ofstream<wchar_t> wofstream;
}
```

2 In this subclause, the type name *`FILE`* is a synonym for the type `FILE`.[249)]

— **File** A File provides an external source/sink stream whose *underlaid character type* is `char` (byte).[250)]

249) `FILE` is defined in `<cstdio>` (27.8.2).
250) A File is a sequence of multibyte characters. In order to provide the contents as a wide character sequence, `filebuf` should convert between wide character sequences and multibyte character sequences.

— **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.

27.8.1.1 Template class `basic_filebuf` [lib.filebuf]

```
namespace std {
  template <class charT, class traits = ios_traits<charT> >
  class basic_filebuf : public basic_streambuf<charT,traits> {
  public:
  // Types:
    typedef charT                    char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;

  // 27.8.1.2 Constructors/destructor:
    basic_filebuf();
    virtual ~basic_filebuf();

  // 27.8.1.3 Members:
    bool is_open() const;
    basic_filebuf<charT,traits>* open(const char* s, ios_base::openmode mode);
    basic_filebuf<charT,traits>* close();

  protected:
  // 27.8.1.4 Overridden virtual functions:
    virtual int      showmanyc();
    virtual int_type underflow();
    virtual int_type pbackfail(int_type c = traits::eof());
    virtual int_type overflow (int_type c = traits::eof());

    virtual basic_streambuf<charT,traits>*
                     setbuf(char_type* s, streamsize n);
    virtual pos_type seekoff(off_type off, ios_base::seekdir way,
                             ios_base::openmode which
                               = ios_base::in | ios_base::out);
    virtual pos_type seekpos(pos_type sp, ios_base::openmode which
                               = ios_base::in | ios_base::out);
    virtual int      sync();
    virtual void     imbue(const locale& loc);
  };
}
```

1 The class `basic_filebuf<charT,traits>` associates both the input sequence and the output sequence with a file.

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 with the Standard C library `FILE`s.

3 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.

4 In order to support file I/O and multibyte/wide character conversion, conversions are performed using `getloc()`. Specifically:

— when input is performed, bytes are read from the file and converted to `charT` ‘‘as if’’ by using `getloc().use<codecvt<char,charT,ios_traits::state_type> >()`

— when output is performed, `charT`’s are converted to `char` ‘‘as if’’ by using `getloc().use<codecvt<charT,char,ios_traits::state_type> >()`.

27.8.1.2 `basic_filebuf` constructors [lib.filebuf.cons]

```
basic_filebuf();
```

Effects: Constructs an object of class `basic_filebuf<charT,traits>`, initializing the base class with `basic_streambuf<charT,traits>()` (27.5.2.1).
Postcondition: `is_open() == false`.

```
virtual ~basic_filebuf();
```

Effects: Destroys an object of class `basic_filebuf<charT,traits>`. Calls `close()`.

27.8.1.3 Member functions [lib.filebuf.members]

```
bool is_open() const;
```

Returns: `true` if the associated file is available and open.

```
basic_filebuf<charT,traits>* open(const char* s, ios_base::openmode mode);
```

Effects: If `is_open() == false`, returns a null pointer. Otherwise, calls `basic_streambuf<charT,traits>::basic_streambuf()` (27.5.2.1).
It then opens a file, if possible, whose name is the NTBS *s* (‘‘as if’’ by calling `::fopen(`*s*`,`*modstr*`)`).
[*Note:* The NTBS *modstr* is determined from *mode* `& ~ios_base::ate` as indicated in Table 83:

Table 83—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"

—*end note*]

1 If the open operation succeeds and `(`*`mode`*` & ios_base::ate) != 0`, positions the file to the end (‘‘as if’’ by calling `::fseek(`*`file`*`,0,SEEK_END)`).[251)]

2 If the repositioning operation fails, calls `close()` and returns a null pointer to indicate failure.
Returns: `this` if successful, a null pointer otherwise.

```
basic_filebuf<charT,traits>* close();
```

Effects: If `is_open() == false`, returns a null pointer. Otherwise, closes the file (‘‘as if’’ by calling `::fclose(`*`file`*`)`).[252)]
Returns: `this` on success, a null pointer otherwise.
Postcondition: `is_open() == false`.

27.8.1.4 Overridden virtual functions [lib.filebuf.virtuals]

```
int showmanyc();
```

Requires: `is_open() == true`.
Effects: Behaves the same as `basic_streambuf::showmanyc()` (27.5.2.4).
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.

```
int_type underflow();
```

Requires: `is_open() == true`.
Effects: Behaves according to the description of `basic_streambuf<charT,traits>::underflow()`, with the specialization that a sequence of characters is read from the input sequence ‘‘as if’’ by reading from the associated file into an internal buffer (`from_buf`) and then ‘‘as if’’ doing

```
char   from_buf[FSIZE];
char*  from_end;
charT  to_buf[TSIZE];
charT* to_end;
codecvt_base::result r
    = getloc().use<codecvt<char,charT,typename ios_traits::state_type> >().
        convert(st,from_buf,from_buf+FSIZE,from_end,
                to_buf, to_buf+to_size, to_end);
```

This must be done in such a way that the class can recover the position (`fpos_t`) corresponding to each character between `to_buf` and `to_end.` If the value of `r` indicates that `convert()` ran out of space in `to_buf`, retry with a larger `to_buf`.

