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 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 (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 (3.4). 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.

— ANSI X3/TR–1–82:1982, *American National Dictionary for Information Processing Systems.*

— ISO/IEC 9899:1990, *C Standard*

— ISO/IEC xxxx:199x *Amendment 1 to C Standard*

Box 1

This last title must be filled in when Amendment 1 is approved. The other titles have not been checked for accuracy.

2 The library described in Clause 7 of the C Standard and Clause 4 of Amendment 1 to the C standard is hereinafter called the *Standard C Library*.[1)]

1.3 Definitions [intro.defs]

1 For the purposes of this International Standard, the definitions given in ANSI X3/TR–1–82 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 may 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.

— **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 ir 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 may 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 (13.2): 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.

— **static type:** The *static type* of an expression is the type (3.7) 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;

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

they are required to be diagnosed.

— **unspecified behavior:** Behavior, for a correct program construct and correct data, that depends on the implementation. The range of possible behaviors is delineated by the standard. The implementation is not required to document which behavior occurs.

Subclause 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 manual, 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. Each byte (except perhaps registers) has a unique address.

2 The constructs in a C++ program create, refer to, access, and manipulate *objects* in memory. Each object (except bit-fields) occupies one or more contiguous bytes. Objects are created by definitions (3.1) and *new-expressions* (5.3.4). The *lifetime* of an object begins after any required initialization (8.5) has completed. For objects with destructors, it ends when destruction starts. Each object has a *type* determined by the construct that creates it. The type in turn determines the number of bytes that the object occupies and the interpretation of their contents. Objects may contain other objects, called *sub-objects* (9.2, 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 *alignment* restrictions. An object of one of those types may appear only at an address that is divisible by a particular integer.

1.6 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 error message when presented with any ill-formed program that contains a violation of any rule that is identified as diagnosable in this Standard or of any syntax rule, except as noted herein.

2 Well-formed C++ programs are those that are constructed according to the syntax rules, semantic rules identified as diagnosable, and the One Definition Rule (3.1). If a program is not well-formed but does not contain any diagnosable errors, this Standard places no requirement on processors with respect to that program.

1.7 Program execution [intro.execution]

1 The semantic descriptions in this Standard define a parameterized nondeterministic abstract machine. This 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 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 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 may 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 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.[3)]

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

8 It is important to note that certain contexts in C++ cause the evaluation of a full-expression that results from a syntactic construct other than For*expression*(5.18). 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 is a function-call upon a constructor function with one *assignment-expression* as an argument; again, the function-call is a full-expression.

3) 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.

9 Also note that the evaluation of a full-expression may include the evaluation of subexpressions that are not lexically part of the full-expression. For 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 defines the default argument.

10 There is a sequence point at the completion of evaluation of each full-expression[4].

11 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[5]. Several contexts in C++ cause evaluation of a function call, even though no corresponding function-call syntax appears in the translation unit. For 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 translation unit may be.

12 In the evaluation of each of the expressions

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

there is a sequence point after the evaluation of the first expression[6].

Box 2

The contexts above all correspond to sequence points already specified in ISO C, although they can arise in new syntactic contexts. The Working Group is still discussing whether there is a sequence point after the operand of dynamic-cast is evaluated; this is a context from which an exception might be thrown, even though no function-call is performed. This has not yet been voted upon by the Working Group, and it may be redundant with the sequence point at function-exit.

[4] 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.

[5] The sequence point at the function return is not explicitly specified in ISO C, and may 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.

[6] 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 may be preserved individually or in libraries. The separate translation units of a program communicate (3.4) 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 may be separately translated and then later linked to produce an executable program. (3.4).

2.1 Phases of translation [lex.phases]

1 The precedence among the syntax rules of translation is specified by the following phases.[7)]

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. 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. For 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 constants 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 2.4). 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*.
8. The translation units that will form a program are combined. All external object and function references are resolved.

7) Implementations must behave as if these separate phases occur, although in practice different phases may be folded together.

Box 3
What about shared libraries?

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 For example,

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

becomes

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

2.3 Preprocessing tokens [lex.pptoken]

preprocessing-token:
header-name
identifier
pp-number
character-constant
string-literal
operator
digraph
punctuator
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 constant, a string literal, an operator, a digraph, 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 constants*, *string literals*, *operators*, *punctuators*, *digraphs*, 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 may appear within a preprocessing token only as part of a header name or between the quotation characters in a character constant or string literal.

3 If the input stream has been parsed into preprocessing tokens up to a given character, the next preprocessing token is the longest sequence of characters that could constitute a preprocessing token.

4 The program fragment `1Ex` is parsed as a preprocessing number token (one that is not a valid floating or integer constant 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 constant token), whether or not `E` is a macro name.

5 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 Digraph sequences [lex.digraph]

1 Alternate representations are provided for the operators and punctuators whose primary representations use the "national characters." These include digraphs and additional reserved words.

digraph:
```
<%
%>
<:
:>
%:
```

2 In translation phase 3 (2.1) the digraphs are recognized as preprocessing tokens. Then in translation phase 7 the digraphs and the additional identifiers listed below are converted into tokens identical to those from the corresponding primary representations, as shown in Table 2.

Table 2—identifiers that are treated as operators

alternate	*primary*	*alternate*	*primary*	*alternate*	*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 (which include strings and character and numeric constants), 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 If the input stream has been parsed into tokens up to a given preprocessing token, the next token is taken to be the longest string of preprocessing tokens that could possibly constitute a token.

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 form-feed or a vertical-tab character in such a comment, only white-space characters may 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.

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 must be 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 may not be used otherwise in phases 7 and 8:

Table 3—keywords

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

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

Table 4—alternate 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 should be avoided by users; no diagnostic is required.

4 The ASCII representation of C++ programs uses as operators or for punctuation the characters shown in Table 5.

Table 5—operators and punctuation characters

!	%	^	&	*	(	)	-	+	=	{	}	|	~
[	]	\	;	′	:	"	<	>	?	,	.	/	

Table 6 shows the character combinationations that are used as operators.

Table 6—character combinations used as operators

->	++	--	.*	->*	<<	>>	<=	>=	==	!=	&&
||	*=	/=	%=	+=	-=	<<=	>>=	&=	^=	|=	::

Each is converted to a single token in translation phase 7 (2.1).

5 Table 7 shows character combinations that are used as alternative representations for certain operators and punctuators (2.4).

Table 7—digraphs

<%	%>	<:	:>	%:

Each of these is also recognized as a single token in translation phases 3 and 7.

6 Table 8 shows additional tokens that are used by the preprocessor.

Table 8—preprocessing tokens

#	##	%:	%:%:

7 Certain implementation-dependent properties, such as the type of a `sizeof` (5.3.3) and the ranges of fundamental types (3.7.1), are defined in the standard header files (16.2)

```
<float.h>   <limits.h>   <stddef.h>
```

These headers are part of the ISO C standard. In addition the headers

```
<new.h>   <stdarg.h>   <stdlib.h>
```

define the types of the most basic library functions. The last two headers are part of the ISO C standard; `<new.h>` is C++ specific.

2.9 Literals [lex.literal]

1 There are several kinds of literals (often referred to as “constants”).

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`

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 digits starting with `0` is taken to be an octal integer (base eight). The digits `8` and `9` are not octal digits. 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. For 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

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`. They are intended for character sets where a character does not fit into a single byte. 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 `\`, may be represented according to Table 9.

Table 9—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 dependent if it exceeds that of the largest `char` (for ordinary literals) or `wchar_t` (for wide literals).

2.9.3 Floating literals [lex.fcon]

floating-constant:
  fractional-constant exponent-part$_{opt}$ *floating-suffix*$_{opt}$
  digit-sequence exponent-part floating-suffix$_{opt}$

fractional-constant:
  digit-sequence$_{opt}$ `.` *digit-sequence*
  digit-sequence `.`

exponent-part:
  `e` *sign*$_{opt}$ *digit-sequence*
  `E` *sign*$_{opt}$ *digit-sequence*

sign: one of
  `+ -`

digit-sequence:
  digit
  digit-sequence digit

floating-suffix: one of
  `f l F L`

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) may be missing; either the decimal point or the letter `e` (or `E`) and the exponent (not both) may 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 (3.6), 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 dependent. 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-character string. A wide-character string is of type "array of *n* `wchar_t`," where *n* is the size of the string as defined below. Concatenation of ordinary and wide-character string literals is undefined.

Box 4
Should this render the program ill-formed? Or is it deliberately undefined to encourage extensions?

3 Adjacent string literals are concatenated. Characters in concatenated strings are kept distinct. For 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 `"` must 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 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.

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 For example, in

```
int j = 24;

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*.

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 may potentially refer to the same entity in these translation units depending on the linkage (3.4) specified in the translation units.

10 An *object* is a region of storage (3.8). In addition to giving it a name, declaring an object gives the object a *storage duration*, (3.6), which determines the object's lifetime. Some objects are *polymorphic*; 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 expressions used to access them.

Box 5
Most of this section needs more work.

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 The following, for example, 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
```

4 In some circumstances, C++ implementations generate definitions automatically. These definitions include default constructors, copy constructors, assignment operators, and destructors. For example, given

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

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

the implementation will generate 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() { }
};
```

5 A class name can also implicitly be declared by an *elaborated-type-specifier* (7.1.5.3).

3.2 One definition rule [basic.def.odr]

Box 6

This is still very much under review by the Committee.

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 may appear explicitly in the program, it may be found in the standard or a user-defined library, or (when appropriate) the implementation may generate it. 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.

Box 7

This says nothing about user-defined libraries. Probably it shouldn't, but perhaps it should be more explicit that it isn't discussing it.

3 Exactly one definition in a program is required for a non-local variable with static storage duration, unless it has a builtin type or is an aggregate and also is unused or used only as the operand of the `sizeof` operator.

Box 8

This is still uncertain.

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.

Box 9

This is not quite right, because it is possible to declare a function that has an undefined class type as its return type, that has arguments of undefined class type.

Box 10

There may be other situations that do not require a class to be defined: extern declarations (i.e. "extern X x;"), declaration of static members, others???

For 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
```

5 There may 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.

Box 11

This will need to be revisited when the ODR is made more precise

6 There may 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.

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

Box 12

This will need to be revisited when the ODR is made more precise

3.3 Declarative regions and scopes [basic.scope]

1 The scope rules are summarized in 10.5.

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 Names of parameters of a function are local to the function and shall not be redeclared in the outermost block of that function.

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 function declarator except the declarator of a function definition, 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 The scope of names introduced by friend declarations is described in 7.3.1.

3 The scope rules for classes are summarized in 9.3.

3.3.6 Name hiding [basic.scope.hiding]

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

4 The region in which a name is visible is called the *reach* of the name.

Box 13
The term 'reach' is defined here but never used. More work is needed with the "descriptive terminology".

3.3.7 Explicit qualification [basic.scope.exqual]

Box 14

The information in this section is very similar to the one provided in 7.3.1.1. The information in these two sections (3.3.7 and 7.3.1.1) should be consolidated in one place.

1 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 file scope name can still be used when it is qualified by the unary `::` operator (5.1).

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). For example:

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

main()
{
    int A;

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

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. For example,

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

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

3 For the point of declaration for an enumerator, see 7.2.

4 The point of declaration of a function with the `extern` or `friend` specifier is in the innermost enclosing namespace just after outermost nested scope containing it which is contained in the namespace.