```
int_type pbackfail(int_type c = traits::eof());
```

Requires: `is_open() == true`.
Effects: Puts back the character designated by *`c`* to the input sequence, if possible, in one of four ways:

— If *`c`* `!= traits::eof()` and if the function makes a putback position available and if `char_type(`*`c`*`) == char_type(gptr()[-1])`, decrements the next pointer for the input sequence, `gptr()`.

— If *`c`* `!= traits::eof()` and if the function makes a putback position available, and if the function is permitted to assign to the putback position, decrements the next pointer for the input sequence, and stores *`c`* there.

251) 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>` (27.8.2).
252) The function signature `fclose(FILE*)` is declared, in `<cstdio>` (27.8.2).

— If *c* `== traits::eof()` and if either the input sequence has a putback position available or the function makes a putback position available, decrements the next pointer for the input sequence, `gptr()`.

Returns: `traits::eof()` to indicate failure, otherwise *c*.

Notes: If `is_open() == false`, 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()`.

```
int_type overflow(int_type c = traits::eof());
```

Requires: `is_open() == true`.

Effects: Behaves according to the description of `basic_streambuf<charT,traits>::overflow(`*c*`)`, except that the behavior of ''consuming characters'' is performed by first coverting ''as if'' by:

```
charT* b = pbase();
charT* p = pptr();
charT* end;
char   buf[BSIZE];
char*  ebuf;
codecvt_base::result r
    = getloc().use<codecvt<charT, char, ios_traits::state_type> >().
        convert(st,b(),p(),end,buf,buf+BSIZE,ebuf);
```

and then

— If `r == codecvt_base::error` then fail.

— If `r == codecvt_base::noconv` then output chrarcters from `b` upto (and not including) `p`.

— If `r == codecvt_base::partial` then output to the file characters from `buf` upto `ebuf`, and repeat using characters from `end` to `p`. If output fails, fail (without repeating).

— Otherwise output from `buf` to `ebuf`, and fail if output fails. At this point if `b != p` and `b == end` (`buf` isn't large enough) then increase `BSIZE` and repeat from the beginning.

Returns: `traits::eof()` to indicate failure. If `is_open() == false`, the function always fails.

```
basic_streambuf* setbuf(char_type* s, int n);
```

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

Requires: `is_open() == true`.

Effects: The current state is determined as follows: If the the last operation was `overflow()`, the current state is obtained by combining the shiftstate contained in `st` with the current position (`fpos_t`) of the file. If the last operation was `underflow()`, the shiftstate and file position are determined (according to whatever means they were saved by `underflow()`) as corresponding to `pptr()`.
Then, alters the stream position within the controlled sequences, if possible, as described below.
If `is_open() == false`, the positioning operation fails. Otherwise, repositions within the associated file (''as if'' by calling `::fseek(`*file*`,`*off*`,`*whence*`)`.[253)]
[*Note:* The function determines one of three values for the argument *whence*, of type `int`, as indicated

[253)] The macros `SEEK_SET`, `SEEK_CUR`, and `SEEK_END` are defined, and the function signature `fseek(FILE*, long, int)` is declared, in `<cstdio>` (27.8.2).

in Table 84:

Table 84—`seekoff` effects

way **Value**	**`stdio` Equivalent**
`basic_ios::beg`	`SEEK_SET`
`basic_ios::cur`	`SEEK_CUR`
`basic_ios::end`	`SEEK_END`

—*end note*]

The function extracts the conversion state from *`off`* by means of ***get_offstate()*** to reset the `rdstate()` member.

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, returns an invalid stream position (27.1.2.4).

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

Requires: `is_open() == true`.

```
int sync();
```

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

Effects: Calls `sync()` and if `sync()` fails, sets a flag and the next call to any virtual will fail.

27.8.1.5 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:
  // Types:
    typedef charT                     char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;

  // 27.8.1.6 Constructors:
    basic_ifstream();
    explicit basic_ifstream(const char* s, openmode mode = in);

  // 27.8.1.7 Members:
    basic_filebuf<charT,traits>* rdbuf() const;

    bool is_open();
    void open(const char* s, openmode mode = in);
    void close();
  private:
//  basic_filebuf<charT,traits> sb;       exposition only
  };
}
```

1 The class `basic_ifstream<charT,traits>` supports reading from named files. It uses a `basic_filebuf<charT,traits>` object to control the associated sequence. For the sake of exposition, the maintained data is presented here as:

— *sb*, the `filebuf` object.

27.8.1.6 basic_ifstream constructors [lib.ifstream.cons]

```
basic_ifstream();
```

Effects: Constructs an object of class `basic_ifstream<charT,traits>`, initializing the base class with `basic_istream(&sb)` and initializing *sb* with `basic_filebuf<charT,traits>())` (_lib.istream.cons_, 27.8.1.2).

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

Effects: Constructs an object of class `basic_ifstream`, initializing the base class with `basic_istream(&sb)` and initializing *sb* with `basic_filebuf<charT,traits>())` (_lib.istream.cons_, 27.8.1.2), then calls `rdbuf()->open(s,mode)`.

27.8.1.7 Member functions [lib.ifstream.members]

```
explicit basic_filebuf<charT,traits>* rdbuf() const;
```

Returns: `(basic_filebuf<charT,traits>*)&sb`.

```
bool is_open();
```

Returns: `rdbuf()->is_open()`.

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

Effects: Calls `rdbuf()->open(s,mode)`. If `is_open()` returns `false`, calls `setstate(failbit)` (which may throw `ios_base::failure` (27.4.4.3)).

```
void close();
```

Effects: Calls `rdbuf()->close()` and, if that function returns `false`, calls `setstate(failbit)` (which may throw `ios_base::failure` (27.4.4.3)).

27.8.1.8 Template class basic_ofstream [lib.ofstream]

```
namespace std {
  template <class charT, class traits = file_traits<charT> >
  class basic_ofstream : public basic_ostream<charT,traits> {
  public:
  // Types:
    typedef charT                     char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;

  // 27.8.1.9 Constructors:
    basic_ofstream();
    explicit basic_ofstream(const char* s, openmode mode = out);
  // 27.8.1.10 Members:
    basic_filebuf<charT,traits>* rdbuf() const;

    bool is_open();
    void open(const char* s, ios_base::openmode mode = out | trunc);
    void close();
  private:
//  basic_filebuf<charT,traits> sb;        exposition only
  };
}
```

1 The class `basic_ofstream<charT,traits>` supports writing to named files. It uses a `basic_filebuf<charT,traits>` object to control the associated sequence. For the sake of exposition, the maintained data is presented here as:

— *`sb`*, the `filebuf` object.

27.8.1.9 `basic_ofstream` constructors [lib.ofstream.cons]

```
basic_ofstream();
```

Effects: Constructs an object of class `basic_ofstream<charT,traits>`, initializing the base class with `basic_ostream(&`*`sb`*`)` and initializing *`sb`* with `basic_filebuf<charT,traits>())` (27.6.2.2, 27.8.1.2).

```
explicit 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(&`*`sb`*`)` and initializing *`sb`* with `basic_filebuf<charT,traits>())` (27.6.2.2, 27.8.1.2), then calls `rdbuf()->open(`*`s`*`, `*`mode`*`)`.

27.8.1.10 Member functions [lib.ofstream.members]

```
basic_filebuf<charT,traits>* rdbuf() const;
```

Returns: `(basic_filebuf<charT,traits>*)&`*`sb`*.

```
bool is_open();
```

Returns: `rdbuf()->is_open()`.

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

Effects: Calls `rdbuf()->open(`*`s`*`,`*`mode`*`)`. If `is_open()` is then `false`, calls `setstate(failbit)` (which may throw `ios_base::failure` (27.4.4.3)).

```
void close();
```

Effects: Calls `rdbuf()->close()` and, if that function fails (returns a null pointer), calls `setstate(failbit)` (which may throw `ios_base::failure` (27.4.4.3)).

27.8.2 C Library files [lib.c.files]

1 Headers `<cstdio>`, and `<cwchar>`.

Table 84—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 84—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, 10.2) 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-literal
  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-suffix*$_{opt}$
- *octal-literal integer-suffix*$_{opt}$
- *hexadecimal-literal integer-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-literal:
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*
    template-id

qualified-id:
  nested-name-specifier `template`$_{opt}$ *unqualified-id*

postfix-expression:
  primary-expression
  postfix-expression `[` *expression* `]`
  postfix-expression `(` *expression-list*$_{opt}$ `)`
  simple-type-specifier `(` *expression-list*$_{opt}$ `)`
  postfix-expression `.` `template`$_{opt}$ *id-expression*
  postfix-expression `->` `template`$_{opt}$ *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
  simple-declaration

jump-statement:
  `break ;`
  `continue ;`
  `return` *expression*$_{opt}$ `;`
  `goto` *identifier* `;`

declaration-statement:
  block-declaration

A.6 Declarations [gram.dcl.dcl]

declaration-seq:
  declaration
  declaration-seq declaration

declaration:
  block-declaration
  function-definition
  template-declaration
  linkage-specification
  namespace-definition

block-declaration:
  simple-declaration
  asm-definition
  namespace-alias-definition
  using-declaration
  using-directive

simple-declaration:
  decl-specifier-seq$_{opt}$ *init-declarator-list*$_{opt}$ `;`

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:
`::`$_{opt}$ *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 `::`$_{opt}$ *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*
  decl-specifier-seq$_{opt}$ *declarator function-try-block*

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:
  mem-initializer-id `(` *expression-list*$_{opt}$ `)`

mem-initializer-id:
    `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name*
    identifier

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` *declaration*

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*

function-try-block:
 `try` *ctor-initializer-opt function-body 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*$_{opt}$

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 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].

— 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].

— 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 processor. 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; 3.9.1 and 4.8. This is also in ISO C.

4 Function names can be overloaded; 13.

5 Operators can be overloaded; 13.5.

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; 13.5.

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 (3.9.1).

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 3.9

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 processor 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 processors 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 7.1.5 (type specifiers)

4 **Change:** Banning implicit int
In C++ a *decl-specifier-seq* must contain a *type-specifier*. In the following example, the left-hand column presents valid C; the right-hand column presents equivalent C++:

```
void f(const parm);            void f(const int parm);
const n = 3;                   const int n = 3;
main()                         int main()
    /* ... */                      /* ... */
```

Rationale: In C++, implicit int creates several opportunities for ambiguity between expressions involving function-like casts and declarations. Explicit declaration is increasingly considered to be proper style. Liaison with WG14 (C) indicated support for (at least) deprecating implicit int in the next revision of C.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Syntactic transformation. Could be automated.
How widely used: Common.

Subclause 7.2

5 **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

6 **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 12: special member functions [diff.special]

Subclause 12.8 (copying class objects)

1 **Change:** Copying volatile objects
The implicitly-declared copy constructor and implicitly-declared copy assignment operator cannot make a copy of a volatile lvalue. For example, the following is valid in ISO C:

```
struct X { int i; };
struct X x1, x2;
volatile struct X x3 = {0};
x1 = x3; // invalid C++
x2 = x3; // also invalid C++
```

Rationale: Several alternatives were debated at length. Changing the parameter to `volatile const X&` would greatly complicate the generation of efficient code for class objects. Discussion of providing two alternative signatures for these implicitly-defined operations raised unanswered concerns about creating ambiguities and complicating the rules that specify the formation of these operators according to the bases and members.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation. If volatile semantics are required for the copy, a user-declared constructor or assignment must be provided. If non-volatile semantics are required, an explicit `const_cast` can be used.
How widely used: Seldom.

C.2.9 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 `--`; 13.5.6.

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 contents of the C++ Standard library included from the Standard C library. It also summarizes the explicit changes in definitions, declarations, or behavior from the ISO/IEC 9899:1990 and ISO/IEC 9899:1990/DAM 1 noted in other subclauses (17.3.1.2, 18.1, 21.2).

2 The C++ Standard library provides 54 standard macros from the C library, as shown in Table 85.

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

Table 85—Standard Macros

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

4 The C++ Standard library provides 45 standard values from the C library, as shown in Table 86:

Table 86—Standard Values

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

5 The C++ Standard library provides 19 standard types from the C library, as shown in Table 87:

Table 87—Standard Types

clock_t	ldiv_t	size_t <cstdio>	wctrans_t
div_t	mbstate_t	size_t <cstring>	wctype_t
FILE	ptrdiff_t<cstddef>	size_t <ctime>	wint_t <cwchar>
fpos_t	sig_atomic_t	time_t	wint_t <cwctype>
jmp_buf	size_t <cstddef>	va_list	

6 The C++ Standard library provides 2 standard structures from the C library, as shown in Table 88:

Table 88—Standard Structs

lconv	tm <ctime>

7 The C++ Standard library provides 208 standard functions from the C library, as shown in Table 89:

Table 89—Standard Functions

abort	fgetpos	gmtime	log10	rewind	strtok	wcscspn
abs	fgets	isalnum	longjmp	scanf	strtol	wcsftime
acos	fgetwc	isalpha	malloc	setbuf	strxfrm	wcslen
asctime	fgetws	iscntrl	mblen	setlocale	swprintf	wcsncat
asin	floor	isdigit	mbrlen	setvbuf	swscanf	wcsncmp
atan	fmod	isgraph	mbrtowc	signal	system	wcsncpy
atan2	fopen	islower	mbsinit	sin	tan	wcspbrk
atexit	fprintf	isprint	mbsrtowcs	sinh	tanh	wcsrchr
atof	fputc	ispunct	mbstowcs	sprintf	time	wcsrtombs
atoi	fputs	isspace	mbtowc	sqrt	tmpfile	wcsspn
atol	fputwc	isupper	memchr	srand	tmpnam	wcsstr
bsearch	fputws	iswalnum	memcmp	sscanf	tolower	wcstod
btowc	fread	iswalpha	memcpy	strcat	toupper	wcstok
calloc	free	iswcntrl	memmove	strchr	towctrans	wcstol
ceil	freopen	iswctype	memset	strcmp	towlower	wcstombs
clearerr	frexp	iswdigit	mktime	strcoll	towupper	wcstoul
clock	fscanf	iswgraph	modf	strcpy	ungetc	wcsxfrm
cos	fseek	iswlower	perror	strcspn	ungetwc	wctob
cosh	fsetpos	iswprint	pow	strerror	vfwprintf	wctomb
ctime	ftell	iswpunct	printf	strftime	vprintf	wctrans
difftime	fwide	iswspace	putc	strlen	vprintf	wctype
div	fwprintf	iswupper	puts	strncat	vsprintf	wmemchr
exit	fwrite	iswxdigit	putwc	strncmp	vswprintf	wmemcmp
exp	fwscanf	isxdigit	putwchar	strncpy	vwprintf	wmemcpy
fabs	getc	labs	qsort	stroul	wcrtomb	wmemmove
fclose	getchar	ldexp	raise	strpbrk	wcscat	wmemset
feof	getenv	ldiv	rand	strrchr	wcschr	wprintf
ferror	gets	localeconv	realloc	strspn	wcscmp	wscanf
fflush	getwc	localtime	remove	strstr	wcscoll	
fgetc	getwchar	log	rename	strtod	wcscpy	

C.4.1 Modifications to headers [diff.mods.to.headers]

1 For compatibility with the Standard C library, the C++ Standard library provides the 18 *C headers* (D.1), but their use is deprecated in C++.

C.4.2 Modifications to definitions [diff.mods.to.definitions]

C.4.2.1 Type `wchar_t` [diff.wchar.t]

1 `wchar_t` is a keyword in this International Standard (2.8). It does not appear as a type name defined in any of `<cstddef>`, `<cstdlib>`, or `<cwchar>` (21.2).

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`, and`xor`, Standard (2.8). They do not appear as macro names defined in `<ciso646>`.

C.4.2.3 Macro `NULL` [diff.null]

1 The macro `NULL`, defined in any of `<clocale>`, `<cstddef>`, `<cstdio>`, `<cstdlib>`, `<cstring>`, `<ctime>`, or `<cwchar>`, is an implementation-defined C++ null-pointer constant in this International Standard (18.1).[254)]

C.4.3 Modifications to declarations [diff.mods.to.declarations]

1 Header `<cstring>`: The following functions have different declarations:

— `strchr`

— `strpbrk`

— `strrchr`

— `strstr`

— `memchr`

2 Subclause (21.2) describes the changes.

C.4.4 Modifications to behavior [diff.mods.to.behavior]

1 Header `<cstdlib>`: The following functions have different behavior:

— `atexit`

— `exit`

Subclause (18.3) describes the changes.

2 Header `<csetjmp>`: The following functions have different behavior:

— `longjmp`

Subclause (18.7) describes the changes.

254) Possible definitions include `0` and `0L`, but not `(void*)0`.

C.4.4.1 Macro `offsetof(`*type*, *member-designator*`)` [diff.offsetof]

1 The macro `offsetof`, defined in `<cstddef>`, accepts a restricted set of *type* arguments in this International Standard. Subclause (18.1) describes the change.

C.4.4.2 Memory allocation functions [diff.malloc]

1 The functions `calloc`, `malloc`, and `realloc` are restricted in this International Standard. Subclause (20.4.6) describes the changes.

Annex D (normative) Compatibility features [depr]

1 This Clause describes features of the C++ Standard that are specified for compatibility with existing implementations.

D.1 Standard C library headers [depr.c.headers]

1 For compatibility with the Standard C library, the C++ Standard library provides the 18 *C headers*, as shown in Table 90:

Table 90—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 `cname`, followed by an explicit *using-declaration* (7.3.3) for each name placed in the standard library namespace by the header (17.3.1.2).

3 [*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. —*end example*]

D.2 Old iostreams members [depr.ios.members]

1 The following member names are in addition to names specified in Clause _lib.iostreams_:

```
namespace std {
  class ios_base {
  public:
    typedef T1  io_state;
    typedef T2 open_mode;
    typedef T3  seek_dir;
    // remainder unchanged
  };
}
```

2 The type `io_state` is a synonym for an integer type (indicated here as *T1*) that permits certain member functions to overload others on parameters of type `iostate` and provide the same behavior.

3 The type `open_mode` is a synonym for an integer type (indicated here as *T2*) that permits certain member functions to overload others on parameters of type `openmode` and provide the same behavior.

4 The type `seek_dir` is a synonym for an integer type (indicated here as *`T3`*) that permits certain member functions to overload others on parameters of type `iostate` and provide the same behavior.

5 An implementation may provide the following additional member function, which has the effect of calling `sbumpc()` (27.5.2.2.3):

```
namespace std {
  template<class charT, class traits = ios_traits<charT> >
  class basic_streambuf {
  public:
    void stossc();
    // remainder unchanged
  };
}
```

6 An implementation may provide the following member functions that overload signatures specified in Clause _lib.iostreams_:

```
namespace std {
  template<class charT, class Traits> class basic_ios {
  public:
    void clear(io_state state);
    void setstate(io_state state);
    // remainder unchanged
  };

  class ios_base {
  public:
    void exceptions(io_state);
    // remainder unchanged
  };

  template<class charT, class traits = ios_traits<charT> >
  class basic_streambuf {
  public:
    pos_type pubseekoff(off_type off, ios_base::seek_dir way,
              ios_base::open_mode which = ios_base::in | ios_base::out);
    pos_type pubseekpos(pos_type sp,
              ios_base::open_mode which = ios_base::in | ios_base::out);
    // remainder unchanged
  };

  template <class charT, class traits = ios_traits<charT> >
  class basic_filebuf : public basic_streambuf<charT,traits> {
  public:
    basic_filebuf<charT,traits>* open(const char* s, ios_base::open_mode mode);
    // remainder unchanged
  };

  template <class charT, class traits = file_traits<charT> >
  class basic_ifstream : public basic_istream<charT,traits> {
  public:
    void open(const char* s, open_mode mode = in);
    // remainder unchanged
  };
  template <class charT, class traits = file_traits<charT> >
  class basic_ofstream : public basic_ostream<charT,traits> {
  public:
    void open(const char* s, ios_base::open_mode mode = out | trunc);
    // remainder unchanged
  };

}
```

7 The effects of these functions is to call the corresponding member function specified in Clause _lib.iostreams_.

D.3 `char*` streams [depr.str.strstreams]

1 The header `<strstream>` (and, as per D.1, `<strstream.h>`) defines three types that associate stream buffers with character array objects and assist reading and writing such objects.

D.3.1 Class `strstreambuf` [depr.strstreambuf]