Box 15

The terms "just after the outermost nested scope" imply name injection. We avoided introducing the concept of name injection in the working paper up until now. We should probably continue to do without.

5 The point of declaration of a class first declared in an *elaborated-type-specifier* is immediately after the identifier;

6 A nonlocal name remains visible up to the point of declaration of the local name that hides it. For example,

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

declares a local array of two integers.

7 The point of instantiation of a template is described in 14.3.

3.4 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 may 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 may 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 may 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).

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

Box 16
What is the linkage of unnamed classes and their members? Unnamed enumeration and their enumerators?

5 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. *

6 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)) may not be used to declare an entity with linkage. For example:

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

This implies that names with no linkage cannot be used as template arguments (14.7).

7 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.

8 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 must be identical, except that such types may differ by the presence or absence of a major array bound (8.3.4). A violation of this rule does not require a diagnostic.

Box 17
This needs to specified more precisely to deal with function name overloading.

9 Linkage to non-C++ declarations can be achieved using a *linkage-specification* (7.5).

3.5 Start and termination [basic.start]

3.5.1 Main function [basic.start.main]

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

2 This function is not predefined by the compiler, it cannot be overloaded, and its type is implementation dependent. The two examples below are allowed on any implementation. It is recommended that any further (optional) parameters be added after `argv`. The function `main()` may be defined as

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

or

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

In the latter form `argc` shall be the number of arguments passed to the program from an 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 zero-terminated strings; and `argv[0]` shall be the pointer to the initial character of a zero-terminated string that represents the name used to invoke the program or `""`. It is guaranteed that `argv[argc]==0`.

3 The function `main()` shall not be called from within a program. The linkage (3.4) of `main()` is implementation dependent. The address of `main()` shall not be taken and `main()` shall not be declared `inline` or `static`. The name `main` is not otherwise reserved. For example, member functions, classes, and enumerations can be called `main`, as may entities in other namespaces.

4 Calling the function

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

declared in `<cstdlib>` (18.3.0.2) terminates the program without leaving the current block and hence without destroying any local variables (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 local variables) 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;
```

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

Box 18
This is still under active discussion by the committee.

1 The initialization of nonlocal static objects (3.6) 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) may 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 default initialization of all static objects to zero (8.5) is performed before any other initialization. Static objects initialized with constant expressions (5.19) are initialized before any dynamic (that is, run-time) initialization takes place. 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()` (15.5.1).

3.5.3 Termination [basic.start.term]

1 Destructors (12.4) for initialized static objects are called when returning from `main()` and when calling `exit()` (18.3.0.2). Destruction is done in reverse order of initialization. The function `atexit()` from `<cstdlib>` can be used to specify that a function must be called at exit. If `atexit()` is to be called, objects initialized before an `atexit()` call may not be destroyed 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()`.

3.6 Storage duration [basic.stc]

1 The storage duration of an object determines its lifetime.

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

3.6.1 Static storage duration [basic.stc.static]

1 All non-local variables have static storage duration; such variables are created and destroyed as described in 3.5 and 6.7.

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.

Box 19
This awaits committee action on the ''as-if'' rule.

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

4 The keyword `static` applied to a class variable in a class definition also determines that it has static storage duration.

3.6.2 Automatic storage duration [basic.stc.auto]

1 Local objects not declared `static` or explicitly declared `auto` or `register` have *automatic* storage duration and are associated with an invocation of a block (7.1.1).

2 Each object with automatic storage duration is initialized (8.5) each time the control flow reaches its definition and destroyed (12.4) whenever control passes from within the scope of the object to outside that scope (6.6).

3 A named automatic object with a constructor or destructor with side effects may not be destroyed before the end of its block, nor may it be eliminated even if it appears to be unused.

3.6.3 Dynamic storage duration [basic.stc.dynamic]

1 Objects can be created dynamically during program execution (1.7), 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` (18.4.1.1.1) and `operator new[]` (18.4.1.3), and the global *deallocation functions* `operator delete` (18.4.1.1.2) and `operator delete[]` (18.4.1.4).

2 These functions are always implicitly declared. The library provides default definitions for them (_lib.header.new_). A C++ program may 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 (3.5). Allocation and/or deallocation functions may 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.

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

1 Allocation functions can be static class member functions or global functions. They may 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).

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 may 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.[8)]

3 If an allocation function is unable to obtain an appropriate block of storage, it may invoke the currently installed `new_handler`[9)] and/or throw an exception (15) of class `alloc` (_lib.alloc_) or a class derived from `alloc`.

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

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

1 Like allocation functions, deallocation functions may 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 but deallocation functions may not be overloaded.

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 null, 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 may free the storage referenced by the pointer given as its argument and renders the pointer *invalid*. The storage may be 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-null, the value of a pointer that refers to deallocated space is *indeterminate*. The effect of dereferencing an indeterminate pointer value is undefined.[10)]

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

1 The storage duration of class subobjects, base class subobjects and array elements is that of their complete object (1.5).

3.6.5 The `mutable` keyword [basic.stc.mutable]

1 The keyword `mutable` is grammatically a storage class specifier but is unrelated to the storage duration (lifetime) of the class member it describes. The mutable keyword is described in 3.8, 5.2.4, 7.1.1 and 7.1.5.1.

3.6.6 Reference duration [basic.stc.ref]

1 A reference can be used to name an existing object denoted by an lvalue. A reference initialization can also create and name a new object from an lvalue (8.5.3).

2 The reference has static storage duration if it is declared non-locally, automatic storage duration if declared locally including as a function parameter, or the storage duration of its class if declared in a class.

3 References may or may not require storage.

4 The duration of a reference is distinct from the duration of the object it refers to except in the case of a reference declaration initialized by an rvalue (8.5.3).

8) 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.

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

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

5 The lifetime of a reference bound in a reference initialization (8.5.3) ends on exit from the scope in which the declaration occurs.

6 The lifetime of a reference bound in a ctor-initializer (12.6.2) is the lifetime of the subobject initialized.

7 The lifetime of a reference bound for a reference parameter in a function call (5.2.2) is until the completion of the call.

8 Access through a reference to an object which no longer exists or has not yet been constructed yields undefined behaviour.

Box 20
Can references be declared auto or static? This section probably does not belong here.

3.7 Types [basic.types]

Box 21
Section 9.2 describes the concept of *layout-compatible* types. Shouldn't this information be described here?

1 There are two kinds of types: fundamental types and compound types. Types may describe objects, references (8.3.2), or functions (8.3.5).

2 Arrays of unknown size and classes that have been declared but not defined are called *incomplete* types because the size and structure of an instance of the type is unknown. Also, the `void` type represents an empty set of values, so that no objects of type `void` ever exist; `void` is an 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*;

3 A class type (such as "`class X`") may 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 may 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.

4 Expressions that have incomplete type are prohibited in some contexts. For 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;

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
}
```

3.7.1 Fundamental types [basic.fundamental]

1 There are several fundamental types. The standard header `<climits>` specifies the largest and smallest values of each for an implementation.

2 Objects declared as characters (`char`) are large enough to store any member of the implementation’s basic character set. If a character from this set is stored in a character variable, its value is equivalent to the integer code of that character. It is implementation-specified whether a `char` object can take on negative values. Characters may 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. 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 subrange 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 may 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 (1.5) as the corresponding signed integer type.[11] An *alignment requirement* is an implementation-dependent restriction on the value of a pointer to an object of a given type (5.4, 1.5).

6 Unsigned integers, declared `unsigned`, obey the laws of arithmetic modulo 2^n where n is the number of bits in the representation of that particular size of integer. This implies that unsigned arithmetic does not overflow.

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` has 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` can be either `true` or `false`.[12] There are no `signed`, `unsigned`, `short`, or `long bool` types or values. As described below, `bool` values behave as integral types. Thus, for example, they 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 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`. Each implementation defines the characteristics of the fundamental floating point types in the

[11] See 7.1.5.2 regarding the correspondence between types and the sequences of *type-specifier*s that designate them.
[12] 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`.

standard header `<cfloat>`.

10 Types `bool`, `char`, `wchar_t`, and the signed and unsigned integer types are collectively called *integral* types. A synonym for integral type is *integer type*. Enumerations (7.2) are not integral, but they can be promoted (4.5) to `int`, `unsigned int`, `long`, or `unsigned long`. *Integral* and *floating* types are collectively called *arithmetic* types.

11 The `void` type specifies an empty set of values. It is used as the return type for functions that do not return a value. No object of type `void` may be declared. Any expression may be explicitly converted to type `void` (5.4); the resulting expression may 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).

3.7.2 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 objects of a given type, 8.3.5;

— *pointers* to 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;

— *structures*, which are classes without default access restrictions, 11;

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

— *pointers to non-static*[13] *class members*, which identify members of a given type within objects of a given class, 8.3.3.

2 In general, 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 Any type so far mentioned is an *unqualified type*. Each unqualified type has three corresponding *qualified versions* of its type:[14] a *const-qualified* version, a *volatile-qualified* version, and a *const-volatile-qualified* version (see 7.1.5). The cv-qualified or unqualified versions of a type are distinct types that belong to the same category and have the same representation and alignment requirements.[15] A compound type is not cv-qualified (3.7.3) by the cv-qualifiers (if any) of the type from which it is compounded. However, an array type is considered to be cv-qualified by the cv-qualifiers of its element type. Moreover, when an array type is cv-qualified, its element type is considered to have the same cv-qualifiers.

4 A pointer to objects of a type `T` is referred to as a "pointer to `T`." For 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`." Pointers to incomplete types are allowed although there are restrictions on what can be done with them (3.7).

5 Objects of cv-qualified (3.7.3) or unqualified type `void*` (pointer to void), can be used to point to objects of unknown type. A `void*` must have enough bits to hold any object pointer.

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

[13] Static class members are objects or functions, and pointers to them are ordinary pointers to objects or functions.
[14] See 8.3.4 and 8.3.5 regarding cv-qualified array and function types.
[15] The same representation and alignment requirements are meant to imply interchangeability as arguments to functions, return values from functions, and members of unions.

3.7.3 CV-qualifiers [basic.type.qualifier]

Box 22

This section covers the same information as section 7.1.5.1. This information should probably be consolidated in one place.

1 There are two *cv-qualifiers*, `const` and `volatile`. When applied to an object, `const` means the program may not change the object, and `volatile` has an implementation-defined meaning.[16)] An object may have both cv-qualifiers.

2 There is a (partial) ordering on cv-qualifiers, so that one object or pointer may be said to be *more cv-qualified* than another. Table 10 shows the relations that constitute this ordering.

Table 10—relations on `const` and `volatile`

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

3 A pointer or reference to cv-qualified type (sometimes called a cv-qualified pointer or reference) need not actually point to a cv-qualified object, but it is treated as if it does. For example, a pointer to `const int` may point to an unqualified `int`, but a well-formed program may not attempt to change the pointed-to object through that pointer even though it may change the same object through some other access path. CV-qualifiers are supported by the type system so that a cv-qualified object or cv-qualified access path to an object may not be subverted without casting (5.4). For example:

```
void f()
{
    int i = 2;          // not cv-qualified
    const int ci = 3;   // cv-qualified (initialized as required)
    ci = 4;             // error: attempt to modify const
    const int* cip;     // pointer to const int
    cip = &i;           // okay: cv-qualified access path to unqualified
    *cip = 4;           // error: attempt to modify through ptr to const
    int* ip;
    ip = cip;           // error: attempt to convert const int* to int*
}
```

4 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.

3.7.4 Type names [basic.type.name]

1 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).