```
namespace std {
  class strstreambuf : public streambuf<char> {
  public:
    explicit strstreambuf(streamsize alsize_arg = 0);
    strstreambuf(void* (*palloc_arg)(size_t), void (*pfree_arg)(void*));
    strstreambuf(char* gnext_arg, streamsize n, char* pbeg_arg = 0);
    strstreambuf(const char* gnext_arg, streamsize n);

    strstreambuf(signed char* gnext_arg, streamsize n,
                 signed char* pbeg_arg = 0);
    strstreambuf(const signed char* gnext_arg, streamsize n);
    strstreambuf(unsigned char* gnext_arg, streamsize n,
                 unsigned char* pbeg_arg = 0);
    strstreambuf(const unsigned char* gnext_arg, streamsize n);

    virtual ~strstreambuf();

    void  freeze(bool = 1);
    char* str();
    int   pcount();

  protected:
    virtual int_type overflow (int_type c = ios_traits<char>::eof());
    virtual int_type pbackfail(int_type c = ios_traits<char>::eof());
    virtual int_type underflow();
    virtual pos_type seekoff(off_type off, ios_base::seekdir way,
                             ios_base::openmode which
                              = ios_base::in | ios_base::out);
    virtual pos_type seekpos(pos_type sp, ios_base::openmode which
                              = ios_base::in | ios_base::out);
    virtual streambuf<char>* setbuf(char* s, streamsize n);
  private:
//  typedef T1 strstate;                        exposition only
//  static const strstate allocated;            exposition only
//  static const strstate constant;             exposition only
//  static const strstate dynamic;              exposition only
//  static const strstate frozen;               exposition only
//  strstate strmode;                           exposition only
//  streamsize alsize;                          exposition only
//  void* (*palloc)(size_t);                    exposition only
//  void (*pfree)(void*);                       exposition only
  };
}
```

1 The class `strstreambuf` associates the input sequence, and possibly the output sequence, with an object of some *character* array type, whose elements store arbitrary values. The array object has several attributes.

2 [*Note:* For the sake of exposition, these are represented as elements of a bitmask type (indicated here as *`T1`*) called *`strstate`*. The elements are:

— *`allocated`*, set when a dynamic array object has been allocated, and hence should be freed by the destructor for the `strstreambuf` object;

— *`constant`*, set when the array object has `const` elements, so the output sequence cannot be written;

— *`dynamic`*, set when the array object is allocated (or reallocated) as necessary to hold a character sequence that can change in length;

— *`frozen`*, set when the program has requested that the array object not be altered, reallocated, or freed. —*end note*]

3 [*Note:* For the sake of exposition, the maintained data is presented here as:

— *`strstate strmode`*, the attributes of the array object associated with the `strstreambuf` object;

— `int` *`alsize`*, the suggested minimum size for a dynamic array object;

— `void* (*`*`palloc`*`)(size_t)`, points to the function to call to allocate a dynamic array object;

— `void (*`*`pfree`*`)(void*)`, points to the function to call to free a dynamic array object. —*end note*]

4 Each object of class `strstreambuf` has a *seekable area*, delimited by the pointers *`seeklow`* and *`seekhigh`*. If *`gnext`* is a null pointer, the seekable area is undefined. Otherwise, *`seeklow`* equals *`gbeg`* and *`seekhigh`* is either *`pend`*, if *`pend`* is not a null pointer, or *`gend`*.

D.3.1.1 `strstreambuf` constructors [depr.strstreambuf.cons]

`explicit strstreambuf(streamsize` *`alsize_arg`* `= 0);`

Effects: Constructs an object of class `strstreambuf`, initializing the base class with `streambuf()`. The postconditions of this function are indicated in Table 91:

Table 91—`strstreambuf(streamsize)` effects

Element	Value
`strmode`	*`dynamic`*
`alsize`	*`alsize_arg`*
`palloc`	a null pointer
`pfree`	a null pointer

```
strstreambuf(void* (*palloc_arg)(size_t), void (*pfree_arg)(void*));
```

Effects: Constructs an object of class `strstreambuf`, initializing the base class with `streambuf()`. The postconditions of this function are indicated in Table 92:

Table 92—`strstreambuf(void* (*)(size_t),void (*)(void*)` effects

Element	Value
strmode	*dynamic*
alsize	an unspecified value
palloc	*palloc_arg*
pfree	*pfree_arg*

```
strstreambuf(char* gnext_arg, streamsize n, char *pbeg_arg = 0);
strstreambuf(signed char* gnext_arg, streamsize n,
                                           signed char *pbeg_arg = 0);
strstreambuf(unsigned char* gnext_arg, streamsize n,
                                           unsigned char *pbeg_arg = 0);
```

Effects: Constructs an object of class `strstreambuf`, initializing the base class with `streambuf()`. The postconditions of this function are indicated in Table 93:

Table 93—`strstreambuf(charT*,streamsize,charT*)` effects

Element	Value
strmode	0
alsize	an unspecified value
palloc	a null pointer
pfree	a null pointer

1 *gnext_arg* shall point to the first element of an array object whose number of elements *N* is determined as follows:

— If `n > 0`, *N* is *n*.

— If `n == 0`, *N* is `::strlen(gnext_arg)`.

— If `n < 0`, *N* is `INT_MAX`.[255)]

2 If `pbeg_arg` is a null pointer, the function executes:

```
setg(gnext_arg, gnext_arg, gnext_arg + N);
```

3 Otherwise, the function executes:

```
setg(gnext_arg, gnext_arg, pbeg_arg);
setp(pbeg_arg,  pbeg_arg + N);
```

255) The function signature `strlen(const char*)` is declared in `<cstring>`. (21.2). The macro `INT_MAX` is defined in `<climits>` (18.2).

```
strstreambuf(const char* gnext_arg, streamsize n);
strstreambuf(const signed char* gnext_arg, streamsize n);
strstreambuf(const unsigned char* gnext_arg, streamsize n);
```

Effects: Behaves the same as `strstreambuf((char*)`*`gnext_arg`*`,`*`n`*`)`, except that the constructor also sets *`constant`* in *`strmode`*.

```
virtual ~strstreambuf();
```

Effects: Destroys an object of class `strstreambuf`. The function frees the dynamically allocated array object only if *`strmode`* `&` *`allocated`* `!= 0` and *`strmode`* `&` *`frozen`* `== 0`. (Subclause _lib.strstreambuf.virtuals_ describes how a dynamically allocated array object is freed.)

D.3.1.2 Member functions [depr.strstreambuf.members]

```
void freeze(bool freezefl = 1);
```

Effects: If *`strmode`* & *`dynamic`* is non-zero, alters the freeze status of the dynamic array object as follows:

— If *`freezefl`* is `false`, the function sets *`frozen`* in *`strmode`*.

— Otherwise, it clears *`frozen`* in `strmode`.

```
char* str();
```

Effects: Calls `freeze()`, then returns the beginning pointer for the input sequence, *`gbeg`*.
Notes: The return value can be a null pointer.

```
int pcount() const;
```

Effects: If the next pointer for the output sequence, *`pnext`*, is a null pointer, returns zero. Otherwise, returns the current effective length of the array object as the next pointer minus the beginning pointer for the output sequence, *`pnext`* - *`pbeg`*.

D.3.1.3 `strstreambuf` overridden virtual functions [depr.strstreambuf.virtuals]

```
int_type overflow(int_type c = ios_traits<char>::eof());
```

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), assigns *`c`* to `*`*`pnext`*`++`.
Returns `(char)`*`c`*.

— If *`c`* `== eof()`, there is no character to append.
Returns a value other than `eof()`.

1 Returns `eof()` to indicate failure.
Notes: The function can alter the number of write positions available as a result of any call.
To make a write position available, the function reallocates (or initially allocates) an array object with a sufficient number of elements *`n`* to hold the current array object (if any), plus at least one additional write position. How many additional write positions are made available is otherwise unspecified.[256] If *`palloc`* is not a null pointer, the function calls `(*`*`palloc`*`)(`*`n`*`)` to allocate the new dynamic array

[256] An implementation should consider *`alsize`* in making this decision.

object. Otherwise, it evaluates the expression `new charT[`*`n`*`]`. In either case, if the allocation fails, the function returns `eof()`. Otherwise, it sets *`allocated`* in *`strmode`*.

2 To free a previously existing dynamic array object whose first element address is *`p`*: If *`pfree`* is not a null pointer, the function calls `(*`*`pfree`*`)(`*`p`*`)`. Otherwise, it evaluates the expression `delete[]` *`p`*.

3 If *`strmode`* `&` *`dynamic`* `== 0`, or if *`strmode`* `&` *`frozen`* `!= 0`, the function cannot extend the array (reallocate it with greater length) to make a write position available.

```
int_type pbackfail(int_type c = ios_traits<char>::eof());
```

4 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)`*`c`* `== (char)`*`gnext`*`[-1]`, assigns *`gnext`* `- 1` to *`gnext`*.
Returns `(char)`*`c`*.

— If *`c`* `!= eof()`, if the input sequence has a putback position available, and if *`strmode`* & *`constant`* is zero, assigns *`c`* to `*--`*`gnext`*.
Returns `(char)`*`c`*.

— If *`c`* `== eof()` and if the input sequence has a putback position available, assigns *`gnext`* `- 1` to *`gnext`*.
Returns `(char)`*`c`*.

5 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. The function can alter the number of putback positions available as a result of any call.