[16)] Roughly, `volatile` means the object may change of its own accord (that is, the processor may not assume that the object continues to hold a previously held value).

3.8 Lvalues and rvalues [basic.lval]

1 Every expression is either an *lvalue* or *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.[17)]

3 Some builtin operators and function calls yield lvalues. For 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 Some builtin operators expect lvalue operands, for example the builtin assignment operators all expect their left hand operands to be lvalues. Other builtin operators yield rvalues, and some expect them. For example the unary and binary `+` operator expect rvalue arguments and yield rvalue results. The discussion of each builtin operator in 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 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.

7 Class rvalues may have qualified types; non-class rvalues always have unqualified types.

8 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.

9 An lvalue or rvalue of class type can also be used to modify its referent under certain circumstances.

Box 23
Provide example and cross-reference.

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

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

12 Array objects cannot be modified, but their elements may 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 may be modified.

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.

[17)] 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 may invoke a member function upon such objects, but the expressions are not lvalues.

4 Standard conversions [conv]

1 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. The destination type is `bool` (6.4).
- 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 will be considered as well; see 12.3.

3 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 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. Such exceptions are given in the descriptions of those operators and contexts.

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

1 An lvalue (3.8) of a 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 unqualified version of `T`. Otherwise (i.e., `T` is a class type), the type of the rvalue is `T`. [18)]

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 See also 3.8.

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.

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.[19)]

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

19) This conversion never applies to member functions because there is no way to obtain an lvalue for a member function.

2 See 13.3 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`." 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.

2 A conversion can add type qualifiers at levels other than the first in multi-level pointers, subject to the following rules:[20)]

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 $0 < k < j$.

3 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`."

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` (3.7.1) 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[21)].

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.

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.

[20)] These rules ensure that const-safety is preserved by the conversion. This conversion never applies to nonstatic member functions because there is no way to obtain an lvalue for a nonstatic member function.
[21)] 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.

4.7 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). 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; otherwise, the value is implementation-defined.

4 If the destination type is `bool`, see 4.13. If the source type is `bool`, the source integer is taken to be zero for `false` and one for `true`.

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 can be 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 result is undefined if the truncated value cannot be represented in the destination type. 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 can be 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 source integer is taken to be zero for `false` and one for `true`.

4.10 Pointer conversions [conv.ptr]

1 A constant expression (5.19) rvalue of an integer type 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`."

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 may 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.[22)]

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 (_class.ambig_) base class of `D`, 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`. *

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

[22)] 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 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 may result in a value and may 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 13.4. 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 (13.4).[23)]

3 This clause defines the operators when applied to types for which they have not been overloaded. Operator overloading cannot modify the rules for operators applied to types for which they are defined by the language itself.

4 Operators may 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 is unspecified. In particular, if a value is modified twice in an expression, the result of the expression is unspecified except where an ordering is guaranteed by the operators involved. For example,

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

5 The handling of overflow and divide by zero in expression evaluation is implementation dependent. 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 (13.2), except when matching an object in a member function call against the `this` parameter type.

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 User-defined conversions of class objects to and from fundamental types, pointers, and so on, can be defined (12.3). If unambiguous (13.2), such conversions will be applied by the compiler wherever a class object appears as an operand of an operator or as a function argument (5.2.2).

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

9 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 will be applied to convert the expression to an rvalue.

10 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."

Box 24

There are many places in this Clause that fail to treat enumerations appropriately. Thus, for example, if `e1` and `e2` are of enumerated type, `e1+e2` is some kind of integer, and what kind depends on whether `e1` or `e2` converts to `long`. The definition of ''usual arithmetic conversions'' should take this into account.

11

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

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

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

— Otherwise, the integral promotions (4.5) are performed on both operands.

— Then, if either operand is `unsigned long` the other is 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` is converted to a `long int`; otherwise both operands are converted to `unsigned long int`.

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

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

— Otherwise, both operands are `int`.

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 In the body of a nonstatic member function (9.4), the keyword `this` names a pointer to the object for which the function was invoked. The keyword `this` cannot be used outside a class member function body.

Box 25

In a constructor it is common practice to allow `this` in *mem-initializers*.

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 is. The identifier or

operator-function-id must be of file scope. Use of `::` allows a type, an object, a function, or an enumerator 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 unadorned 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*

Box 26
Issue: now it's allowed to invoke `~int()`, but `~class-name` doesn't allow for that.

7 An *identifier* is an *id-expression* provided it has been suitably declared (7). For *operator-function-id*s, see 13.4. For *conversion-function-id*s, see 12.3.2. A *class-name* prefixed by ~ denotes a destructor; see 12.4.

qualified-id:
    nested-name-specifier unqualified-id

8 A *nested-name-specifier* that names a class (7.1.5) followed by `::` and the name of a member of that class (9.2), or a member of a base of that class (10), is a *qualified-id*; its type is the type of the member. The result is the member. The result is an lvalue if the member is. The *class-name* may be hidden by a nontype name, in which case 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 must refer to the same class; this notation names the destructor (12.4). Multiply qualified names, such as `N1::N2::N3::n`, can be used to refer to nested types (9.8).

9 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*. For the purpose of this evaluation, the name, if any, of each class is also considered a nested class member of that class.

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 `.` *id-expression*
    postfix-expression `->` *id-expression*
    postfix-expression `++`
    postfix-expression `--`
    `dynamic_cast <` *type-id* `> (` *expression* `)`
    `static_cast <` *type-id* `> (` *expression* `)`
    `reinterpret_cast <` *type-id* `> (` *expression* `)`
    `const_cast <` *type-id* `> (` *expression* `)`
    `typeid (` *expression* `)`
    `typeid (` *type-id* `)`

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

5.2.1 Subscripting [expr.sub]

1 A postfix expression followed by an expression in square brackets is a postfix expression. The intuitive meaning is that of a subscript. One of the expressions must have the type “pointer to `T`” and the other must be of enumeration or integral type. The result is an lvalue of type “`T`.” The type “`T`” must be complete. The expression `E1[E2]` is identical (by definition) to `*((E1)+(E2))`. 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[24] (9.4) call. A function call is a postfix expression followed by parentheses containing a possibly empty, comma-separated list of expressions which constitute the arguments to the function. For ordinary function call, the postfix expression must be a function name, or a pointer or reference to function. For member function call, the postfix expression must be an implicit (9.4) 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. If a function or member function name is used, the name may be overloaded (13), in which case the appropriate function will be selected according to the rules in 13.2. 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 (i.e., the type of the object pointed or referred to by the current value of the object expression).

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 must be complete or the type `void`.

3 When a function is called, each parameter (8.3.5) is initialized (8.5.3, 12.8, 12.1) with its corresponding argument. Standard (4) and user-defined (12.3) conversions are 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. A function may 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

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

variable 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.

4 A function may 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).

5 If no declaration of the called function is accessible from the scope of the call the program is ill-formed. This implies that, except where the ellipsis (`...`) is used, a parameter is available for each argument.

6 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 object of a class for which no parameter is declared is passed as a data structure.

Box 27

To ‘‘pass a parameter as a data structure’’ means, roughly, that the parameter must be a PODS, and that otherwise the behavior is undefined. This must be made more precise.

7 An object of a class for which a parameter is declared is passed by initializing the parameter with the argument by a constructor call before the function is entered (12.2, 12.8).

8 The order of evaluation of arguments is unspecified; take note that compilers differ. 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.

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

10 Recursive calls are permitted.

11 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 must be a class with a suitably declared constructor (8.5, 12.1).

2 A *simple-type-specifier* (7.1.5) followed by a (empty) pair of parentheses constructs a value of the specified type. If the type is a class with a default constructor (12.1), that constructor will be called; otherwise the result is the default value given to a static object of the specified type. See also (5.4).

5.2.4 Class member access [expr.ref]

1 A postfix expression followed by a dot (`.`) or an arrow (`->`) followed by an *id-expression* is a postfix expression. The postfix expression before the dot or arrow is evaluated;[25] 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 may be explicitly invoked for a built-in type, see 12.4. Therefore, 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)[26].

[25] 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.
[26] Note that if `E1` has the type “pointer to class `X`”, then `(*(E1))` is an lvalue.

3 If the *id-expression* is a *qualified-id*, the *nested-name-specifier* of the *qualified-id* may 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 *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. For the purpose of this type lookup, the name, if any, of each class is also considered a nested class member of that class. These searches shall yield a single type which 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). For the purpose of this evaluation, the name, if any, of each class is also considered a nested class member of that class.

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`.

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 “*cq vq* `T`”, then `E1.E2` is an lvalue; the expression designates the named member of the class. The type of `E1.E2` is “*cq vq* `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 “cv-qualifier function of(parameter type list) returning `T`”.

— 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-qualifier 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 the left-hand operand. The type of `E1.E2` is “class `X`’s cv-qualifier 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 constant, and the type of `E2` is “`T`,” the expression `E1.E2` is not an lvalue. The type of `E1.E2` is “`T`”.

7 Note that “class objects” can be structures (9.2) and unions (9.6). Classes are discussed in 9.

5.2.5 Increment and decrement [expr.post.incr]

1 The value obtained by applying a postfix `++` is (a copy of) the value that the operand had before applying the operator. 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` (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 of type `T`, which shall be a pointer or a reference to a complete class type or "pointer to *cv* `void`". The type of `v` shall be a complete pointer type if `T` is a pointer, or a complete reference type if `T` is a reference. Types shall not be defined in a `dynamic_cast`. The `dynamic_cast` operator cannot cast away constness (5.2.10).

2 If `v` has type `T`, the result is `v`. If `T` is a pointer to class `B` and `v` is a pointer to class `D` such that `B` is an unambiguous accessible direct or indirect 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 a reference to class `B` and `v` is a reference to class `D` such that `B` is an unambiguous accessible direct or indirect base class of `D`, the result is a reference to the unique[27)] `B` sub-object of the `D` object referred to by `v`. For example,

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

Otherwise `v` must be a pointer or reference to a polymorphic type (10.3).

3 If `T` is cv-qualified `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`.

4 The run-time check logically executes like this: If, in the complete object pointed (referred) to by `v`, `v` points (refers) to an unambiguous base class sub-object of a `T` object, the result is a pointer (reference) 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*.

Box 28

Comment from Bill Gibbons: the original papers allowed all strict downcasts from accessible bases. This wording does not. The paragraph can be fixed by changing the first instance of ''an unambiguous'' to ''a public.''

5 The value of a failed cast to pointer type is the null pointer, as is the result of applying `const_cast` to a null pointer. A failed cast to reference type throws `bad_cast` (18). For example,

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

```
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
}
```

5.2.7 Type identification [expr.typeid]

1 The result of a `typeid` expression is of type `const type_info&` (18). The value is a reference to a `type_info` object 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. Where the *expression* is a pointer to a polymorphic type dereferenced using `*` or `[`*expression*`]` the `type_info` for the complete object pointed to is the result. If the pointer is zero, the expression throws the `bad_typeid` exception (18). Otherwise, if the pointer does not point to a valid object, the result is undefined.

3 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*.

5.2.8 Static cast [expr.static.cast]

1 The result of the expression `static_cast<T>(v)` is of type `T`. Types may not be defined in a `static_cast.` The conversions that may be performed explicitly using a `static_cast` are listed below. No other conversion may be performed explicitly using a `static_cast.`

2 The `static_cast` operator cannot cast away constness. See 5.2.10.

3 Any implicit conversion (including standard conversions and user-defined conversions) can be performed explicitly using `static_cast.`

4 A value of integral type may 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.

5 An rvalue of type “pointer to *cv* `B`”, where `B` is a complete class, can be explicitly converted to an rvalue of type “pointer to *cv* `D`”, where `D` is a complete class derived (10) from `B`, if a valid pointer conversion from “pointer to *cv* `D`” to “pointer to *cv* `B`” exists (4.10) and if `B` is not a virtual base class of `D`. Such a cast is valid only if the rvalue of type “pointer to *cv* `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 rvalue of type “pointer

to *cv* `B`” does not point to a sub-object of an object of type `D`), the result of the cast is undefined.

6 Aside from this pointer conversion (pointer-to-base to pointer-to-derived), the inverse of any implicit conversion (4) can be performed explicitly using `static_cast` subject to the restriction that the explicit conversion does not cast away constness (5.2.10).

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

8 An rvalue of type “pointer to member of `D` of type *cv* `T`” where `D` is a class type, may be converted to an rvalue of type “pointer to member of `B` of type *cv* `T`, where `B` is a base class (10) of `D`, if a valid pointer conversion from “pointer to *cv* `D`” to “pointer to *cv* `B`” exists (4.10). The null member pointer value (4.11) is converted to the null member pointer value of the destination type. If the pointer to member converted is not the null member pointer value and class `B` does not contain or inherit the original member, the result of the cast is undefined.

9 An lvalue expression of type `T1` may be cast to an lvalue of type `T2` using the reference cast `static_cast<T2&>` if an expression of type “pointer to `T1`” may be explicitly converted to the type “pointer to `T2`” using a `static_cast.`

10 Constructors (12.1) or conversion functions (12.3) are not called as the result of a cast to a reference. Operations performed on the result of a pointer or reference cast refer to the same object as the original (uncast) expression. 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. An implementation shall not create a temporary when an lvalue is cast using a reference cast. For example,

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

Box 29

Is it possible to cast an rvalue to an lvalue using a reference cast? An editorial box in this subclause used to say: [An rvalue expression of type “T” may be explicitly converted to the type “reference to `const X`” if a variable of type “reference to `const X`” can be initialized with an rvalue expression of type “`T`”.] This is not true for lvalues. i.e. the semantics for reference casts on lvalues are quite different from the semantics of reference initialization using lvalues. Does it make sense to have reference casts have different semantics depending on whether they apply to lvalues or rvalues?

11 The result of a reference cast is an lvalue; the results of other casts are not. *

12 An expression may be converted to a class type (only) if an appropriate constructor or conversion operator has been declared (12.3).

13 Any expression may be explicitly converted to type *cv* `void` (3.7.3).

5.2.9 Reinterpret cast [expr.reinterpret.cast]

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

2 The `reinterpret_cast` operator cannot cast away constness; see 5.2.10.

3 The mapping performed by `reinterpret_cast` is implementation-defined; it may, or may 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, but 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 may be an rvalue of type "pointer to incomplete class type". The target type of a pointer cast may be "pointer to incomplete class type". In such cases, if there is any inheritance relationship between the source and target classes, the behavior is undefined.

7 A pointer to a function may 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. See also 4.10.

8 A pointer to an object can be explicitly converted to a pointer to an object of different type. In general, the results of this are unspecified; 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.

Box 30

This does not allow conversion of function pointers to other function pointer types and back. Should it?

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`", may be explicitly converted to an rvalue of type "pointer to member of `Y` of type `T2`", if `T1` and `T2` are both member 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. In general, the result of this conversion is unspecified, except that:

— 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` may be cast to an lvalue of type `T2` using the reference `reinterpret_cast<T2&>` if an rvalue of type "pointer to `T1`" may be explicitly converted to the type "pointer to `T2`" with `reinterpret_cast.`

13 Constructors (12.1) or conversion functions (12.3) are not called as the result of a cast to a reference. Operations performed on the result of a pointer or reference cast refer to the same object as the original (uncast) expression. 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. An implementation shall not creat e a temporary when an lvalue is cast using a reference cast.

14 The result of a cast to a reference type is an lvalue; the results of other casts are not.

5.2.10 Const cast [expr.const.cast]

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

2 An rvalue of type "pointer to *cv1* `T`" may be explicitly converted to the type "pointer to *cv2* `T`", where `T` is any object type and where *cv1* is a different cv-qualification from *cv2*, using the cast `const_cast<`*cv2* `T*>`. An lvalue of type *cv1* `T` may be explicitly converted to an lvalue of type *cv2* `T`, where `T` is any object type and where *cv1* is a different cv-qualification from *cv2*, 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`” may 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* is a different cv-qualification from *cv2* 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 that 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 pouinters. 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 cast-away constness may produce undefined behavior (7.1.5.1).

Box 31
This will need to be reworked once the memory model and object model are ironed out.

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 must be a pointer, and the result is an lvalue referring to the object 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 must be an lvalue, or a *qualified-id*. In the first two cases, 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. For example, the address of an object of type "`const int`" has type "pointer to `const int`." For a *qualified-id*, if the member is not static and of type "`T`" in `class C`, the type of the result is "pointer to member of `class C` of type `T`." For a static member of type "`T`", the type is plain "pointer to `T`." Note that a pointer to member is only formed when an explicit `&` is used and its operand is a *qualified-id* not enclosed in parentheses. For 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.

Box 32

This section probably needs to take into account `const` and its relationship to `mutable`.

3 The address of an object of incomplete type may be taken, but only if the complete type of that object does not have the address-of operator (`operator&()`) overloaded; 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 13.3). Note that since the context may determine whether the operand is a static or nonstatic member function, the context may also affect whether the expression has type "pointer to function" or "pointer to member function."

5 The operand of the unary `+` operator shal 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 ~ must 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`. 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 size, in bytes, of its operand. The operand is either an expression, which is not evaluated, or a parenthesized *type-id*. The `sizeof` operator may not be applied to an expression that has function or incomplete type, or to the parenthesized name of such a type, or to an lvalue that designates a bit-field. A *byte* is unspecified by the language except in terms of the value of `sizeof`; `sizeof(char)` is `1`, but `sizeof(bool)` and `sizeof(wchar_t)` are implementation-defined. [28)]

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.

3 The `sizeof` operator may be applied to a pointer to a function, but not 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 may not be defined in a `sizeof` expression.

6 The result is a constant of type `size_t`, an implementation-dependent unsigned integral type defined in the standard header `<cstddef>`.

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 object or array type (1.5, 3.7).

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

28) `sizeof(bool)` is not required to be `1`.

direct-new-declarator:
  [*expression*]
  direct-new-declarator [*constant-expression*]

new-initializer:
  (*expression-list*$_{opt}$)

Entities created by a *new-expression* have dynamic storage duration (3.6.3). That is, the lifetime of such an entity is not 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. For 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 Parenthese must not appear in a *new-type-id* used as the operand for `new`. For example,

4
```
new int(*[10])();       // error
```

is ill-formed because the binding is

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

The explicitly parenthesized version of the `new` operator can be used to create objects of compound types (3.7.2). For example,

```
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 `const`, `volatile`, 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. Thus, 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 (3.7.1) with a non-negative value. For 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* will be 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* (3.6.3.1). When the allocation function is called, the first argument will be amount of space requested (which may be larger than the size of the object being created only if that object is an array).

10 An implementation provides default definitions of the global allocation functions `operator new()` for non-arrays (18.4.1.1.1) and `operator new[]()` for arrays (18.4.1.3). A C++ program may provide alternative definitions of these functions (_lib.alternate.definitions.for.functions_), and/or class-specific versions (12.5).

11 The *new-placement* syntax can be used to supply additional arguments to an allocation function. 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*, if used (the second and succeeding arguments).

12 For 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[](x)`, and

— `new(2,f) T[5]` results in a call of `operator new[](y,2,f)`.

13 The return value from the allocation function, if non-null, will be assumed to point to a block of appropriately aligned available storage of the requested size, and the object will be created in that block (but not necessarily at the beginning of the block, if the object is an array).

14 If a class has one or more constructors (12.1), a *new-expression* for that class calls one of them to initialize the object. An object of a class can be created by `new` only if suitable arguments are provided to the class' constructors, or if the class has a default constructor (3.1).[29] If the class does not have a default constructor, suitable arguments (13.2) must be provided in a *new-initializer*. If there is no constructor and a *new-initializer* is used, it must be of the form `(`*expression*`)` or `()`. If an expression is present it will be used to initialize the object; if not, or a *new-initializer* is not used, the object will start out with an unspecified value.

15 No initializers can be specified for arrays. Arrays of objects of a class can be created by a *new-expression* only if the class has a default constructor.[30] In that case, the default constructor will be called for each element of the array, in order of increasing address.

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

17 The allocation function may indicate failure by throwing an `alloc` exception (15, _lib.alloc_). In this case no initialization is done.

18 If the constructor throws an exception and the *new-expression* does not contain a *new-placement*, then the deallocation function (3.6.3.2, 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*.

19 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.

Box 33

This is a correction to San Diego resolution 3.5, which on its face seems to require that whether to use `delete` or `delete[]` must be decided purely on syntactic grounds. I believe the intent of the committee was to make the form of `delete` correspond to the form of the corresponding `new`.

20 Whether the allocation function is called before evaluating the constructor arguments, after evaluating the constructor arguments but before entering the constructor, or by the constructor itself 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.

[29] This means that `struct s{}; s* ps = new s;` is allowed on the grounds that `class s` has an implicitly declared constructor.

[30] PODS structs have an implicitly-declared default constructor.

5.3.5 Delete [expr.delete]

1 The *delete-expression* operator destroys a complete object (1.5) 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 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 (1.5) representing a base class of such an object (10).

Box 34
Issue: ... or a class with an unambiguous conversion to such a pointer type ...

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.

3 In the first alternative (*delete object*), if the static type of the operand is different from its dynamic type, the static type must have a virtual destructor or the result is undefined. In the second alternative (*delete array*) if the dynamic type of the object to be deleted is a class that has a destructor and its static type is different from its dynamic type, the result is undefined.

Box 35
This should probably be tightened to require that the static and dynamic types match, period.

4 The deletion of an object may change 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 (3.6.3.2).

5 A C++ program that applies `delete` to a pointer to constant is ill-formed (1.6, 1.7).

6 If the class of the object being deleted is incomplete at the point of deletion and the class has a destructor or an allocation function or a deallocation function, the result is undefined.

7 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).

8 To free the storage pointed to, the *delete-expression* will call a *deallocation function* (3.6.3.2).

9 An implementation provides default definitions of the global deallocation functions `operator delete()` for non-arrays (18.4.1.1.2) and `operator delete[]()` for arrays (18.4.1.4). A C++ program may provide alternative definitions of these functions (_lib.alternate.definitions.for.functions_), and/or class-specific versions (12.5).

10 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 may 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, may be performed using the cast notation of explicit type conversion. The same semantic restrictions and behaviors apply.

5 In addition to those conversions, a pointer to an object of a derived class (10) may 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 must be of type "pointer to member of `T`" to its first operand, which must 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 must be of type "pointer to member of `T`" to its first operand, which must 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 If the result of `.*` or `->*` is a function, then that result can be used only as the operand for the function call operator `()`. For 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 result 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 `/` must have arithmetic type; the operands of `%` must have integral type. The usual arithmetic conversions (_conv.arith_) 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 result 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 dependent.

5.7 Additive operators [expr.add]

1 The additive operators `+` and `-` group left-to-right. The usual arithmetic conversions (_conv.arith_) 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 qualified or 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). 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.[31)]

[31)] 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.

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 must be of integral type and integral promotions are performed. The type of the result is that of the promoted left operand. The result 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. The right shift is guaranteed to be logical (zero-fill) if `E1` has an unsigned type or if it has a non-negative value; otherwise the result is implementation dependent.

5.9 Relational operators [expr.rel]

1 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)`.

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

The operands must 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 must be a qualified or unqualified version of the type of one of the operands. This implies that any pointer may be compared to an integral constant expression evaluating to zero and any pointer can be compared to a pointer of qualified or 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) may be compared; the result depends on the relative positions of the pointed-to objects or functions in the address space.

3 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 higher provided the two members 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, 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.

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

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. (Thus `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 may be compared. Pointer to member conversions (4.11) are performed. A pointer to member may 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 must be a qualified or 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[32) which must be a qualified or 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 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