```
int_type underflow();
```

Effects: Reads a character from the *input sequence*, if possible, without moving the stream position past it, as follows:

— If the input sequence has a read position available the function signals success by returning `(char*)`*`gnext`*.

— Otherwise, if the current write next pointer *`pnext`* is not a null pointer and is greater than the current read end pointer *`gend`*, makes a *read position* available by: assigning to *`gend`* a value greater than *`gnext`* and no greater than *`pnext`*.
Returns `(char)*`*`gnext`*.

6 Returns `eof()` to indicate failure.
Notes: The function can alter the number of read positions available as a result of any call.

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

Effects: Alters the stream position within one of the controlled sequences, if possible, as indicated in Table 94:

Table 94—`seekoff` positioning

Conditions	Result
`(`*`which`*` & ios::in) != 0`	positions the input sequence
`(`*`which`*` & ios::out) != 0`	positions the output sequence
Otherwise, `(`*`which`*` & (ios::in | ios::out)) == (ios::in | ios::out))` and *`way`*` == ` either `ios::beg` or `ios::end`	positions both the input and the output sequences
Otherwise,	the positioning operation fails.

7 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 95:

Table 95—`newoff` values

Condition	**`newoff` Value**
`way`` == ios::beg`	0
`way`` == ios::cur`	the next pointer minus the beginning pointer (*`xnext`* - *`xbeg`*)
`way`` == ios::end`	*`seekhigh`* minus the beginning pointer (*`seekhigh`* - *`xbeg`*)
`If (`*`newoff`*` + `*`off`*`) < (`*`seeklow`*` - `*`xbeg`*`),` or `(`*`seekhigh`*` - `*`xbeg`*`) < (`*`newoff`*` + `*`off`*`)`	the positioning operation fails

8 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.

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

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`*` & ios::in) != 0`, positions the input sequence.

— If `(`*`which`*` & ios::out) != 0`, positions the output sequence.

— If the function positions neither sequence, the positioning operation fails.

9 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 (*`seekhigh`* - *`seeklow`*), 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.

```
streambuf<char>* setbuf(char* s, streamsize n);
```

Effects: Performs an operation that is defined separately for each class derived from `strstreambuf`.
Default behavior: the same as for `streambuf::setbuf(char*, streamsize)`.

D.3.2 Template class `istrstream` [depr.istrstream]

```
namespace std {
  class istrstream : public istream<char> {
  public:
    explicit istrstream(const char* s);
    explicit istrstream(char* s);
    istrstream(const char* s, streamsize n);
    istrstream(char* s, streamsize n);
    virtual ~istrstream();

    strstreambuf* rdbuf() const;
    char *str();
  private:
//  strstreambuf sb;      exposition only
  };
}
```

1 The class `istrstream` supports the reading of objects of class `strstreambuf`. It supplies a `strstreambuf` object to control the associated array object. For the sake of exposition, the maintained data is presented here as:

— *sb*, the `strstreambuf` object.

D.3.2.1 `istrstream` constructors [depr.istrstream.cons]

```
explicit istrstream(const char* s);
explicit istrstream(char* s);
```

Effects: Constructs an object of class `istrstream`, initializing the base class with `istream(&`*sb*`)` and initializing *sb* with `strstreambuf(`*s*`,0))`. *s* shall designate the first element of an NTBS.

```
istrstream(const char* s, streamsize n);
```

Effects: Constructs an object of class `istrstream`, initializing the base class with `istream(&`*sb*`)` and initializing *sb* with `strstreambuf(`*s*`,`*n*`))`. *s* shall designate the first element of an array whose length is *n* elements, and *n* shall be greater than zero.

D.3.2.2 Member functions [depr.istrstream.members]

```
strstreambuf* rdbuf() const;
```

Returns: `(strstreambuf*)&`*sb*.

```
char* str();
```

Returns: `rdbuf()->str()`.

D.3.3 Template class `ostrstream` [depr.ostrstream]

```
namespace std {
  class ostrstream : public ostream<char> {
  public:
    ostrstream();
    ostrstream(char* s, int n, ios_base::openmode mode = ios_base::out);
    virtual ~ostrstream();

    strstreambuf* rdbuf() const;
    void freeze(int freezefl = 1);
    char* str();
    int pcount() const;
  private:
//  strstreambuf sb;      exposition only
  };
}
```

1 The class `ostrstream` supports the writing of objects of class `strstreambuf`. It supplies a `strstreambuf` object to control the associated array object. For the sake of exposition, the maintained data is presented here as:

— *sb*, the `strstreambuf` object.

D.3.3.1 `ostrstream` constructors [depr.ostrstream.cons]

```
ostrstream();
```

Effects: Constructs an object of class `ostrstream`, initializing the base class with `ostream(&`*sb*`)` and initializing *sb* with `strstreambuf())`.

```
ostrstream(char* s, int n, ios_base::openmode mode = ios_base::out);
```

Effects: Constructs an object of class `ostrstream`, initializing the base class with `ostream(&`*sb*`)`, and initializing *sb* with one of two constructors:

— If *mode* `& app == 0`, then *s* shall designate the first element of an array of *n* elements.
The constructor is `strstreambuf(`*s*`, `*n*`, `*s*`)`.

— If *mode* `& app != 0`, then *s* shall designate the first element of an array of *n* elements that contains an NTBS whose first element is designated by *s*.
The constructor is `strstreambuf(`*s*`, `*n*`, `*s*` + ::strlen(`*s*`))`.[257)]

D.3.3.2 Member functions [depr.ostrstream.members]

```
strstreambuf* rdbuf() const;
```

Returns: `(strstreambuf*)&`*sb*.

```
void freeze(int freezefl = 1);
```

257) The function signature `strlen(const char*)` is declared in `<cstring>` (21.2).

Effects: Calls `rdbuf()->freeze(`*`freezefl`*`)`.

```
char* str();
```

Returns: `rdbuf()->str()`.

```
int pcount() const;
```

Returns: `rdbuf()->pcount()`.