```
= *= /= %=  += -= >>= <<= &= ^= |=
```

32) This is one instance in which the "composite type", as described in the C Standard, is still employed in C++.

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 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=()` (13.4.3). 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` can be a pointer to a possibly-qualified completely defined object type, in which case `E2` shall have integral type and is converted as explained in 5.7; In all other cases, `E1` and `E2` shall have arithmetic type.

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, for 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; for 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 constant: as array bounds (8.3.4), as `case` expressions (6.4.2), as bit-field lengths (9.7), and as enumerator initializers (7.2).

constant-expression:
  conditional-expression

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

2 Other expressions are considered *constant-expression*s only for the purpose of non-local static object initialization (3.5.2). Such constant expressions shall evaluate to one of the following:
  constant expression, or

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

4 An *address constant* 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) may be used in the creation of an address constant, but the value of an object shall not be accessed by the use of these operators. Function calls cannot 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 cannot 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 can occur only in switch statements.

6.2 Expression statement [stmt.expr]

1 Most statements are expression statements, which have the form

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

Usually expression statements are assignments or function calls. 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; it 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).

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).

2 Note that a declaration is a *statement* (6.7).

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*

The *statement* in a *selection-statement* (both statements, in the `else` form of the `if` statement) implicitly defines a local scope (3.3). That is, if the statement 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 statement. For example,

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

may be equivalently rewritten as

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

Thus after the `if` statement, `i` is no longer in scope.

2 The rules for *condition*s apply both to *selection-statement*s and to the `for` and `while` statements (6.5). The *declarator* may 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 statements controlled by the condition. The value of a *condition* that is an initialized declaration is the value of the initialized 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.

4 A variable, constant, etc. in the outermost block of a statement controlled by a condition may not have the same name as a variable, constant, etc. declared in the condition.

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 `else` is used and the condition yields `false`, the second substatement is executed. The `else` ambiguity is resolved by connecting an `else` with the last encountered `else`-less `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 must be of integral type or of a class type for which an unambiguous conversion to integral type exists (12.3). Integral promotion is performed. Any statement within the statement may 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 may have the same value.

3 There may be at most one label of the form

```
default :
```

within a `switch` statement.

4 Switch statements may 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.

7 Usually, the statement that is the subject of a switch is compound. Declarations may appear in the *statement* 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*
>   *declaration-statement*

2 Note that a *for-init-statement* ends with a semicolon.

3 The *statement* in an *iteration-statement* implicitly defines a local scope (3.3) which is entered and exited each time through the loop. That is, if the statement 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. For example,

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

may be equivalently rewritten as

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

Thus after the `while` statement, `i` is no longer in scope.

4 See 6.4 for the rules on *condition*s.

6.5.1 The `while` statement [stmt.while]

1 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 statement.

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

6.5.2 The `do` statement [stmt.do]

1 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.

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

6.5.3 The `for` statement [stmt.for]

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

2 Either or both of the condition and the expression may be dropped. A missing *condition* makes the implied `while` clause equivalent to `while(true)`.

3 If the *for-init-statement* is a declaration, the scope of the name(s) declared extends to the end of the *for-statement*. For example:

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

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

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

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 (3.6.2) (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). However, the program may be terminated (by calling `exit()` or `abort()`, for example) without destroying class objects with automatic storage duration.

6.6.1 The `break` statement [stmt.break]

1 The `break` statement may 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 may 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 may 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 must 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:
  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 (3.6.2) 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). For 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
}
```

4 The default initialization to zero (8.5) of all local objects with static storage duration (3.6.1) is performed before any other initialization takes place. A local object with static storage duration (3.6.1) 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 must be called either immediately before or as part of the calls of the `atexit()` functions (3.5.3). 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*.

2 To disambiguate, the whole *statement* may have to be examined to determine if it is an *expression-statement* or a *declaration*. This disambiguates many examples. For 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
```

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.

3 The remaining cases are *declaration*s. For example,

```
T(a);          // declaration
T(*b)();       // declaration
T(c)=7;        // declaration
T(d),e,f=3;    // declaration
T(g)(h,2);     // declaration
```

4 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.

5 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). For example,

```
void g()
{
    int f();    // declaration
    int a;      // declaration
    f();        // expression-statement
    a;          // expression-statement
}
```

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:*
> > *decl-specifier-seq*$_{opt}$ *init-declarator-list*$_{opt}$ `;`
> > *function-definition*
> > *template-declaration*
> > *asm-definition*
> > *linkage-specification*
> > *namespace-definition*
> > *namespace-alias-definition*
> > *using-declaration*
> > *using-directive*

*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 _temp.dcls_. *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 10.5. 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, may have declarations nested within them. Unless otherwise stated, utterances in this chapter 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* may 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 may the *decl-specifier-seq* be omitted.

8 Generally speaking, the names declared by a declaration are introduced into the scope in which the declaration occurs. The presence of a `friend` specifier, certain uses of the *elaborated-type-specifier*,and *using-directive*s alter this general behavior, however (see 11.4, 9.1 and 7.3.4)

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 must be self-consistent as described below. For 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` must 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 example,

```
void f(const Pc);       // void f(char* const)  (not const char*)
void g(const int Pc);   // void g(const int)
```

3 Note that 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. For example,

```
void h(unsigned Pc);       // void h(unsigned int)
void k(unsigned int Pc);   // void k(unsigned int)
```

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* may 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 must not be empty. 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 (3.6.2). 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 compiler that the object so declared will be heavily used. The hint may 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 (3.6.1). A `static` specifier may be used in the declaration of class members and its affect is described in 9.5. 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 (3.4) unless the inline declaration 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. A name declared in namespace scope with the `extern` specifier has external linkage unless the declaration matches a previous declaration, in which case the name retains the linkage of the previous declaration. An object or function declared at block scope with the `extern` specifier has external linkage unless the declaration matches a visible declaration of namespace scope that has internal linkage, in which case the object or function has internal linkage and refers to the same object or function denoted by the declaration of namespace scope.[33)]

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 must agree. That is, within a given scope, each declaration declaring the same object name or the same overloading of a function name must imply the same linkage. Each function in a given set of overloaded functions may have a different linkage, however. For example,

```
static char* f(); // f() has internal linkage
char* f()         // f() 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
```

33) Here, ‘‘previously’’ includes enclosing scopes. This implies that a name specified `static` and then specified `extern` in an inner scope still has 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 d;         // 'd' has external linkage
static int d;     // error: inconsistent linkage
```

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. For example,

```
struct S;
extern S a;
extern S f();
extern void g(S);

void h()
{
    g(a);       // error: S undefined
    f();        // error: S undefined
}
```

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`. For example

```
class X {
        mutable const int* p;   // ok
        mutable int* const q;   // ill-formed
};
```

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
```

2 The `inline` specifier is a hint to the compiler that inline substitution of the function body is to be preferred to the usual function call implementation. The hint may be ignored. The `inline` specifier shall not appear on a block scope function declaration. For the linkage of inline functions, see 3.4 and 7.1.1. A function (8.3.5, 9.4, 11.4) defined within the class definition is inline by default.

3 An inline function must be defined in every translation unit in which it is used (3.2), and must 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 must be declared inline in all translation units in which it appears. A call to an inline function may not precede its definition. For 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()
{
    // ...
}
```

4 The `virtual` specifier may be used only in declarations of nonstatic class member functions within a class declaration; see 10.3. *

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 (3.7.1) or compound (3.7.2) types. The `typedef` specifier may not be used in a *function-definition* (8.4), and it may 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. If, in a *decl-specifier-seq* containing the *decl-specifier* `typedef`, there is no *type-specifier*, or the only *type-specifier*s are *cv-qualifier*s, the `typedef` declaration is ill-formed unless the declaration introduces a *type-name-declaration* in the scope of a template definition (14.2). 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. For 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 may be used to redefine the name of any type declared in that scope to refer to the type to which it already refers. For example,

```
typedef struct s { /* ... */ } s;
typedef int I;
typedef int I;
typedef I I;
```

3 In a given scope, a `typedef` specifier may not be used to redefine the name of any type declared in that scope to refer to a different type. For example,

```
class complex { /* ... */ };
typedef int complex;    // error: redefinition
```

Similarly, in a given scope, a class may 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 itself. For example,

```
typedef int complex;
class complex { /* ... */ };  // error: redefinition
```

4 A *typedef-name* that names a class is a *class-name* (9.1). The *typedef-name* may not be used after a `class`, `struct`, or `union` prefix and not in the names for constructors and destructors within the class declaration itself. For example,

```
struct S {
    S();
    ~S();
};

typedef struct S T;

S a = T();      // ok
struct T * p;   // error
```

5 An unnamed class defined in a declaration with a `typedef` specifier gets a dummy name. For linkage purposes only (3.4), the *typedef-name* declared by the declaration is used to denote the class type in place of the dummy name. The *typedef-name* is still only a synonym for the dummy name and may not be used where a true class name is required. Such a class cannot have explicit constructors or destructors because they cannot be named by the user. For example,

```
typedef struct {
    S();    // error: requires a return type since S is
            // an ordinary member function, not a constructor
} S;
```

6 A *typedef-name* that names an enumeration is an *enum-name* (7.2). The *typedef-name* may 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` may be combined with any other *type-specifier*.

— `signed` or `unsigned` may be combined with `char`, `long`, `short`, or `int`.

— `short` or `long` may be combined with `int`.

— `long` may 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. If there is no *type-specifier* or if the only *type-specifier*s present in a *decl-specifier-seq* are *cv-qualifier*s, then the `int` specifier is assumed as default.[34] Regarding the prohibition of the default `int` specifier in `typedef` declarations, see 7.1.3; in all other instances, the use of *decl-specifier-seq*s which contain no *simple-type-specifier*s (and thus default to plain `int`) is deprecated.

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.

7.1.5.1 The *cv-qualifiers* [dcl.type.cv]

Box 36

This section covers the same information as section 3.7.3. This information should probably be consolidated in one place.

1 The presence of a `const` specifier in a *decl-specifier-seq* specifies a `const` object. Except that any class member declared `mutable` (7.1.1) may be modified, any attempt to modify a `const` object after it has been initialized and before it is destroyed results in undefined behavior.

2 Example

```
class X {
    public:
        mutable int i;
        int j;
};
class Y { public: X x; }
const Y y;
y.x.i++;        // defined behavior
y.x.j++;        // undefined behavior
Y* p = const_cast<Y*>(&y);      // cast away const-ness of y
p->x.i = 99;    // defined behavior
p->x.j = 99;    // undefined behavior
```

Unless explicitly declared `extern`, a `const` object does not have external linkage and must be initialized (8.5; 12.1). An integral `const` initialized by an integral constant expression may be used in integral constant expressions (5.19). Each element of a `const` array is `const` and each non-function, non-static, non-mutable member of a `const` class object is `const` (9.4.1).

3 There are no implementation-independent semantics for `volatile` objects; `volatile` is a hint to the compiler to avoid aggressive optimization involving the object because the value of the object may be changed by means undetectable by a compiler. Each element of a `volatile` array is `volatile` and each nonfunction, nonstatic member of a `volatile` class object is `volatile` (9.4.1). An object may be both `const` and `volatile`, with the *type-specifier*s appearing in either order.

[34] Redundant cv-qualifiers are allowed to be introduced through the use of typedefs or template type arguments and are ignored.

Box 37

Notwithstanding the description above, the semantics of `volatile` are intended to be the same in C++ as they are in C. However, it's not possible simply to copy the wording from the C standard until we understand the ramifications of sequence points, etc.

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 (3.7.1). Table 11 summarizes the valid combinations of *simple-type-specifier*s and the types they specify.

Table 11—*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 may 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 may 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* of the form

class-key identifier `;`

then the *elaborated-type-specifier* declares the *identifier* to be a *class-name* in the scope that contains the declaration (9.1). Otherwise, the *identifier* following the *class-key* or `enum` keyword is resolved as described in 10.5 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 enclosing the *elaborated-type-specifier* (3.3).

3 The *class-key* or `enum` keyword present in the *elaborated-type-specifier* must 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* since it can be construed as referring to the definition of the class. Thus, in any *elaborated-type-specifier*, the `enum` keyword must be used to refer to an enumeration (7.2), the `union` *class-key* must be used to refer to a union (9), and either the `class` or `struct` *class-key* must be used to refer to a structure (9) or to a class declared using the `class` *class-key*. For 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
        /* ... */
};

struct Base {
        struct Data;                    // ok: Declares nested Data
        struct ::Data*     thatData;    // ok: Refers to ::Data
        struct Base::Data* thisData;    // ok: Refers to nested Data

        struct Data { /* ... */ };      // Defines nested Data

        struct Data;                    // ok: Redeclares nested Data
};

struct Data;            // ok: Redeclares Data at global scope

struct ::Data;          // error: qualified and nothing declared.
struct Base::Data;      // error: qualified and nothing declared.
struct Base::Datum;     // error: Datum undefined

struct Base::Data* pBase;       // ok: refers to nested Data
```

7.2 Enumeration declarations [dcl.enum]

1 An enumeration is a distinct type (3.7.1) with named constants. Its name becomes an *enum-name*, that is, a reserved word 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 may 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* must be of integral type.

2 For 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*. For 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`,[35] 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 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 $\max(abs(e_{\min}),abs(e_{\max}))$ 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 The value of an enumerator or an object of an enumeration type is converted to an integer by integral promotion (4.5). For 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` may be assigned only values of type `color`. For example,

[35] The type should be larger than `int` only if the value of an enumerator won't fit in an `int`.

```
        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.

8 An expression of arithmetic type or of type `wchar_t` may 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.

Box 38
This means the program does not crash.

9

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 (10.5). An enumerator declared in class scope may be referred to using the class member access operators (`::`, `.` (dot) and `->` (arrow)), see 5.2.4. For 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
            // ...
        }
```

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 a single translation unit.

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:
  original-namespace-definition
  extension-namespace-definition
  unnamed-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* must appear in the global scope or in a namespace scope (3.3.4).

7.3.1.1 Explict qualification [namespace.qual]

Box 39

The information in this section is very similar to the information provided in section 3.3.7. The information should probably be consolidated in one place.

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*.

Box 40
I believe "class-specifier" and "namespace-alias-definition" above should be replaced with "type-name" to include "original-namespace-specifier" and "typedef" as well.

The *class-name*s in a *nested-namespace-specifier* shall have been previously defined by a *class-specifier* 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 names 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.[36] For 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
}
```

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 into the declarative region of the namespace are said to be *member names* of the namespace. For example

```
namespace N {
        int i;
        int g(int a) { return a; }
        void k();
        void q();
}

namespace { int k=1; }
```

[36] 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.

```
namespace N {
        int g(char a)         // overloads N::g(int)
        {
                return k+a;   // k is from unnamed namespace
        }

        int i;                // error: duplicate definition

        void k();             // ok: duplicate function declaration

        void k()              // ok: definition of N::k()
        {
                return g(a);  // calls N::g(int)
        }

        int q();              // error: different return type
}
```

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. For example:

```
namespace Outer {
        int i;
        namespace Inner {
                void f() { i++; } // Outer::i
                int i;
                void g() { i++; } // Inner::i
        }
}
```

3 The use of the `static` keyword is deprecated when declaring objects in a namespace scope (see D); 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. For example:

```
namespace X {
        void f() { /* ... */ }
}
```

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. For 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
}
```

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. For 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
}
```

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. For 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
```

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. For 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;
```

4 A *namespace-name* 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.

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* `;`

Box 41

There is still an open issue regarding the "opt" on the nested-name-specifier.

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. For 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)
};
```

4 A *using-declaration* used as a *member-declaration* must refer to a member of a base class of the class being defined. For 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
};
```

5 A *using-declaration* for a member must declare a member of a derived class. For example:

```
struct X {
        int i;
        static int s;
};

void f()
{
        using X::i;  // error: X::i is a class member
        using X::s;  // error: X::s is a class member
}
```

Box 42

The *using-declaration* for the static member `X::s` could be made to work. My recollection is that it was decided not to allow it.

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). For 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
}
```

7 A *using-declaration* is a *declaration* and can therefore be used repeatedly where (and only where) multiple declarations are allowed. For example:

```
namespace A {
        int i;
}

void f()
{
        using A::i;
        using A::i; // ok: double declaration
}
```

```
class B {
        int i;
};

class X : public B {
        using B::i;
        using B::i;  // error: double member declaration
};
```

Box 43

This resolution is editorial.

8 The entity declared by an *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. For 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)
}
```

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. For 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
}
```

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. For 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)
}
```

12 When a *using-declaration* brings names from a base class into a derived class scope, member functions in the derived class override virtual member functions with the same name and argument types in a base class (rather than conflicting). For 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);   // error: D::h(int) conflicts with 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)
}
```

Box 44

Please check the examples above carefully. They reconcile apparently contradictory votes and WP versions. The examples reflect the view that *using-declarations* and ordinary declarations behave identically for local and class scope except that overriding of virtual functions is allowed.

13 All instances of the name mentioned in a *using-declaration* must be accessible. In particular, if a derived class uses a *using-declaration* to access a member of a base class, the member name must be accessible. If the name is that of an overloaded member function, then all functions named must be accessible.

14 The alias created by the *using-declaration* has the usual accessibility for a *member-declaration.* For 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
};
```

15 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 a namespace at the points where the 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. For example:

```
namespace M {
        int i;
}

namespace N {
        int i;
        using namespace M;
}

void f()
{
        N::i = 7; // error: ambiguous: M::i or N::i?
}
```

4 During overload resolution, all functions from the transitive search must be 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. For example:

```
namespace D {
        int d1;
        void f(int);
        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)
}
```

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 dependent. Typically it is used to pass information through the compiler to an assembler.

7.5 Linkage specifications [dcl.link]

1 Linkage (3.4) 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 dependent. 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++"`. For example,

```
complex sqrt(complex);    // C++ linkage by default
extern "C" {
    double sqrt(double);  // C linkage
}
```

Box 45

This example may need to be revisited depending on what the rules ultimately are concerning C++ linkage to standard library functions from the C library.

2 Linkage specifications nest. A linkage specification does not establish a scope. A *linkage-specification* may 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 has more than one *linkage-specification*, they must agree; that is, they must specify the same *string-literal*. Except for functions with C++ linkage, a function declaration without a linkage specification may not precede the first linkage specification for that function. A function may 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. For example,

```
extern "C" {
    // ...
    _iobuf _iob[_NFILE];
    // ...
    int _flsbuf(unsigned,_iobuf*);
    // ...
}
```

Functions and objects may be declared `static` or `inline` within the `{}` of a linkage specification. The linkage directive is ignored for a function or object with internal linkage (3.4). A function first declared in a linkage specification behaves as a function with external linkage. For 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 Linkage from C++ to objects defined in other languages and to objects defined in C++ from other languages is implementation and language dependent. Only where the object layout strategies of two language implementations are similar enough can such linkage be achieved.

7 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, for 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 may 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.[37)]

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* `)`

37) 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` or `new`, the name of a type must 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. For 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).

3 Note that an *exception-specification* does not affect the function type, so its appearance in an *abstract-declarator* will have empty semantics.

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. For example,

```
struct S {
    S(int);
};

void foo(double a)
{
    S x(int(a));        // function declaration
    S y((int)a);        // object declaration
    S z = int(a);       // object declaration
}
```

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 For example,

```
#include <stddef.h>
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 example,

```
template <class T>
struct S {
T *p;
};
S<int()> x;             // type-id
S<int(1)> y;            // expression (ill-formed)
```

5 For example,

```
void foo()
{
        sizeof(int(1)); // expression
        sizeof(int());  // type-id (ill-formed)
}
```

6 For example,

```
void foo()
{
        (int(1));       // expression
        (int())1;       // type-id (ill-formed)
}
```

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, 13.4), 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. For example, in the declaration

```
int unsigned i;
```

the type specifiers `int unsigned` determine the type "`unsigned int`" (7.1.5.2). Or in general, in the declaration

```
T D
```

the *decl-specifier-seq* `T` determines the type "`T`."

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 may 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 "*type-modifier* `T`," then the type of the identifier of `D` is "*type-modifier cv-qualifier-seq* pointer to `T`." The *cv-qualifier*s apply to the pointer and not to the object pointed to.

2 For 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 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 an unqualified pointer later, for example:

```
        *ppc = &ci;  // okay, but would make p point to ci ...
                     // ... because of previous error
        *p = 5;      // clobber ci
```

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 "*type-modifier* `T`," then the type of the identifier of `D` is "*type-modifier* reference to `T`." At all times during the determination of a type, types of the form "*cv-qualified* reference to `T`" is adjusted to be "reference to `T`". For 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 For 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`.

```
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 There can 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 must 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 an parameter or a return type (8.3.5); see 3.1. A reference must 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 “*type-modifier* `T`,” then the type of the identifier of `D` is “*type-modifier cv-qualifier-seq* pointer to member of *class nested-name-specifier of type* `T`.”

2 For 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
```

3 Note that a pointer to member cannot point to a static member of a class (9.5), a member with reference type, or "*cv* `void`." There are no references to members. 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 "*type-modifier* `T`," then the type of the identifier of `D` is an array type. `T` shall not be a reference 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 "*type-modifier* array of `N T`." If the constant expression is omitted, the type of the identifier of `D` is "*type-modifier* array of unknown bound of `T`," an incomplete object type. The type "*type-modifier* array of `N T`" is a different type from the type "*type-modifier* array of unknown bound of `T`," see 3.7. Any cv-qualifiers that appear in *type-modifier* are applied to the type `T` and not to the array type, as in this 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’’
```

2 An array may be constructed from one of the fundamental types[38] (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 may be omitted only for the first member of the sequence. 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* may 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 The declaration

```
float fa[17], *afp[17];
```

declares an array of `float` numbers and an array of pointers to `float` numbers. The declaration

```
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]` may reasonably appear in an expression.

5 Conversions affecting lvalues of array type are described in 4.2. Objects of array types cannot be modified, see 3.8.

6 Except where it has been declared for a class (13.4.5), 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

[38] The enumeration types are included in the fundamental types.

appearance, subscripting is a commutative operation.

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 For 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 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*$_{opt}$

and the type of the contained *declarator-id* in the declaration `T D1` is "*type-modifier* `T1`," the type of the *declarator-id* in `D` is "*type-modifier cv-qualifier-seq*$_{opt}$ function with parameters of type *parameter-declaration-clause* and returning `T1`"; a type of this form is a *function type*[39].

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 is known only to 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` may not be a parameter type (though types derived from `void`, such as `void*`, may). Where syntactically correct, "`, ...`" is synonymous with "`...`". The standard header `<stdarg.h>` contains a mechanism for accessing arguments passed using the ellipsis, see 17.3.1.2.

[39] As indicated by the syntax, cv-qualifiers are a significant component in function return types.

Box 46

Something should probably be said about how `...` arguments work in C++. For example, do they work for member functions? Virtual member functions? If so, what are the rules?

See 12.1 for the treatment of array arguments.

3 A single name may 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 must 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 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*.

Box 47

Issue: a definition for "signature" will be added as soon as the semantics are made precise.

The return type and the parameter type list, but not the default arguments (8.3.6), 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.[40] 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 9.4.1. It is part of the function type.

4 Functions cannot return arrays or functions, although they can return pointers and references to such things. There are no arrays of functions, although there may be arrays of pointers to functions.

5 Types may not be defined in return or parameter types.

6 The *parameter-declaration-clause* is used to check and convert arguments in calls and to check pointer-to-function and reference-to-function assignments and initializations.

7 An identifier can optionally be provided as a parameter name; if present in a function declaration, it cannot be used since it goes out of scope at the end of the function declarator (3.3); if present in a function definition (8.4), it names a parameter (sometimes called "formal argument"). 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.

8 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

[40] 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" modifiers. This exclusion applies to the parameters of the function, and if a parameter is a pointer to function then to its parameters also, etc.

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.

9 Typedefs are sometimes convenient when the return type of a function is complex. For example, the function `fpif` above could have been declared

```
typedef int  IFUNC(int);
IFUNC*  fpif(int);
```

10 The declaration

```
fseek(FILE*, long, int);
```

declares a function taking three arguments of the specified types. Since no return value type is specified it is taken to be `int` (7.1.5). The declaration

```
printf(const char* ...);
```

declares a function that can be called with varying numbers and types of arguments. For example,

```
printf("hello world");
printf("a=%d b=%d", a, b);
```

It must always have a value, however, that can be converted to a `const char*` as its first argument.

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. All subsequent parameters must have default arguments supplied in this or previous declarations of this function. Default arguments will be used in calls where trailing arguments are missing. A default argument shall not be redefined by a later declaration (not even to the same value). A declaration may add default arguments, however, not given in previous declarations.

2 The declaration

```
point(int = 3, int = 4);
```

declares a function that can be called with zero, one, or two arguments of type `int`. It may 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 Default argument expressions in non-member functions have their names bound and their types checked at the point of declaration, and are evaluated at each point of call. In member functions, names in default argument expressions are bound at the end of the class declaration, like names in inline member function bodies (9.4.2). In the following example, `g` will be called with the value `f(2)`:

```
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))
    }
}
```

Local variables shall not be used in default argument expressions. For example,

```
void f()
{
    int i;
    extern void g(int x = i);   // error
    // ...
}
```

4 Note that default arguments are evaluated before entry into a function and that the order of evaluation of function arguments is implementation dependent. Consequently, parameters of a function may not be used in default argument expressions. Parameters of a function declared before a default argument expression are in scope and may hide namespace and class member names. For 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: ‘float’ called
```

5 Similarly, the declaration of `X::mem1()` in the following example is undefined because no object is supplied for the nonstatic member `X::a` used as an initializer.

```
int b;
class X {
    int a;
    mem1(int i = a); // error: nonstatic member ‘a’
                     // used as default argument
    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.

6 A default argument is not part of the type of a function.

```
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
```

7 An overloaded operator (13.4) shall not have default arguments.

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*

function-body:
  compound-statement

The *declarator* in a *function-definition* shall have the form

`D1 (` *parameter-declaration-clause* `)` *cv-qualifier-seq*$_{opt}$

as described in 8.3.5. A function may be defined only in namespace or class scope. |

2 The parameters are in the scope of the outermost block of the *function-body*.

3 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 9.4.1. It is part of the function type.

6 Note that unused parameters need not be named. For example,

```
void print(int a, int)
{
    printf("a = %d\n",a);
}
```

8.5 Initializers [dcl.init]

1 A declarator may 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 may be initialized by arbitrary expressions involving constants and previously declared variables and functions. |

```
int f(int);
int a = 2;
int b = f(a);
int c(b);
```

3 An expression of type "*cv1* `T`" can initialize an object of type "*cv2* `T`" independently of the cv-qualifiers *cv1* and *cv2*. For example, *

```
int a;                                                          *
const int b = a;
int c = b;
```

4 Default argument expressions are more restricted; see 8.3.6. *

5 The order of initialization of static objects is described in 3.5 and 6.7.

6 Variables with static storage duration (3.6) that are not initialized and do not have a constructor are guaranteed to start off as zero converted to the appropriate type. If the object is a `class` or `struct`, its data members start off as zero converted to the appropriate type. If the object is a `union`, its first data member starts off as zero converted to the appropriate type. The initial values of automatic and register variables that are not initialized are indeterminate. |

7 Note that since `()` is not an 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`.

8 An initializer for a static member is in the scope of the member's class. For example,

```
int a;

struct X {
    static int a;
    static int b;
};

int X::a = 1;
int X::b = a;   // X::b = X::a
```

See 8.3.6 for initializers used as default arguments.

9 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. |

— If the destination type is a reference type, see 8.5.3. |

— If the destination type is a (possibly cv-qualified) class type that is an aggregate (8.5.1), and the initializer is a brace-enclosed list, see 8.5.1. |

— Otherwise, if the destination type or the source type is a (possibly cv-qualified) class type, user-defined conversions are considered. The applicable user-defined conversions are enumerated (13.2.1.3), and the best one is chosen through overload resolution (13.2). The user-defined conversion so selected is called to copy or convert the initializer expression into the object being initialized. If the conversion cannot be done or is ambiguous, the initialization is ill-formed. |

Box 48
Revise 12.6.1 (12.6.1) to reference this. *

— 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 destination type; no user-defined conversions are considered. If the conversion cannot be done, the initialization is ill-formed. |

8.5.1 Aggregates [dcl.init.aggr]

1 An *aggregate* is an array or an object of a class (9) with no user-declared constructors (12.1), no private or protected members (11), no base classes (10), and no virtual functions (10.3). When an aggregate is initialized the *initializer* may be an *initializer-clause* consisting of a brace-enclosed, comma-separated list of initializers 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. If there are fewer initializers in the list than there are members of the aggregate, then the aggregate is padded with zeros of |

the appropriate types.[41)]

2 For 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 zero.

3 An aggregate that is a class can also be initialized with a single non-brace-enclosed expression, as described in 8.5.

4 Braces may be elided as follows. If the *initializer-clause* begins with a left brace, then the succeeding comma-separated list of initializers initializes the members of the aggregate; it is erroneous for there to be more initializers than members. If, however, the *initializer-clause* or a subaggregate does not begin with a left brace, then only enough elements from the list are taken to account for the members of the aggregate; any remaining elements are left to initialize the next member of the aggregate of which the current aggregate is a part.

5 For example,

```
int x[] = { 1, 3, 5 };
```

declares and initializes `x` as a one-dimensional array that has three members, since no size was specified and there are three initializers.

```
float y[4][3] = {
    { 1, 3, 5 },
    { 2, 4, 6 },
    { 3, 5, 7 },
};
```

is a completely-bracketed 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]` is initialized with zeros. Precisely the same effect could have been achieved by

```
float y[4][3] = {
    1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

The last (rightmost) index varies fastest (8.3.4).

6 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]`. Also,

```
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.

7 Initialization of arrays of objects of a class with constructors is described in 12.6.1.

8 The initializer for a union with no constructor is either a single expression of the same type, or a brace-enclosed initializer for the first member of the union. For example,

[41)] The syntax provides for empty initializer clauses, but nonetheless C++ does not have zero length arrays.

```
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
```

9 There may not be more initializers than there are members or elements to initialize. For example,

```
char cv[4] = { 'a', 's', 'd', 'f', 0 };  // error
```

is ill-formed.

10 A *POD-struct*[42] is an aggregate structure that contains neither references nor pointers to members. Similarly, a *POD-union* is an aggregate union that contains neither references nor pointers to members.

8.5.2 Character arrays [dcl.init.string]

1 A `char` array (whether plan `char`, `signed`, or `unsigned`) can be initialized by a string; a `wchar_t` array may be initialized by a wide-character string; successive characters of the string initialize the members of the array. For 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 may not be more initializers than there are array elements. For 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), must be initialized by an object, or function, of type `T` or by an object that can be converted into a `T`. For 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]'
}
```

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.

[42] The acronym POD stands for "plain ol' data."

3 The initializer may 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. For example,

```
int& r1;          // error: initializer missing
extern int& r2;   // ok
```

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 an accessible unambiguous 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 13.2.3.2).

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

  — "*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*, [43] then

6 the reference is bound directly to the initializer expression lvalue. Note that 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.

```
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'
```

*

— Otherwise, the reference must be to a const type (i.e., *cv1* must include const qualification); if not, the program is ill-formed.

  — 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 3.8) 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. [44]

7 The appropriate copy constructor must be callable whether or not the copy is actually done.

[43] This requires a conversion function (12.3.2) returning a reference type, and therefore applies only when `T2` is a class type.
[44] 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.

```
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
```

— 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.

```
const double& rcd2 = 2; // rcd2 refers to temporary
                        // with value '2.0'
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

const volatile int cvi = 1;
const int& r = cvi;     // error: type qualifiers dropped
```

8 The lifetime of a temporary object bound to a reference or containing a sub-object that is bound to a reference is the lifetime of the reference itself (3.6.6).

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 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. 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 Objects of an empty class have a nonzero size.

Box 49

Bill Gibbons suggest that a base class subobject should be allowed to occupy zero bytes of the complete object. This would permit two base class subobjects to have the same address, for example.

4 Class objects may 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 13.4.

5 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).

9.1 Class names [class.name]

1 A class definition introduces a new type. For example,

```
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). For 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()
    // ...
}
```

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. For 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
}
```

Such declarations allow definition of classes that refer to each other. For 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 13.4.

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. For example,

```
struct s { int a; };

void g(int s)
{
    struct s* p = new struct s;    // global 's'
    p->a = s;                      // local 's'
}
```

4 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* must declare at least one member name of the class. A member may not be declared twice in the *member-specification*, except that a nested class may be declared and then later defined.

2 Note that 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. In that case, the member can appear in integral constant expressions (5.19) within its declarative region after its declaration. The member must still be defined elsewhere and the declarator that defines the member shall not contain an *initializer*.

4 A member can be initialized using a constructor; see 12.1.

5 A member may 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* may be used only in the declaration of a virtual function (10.3).

7 Non-`static` (9.5) members that are class objects must be objects of previously declared classes. In particular, a class `cl` may not contain an object of class `cl`, but it may 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 must be specified.

8 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`.

9 Nonstatic data members of a 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 dependent (11.1). Implementation alignment requirements may cause two adjacent members not to be allocated immediately after each other; so may requirements for space for managing virtual functions (10.3) and virtual base classes (10.1); see also 5.4.

10 If two types `T1` and `T2` are the same type, then `T1` and `T2` are *layout-compatible* types.

11 Two POD-struct (8.5.1) types are layout-compatible if they have the same number of members, and corresponding members (in order) have layout-compatible types.

12 Two POD-union (8.5.1) types are layout-compatible if they have the same number of members, and corresponding members (in any order) have layout-compatible types.

Box 50
Shouldn't this be the same *set* of types?

13 Two enumeration types are layout-compatible if they have the same sets of enumerator values.

Box 51
Shouldn't this be the same *underlying type*?

14 If a POD-union contains several 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.

15 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. There may therefore be unnamed padding within a POD-struct object, but not at its beginning, as necessary to achieve appropriate alignment.

16 The range of nonnegative values of a signed integral type is a subrange of the corresponding unsigned integral type, and the representation of the same value in each type is the same.

17 Even if the implementation defines two or more basic types to have the same representation, they are nevertheless different types.

18 The representations of integral types shall define values by use of a pure binary numeration system.

Box 52
Does this mean two's complement? Is there a definition of "pure binary numeration system?"

19 The qualified or unqualified versions of a type are distinct types that have the same representation and alignment requirements.

20 A qualified or unqualified `void*` shall have the same representation and alignment requirements as a qualified or unqualified `char*`.

21 Similarly, pointers to qualified or unqualified versions of layout-compatible types shall have the same representation and alignment requirements.

22 If the program attempts to access the stored value of an object through an lvalue of other than one of the following types:

— the dynamic type of the object,

— a 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 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), or

— a character type.[45] the result is undefined. *

23 A function member (9.4) with the same name as its class is a constructor (12.1). A static data member, enumerator, member of an anonymous union, or nested type may not have the same name as its class.

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 text 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` must 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 meaning 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 both function member bodies and static data member i nitializations).

2 For 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;
};
```

[45] The intent of this list is to specify those circumstances in which an object may or may not be aliased.

9.4 Member functions [class.mfct]

1 A function declared as a member (without the `friend` specifier; 11.4) is called a member function, and is called for an object using the class member syntax (5.2.4). For example,

```
struct tnode {
    char tword[20];
    int count;
    tnode *left;
    tnode *right;
    void set(char*, tnode* l, tnode* r);
};
```

Here `set` is a member function and can be called like this:

```
void f(tnode n1, tnode n2)
{
    n1.set("abc",&n2,0);
    n2.set("def",0,0);
}
```

2 The definition of a member function is considered to be within the scope of its class. This means that (provided it is nonstatic 9.5) it can use names of members of its class directly. Such names then refer to the members of the object for which the function was called.

3 A static local variable in a member function always refers to the same object. A static member function can use only the names of static members, enumerators, and nested types directly. If the definition of a member function is lexically outside the class definition, the member function name must be qualified by the class name using the `::` operator. For example,

```
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;
}
```

The notation `tnode::set` specifies that the function `set` is a member of and in the scope of class `tnode`. The member names `tword`, `count`, `left`, and `right` refer to members of the object for which the function was 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` must be declared elsewhere.

4 Members may be defined (3.1) outside their class definition if they have already been declared but not defined in the class definition; they may not be redeclared. See also 3.4. Function members may be mentioned in friend declarations after their class has been defined. Each member function that is called must have exactly one definition in a program, (no diagnostic required).

5 The effect of calling a nonstatic member function (9.5) of a class `X` for something that is not an object of class `X` is undefined.

9.4.1 The `this` pointer [class.this]

1 In 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*` unless the member function is declared `const` or `volatile`; in those cases, the type of `this` is `const X*` or `volatile X*`, respectively. A function declared `const` and `volatile` has a `this` with the type `const volatile X*`. See also C.3.3. For 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`.

2 A `const` member function (that is, a member function declared with the `const` qualifier) may be called for `const` and non-`const` objects, whereas a non-`const` member function may be called only for a non-`const` object. For 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 could (and does) modify the object for which it was called.

3 Similarly, only `volatile` member functions (that is, a member function declared with the `volatile` specifier) may be invoked for `volatile` objects. A member function can be both `const` and `volatile`.

4 Constructors (12.1) and destructors (12.4) may be invoked for a `const` or `volatile` object. Constructors (12.1) and destructors (12.4) cannot be declared `const` or `volatile`.

9.4.2 Inline member functions [class.inline]

1 A member function may be defined (8.4) in the class definition, in which case it is `inline` (7.1.2). Defining a function within a class definition is equivalent to declaring it `inline` and defining it immediately after the class definition; this rewriting is considered to be done after preprocessing but before syntax analysis and type checking of the function definition. Thus

```
int b;
struct x {
    char* f() { return b; }
    char* b;
};
```

is equivalent to

```
int b;
struct x {
    inline char* f();
    char* b;
};

inline char* x::f() { return b; } // moved
```

Thus the `b` used in `x::f()` is `X::b` and not the global `b`. See also _class.local.type_.

2 Member functions can be defined even in local or nested class definitions where this rewriting would be syntactically incorrect. See 9.9 for a discussion of local classes and 9.8 for a discussion of nested classes.

9.5 Static members [class.static]

1 A data or function member of a class may be declared `static` in the class definition. There is only one copy of a static data member, shared by all objects of the class and any derived classes in a program. A static member is not part of objects of a class. Static members of a global class have external linkage (3.4). The declaration of a static data member in its class definition is *not* a definition and may be of an incomplete type. A definition is required elsewhere; see also C.3. A static data member cannot be mutable.

2 A static member function does not have a `this` pointer so it can access nonstatic members of its class only by using `.` or `->`. A static member function cannot be `virtual`. There cannot be a static and a nonstatic member function with the same name and the same parameter types.

3 Static members of a local class (9.9) have no linkage and cannot be defined outside the class definition. It follows that a local class cannot have static data members.

4 A static member `mem` of class `cl` can be referred to as `cl::mem` (5.1), that is, independently of any object. It can also be referred to using the `.` and `->` member access operators (5.2.4). The static member `mem` exists even if no objects of class `cl` have been created. For example, in the following, `run_chain`, `idle`, and so on exist even if no `process` objects have been created:

```
class process {
    static int no_of_processes;
    static process* run_chain;
    static process* running;
    static process* idle;
    // ...
public:
    // ...
    int state();
    static void reschedule();
    // ...
};
```

and `reschedule` can be used without reference to a `process` object, as follows:

```
void f()
{
    process::reschedule();
}
```

5 Static members of a global class are initialized exactly like global objects and only in file scope. For example,

```
void process::reschedule() { /* ... */ };
int process::no_of_processes = 1;
process* process::running = get_main();
process* process::run_chain = process::running;
```

Static members obey the usual class member access rules (11) except that they can be initialized (in file scope). The initializer of a static member of a class has the same access rights as a member function, as in `process::run_chain` above.

6 The type of a static member does not involve its class name; thus the type of `process :: no_of_processes` is `int` and the type of `&process :: reschedule` is `void(*)()`.

9.6 Unions [class.union]

1 A union may 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 may have member functions (including constructors and destructors), but not virtual (10.3) functions. A union may not have base classes. A union may not be used as a base class. An object of a class with a constructor or a destructor or a user-defined assignment operator (13.4.3) cannot be a

member of a union. A union can have no `static` data members.

Box 53
Shouldn't we prohibit references in unions?

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 must 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). For 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 A global anonymous union must be declared `static`. An anonymous union may not have `private` or `protected` members (11). An anonymous union may not have function members.

4 A union for which objects or pointers are declared is not an anonymous union. For 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 associated with any particular object.

5 Initialization of unions that do not have 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 dependent. 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 dependent. 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 may not be a static member. A bit-field must have integral or enumeration type (3.7.1). It is implementation dependent whether a plain (neither explicitly signed nor unsigned) `int` field is signed or unsigned. The address-of operator & may 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 may 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.

```
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
```

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. For 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
    }
};
```

Member functions and static data members of a nested class can be defined in the global scope. For example,

```
class enclose {
    class inner {
        static int x;
        void f(int i);
    };
};

typedef enclose::inner ei;
int ei::x = 1;

void enclose::inner::f(int i) { /* ... */ }
```

A nested class may be declared in a class and later defined in the same or an enclosing scope. For 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
```

Like a member function, a friend function defined within a class is in the lexical scope of that class; it obeys the same rules for name binding as the member functions (described above and in 10.5) and like them has no special access rights to members of an enclosing class or local variables of an enclosing function (11).

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. For 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
```

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 must be defined within their class definition. A local class may 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. For 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
```

10 Derived classes [class.derived]

1 A list of base classes may be specified in a class declaration 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* must denote a previously declared class (9), which is called a *direct base class* for the class being declared. 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. 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. The scope resolution operator `::` (5.1) may 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 may be implicitly converted to a pointer to an accessible unambiguous base class (4.10). A reference to a derived class may be implicitly converted to a reference to an accessible unambiguous base class (_conv.ref_).

2 For example,

```
class Base {
public:
    int a, b, c;
};

class Derived : public Base {
public:
    int b;
};

class Derived2 : public Derived {
public:
    int c;
};
```

3 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`. 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." For example,

$$\begin{tikzcd} \mathtt{Base} \\ \mathtt{Derived} \arrow[u] \\ \mathtt{Derived2} \arrow[u] \end{tikzcd}$$

Note that the arrows need not have a physical representation in memory and the order in which the sub-objects appear in memory is unspecified.

4 Initialization of objects representing base classes can be specified in constructors; see 12.6.2. *

10.1 Multiple base classes [class.mi]

1 A class may be derived from any number of base classes. For example,

```
class A { /* ... */ };
class B { /* ... */ };
class C { /* ... */ };
class D : public A, public B, public C { /* ... */ };
```

The use of more than one direct base class is often called multiple inheritance.

2 The order of derivation is not significant except possibly for default initialization by constructor (12.1), for cleanup (12.4), and for storage layout (5.4, 9.2, 11.1).

3 A class may not be specified as a direct base class of a derived class more than once but it may be an indirect base class more than once.

```
class B { /* ... */ };
class D : public B, public B { /* ... */ };  // illegal

class L { /* ... */ };
class A : public L { /* ... */ };
class B : public L { /* ... */ };
class C : public A, public B { /* ... */ };   // legal
```

Here, an object of class `C` will have two sub-objects of class `L` as shown below.

$$\begin{tikzcd} \mathtt{L} & & \mathtt{L} \\ \mathtt{A} \arrow[u] & & \mathtt{B} \arrow[u] \\ & \mathtt{C} \arrow[ul] \arrow[ur] & \end{tikzcd}$$

4 The keyword `virtual` may be added to a base class specifier. A single sub-object of the virtual base class is shared by every base class that specified the base class to be virtual. For example,

```
class V { /* ... */ };
class A : virtual public V { /* ... */ };
class B : virtual public V { /* ... */ };
class C : public A, public B { /* ... */ };
```

Here class `C` has only one sub-object of class `V`, as shown below.

$$\begin{tikzcd} & V & \\ A \arrow[ur] & & B \arrow[ul] \\ & C \arrow[ul] \arrow[ur] & \end{tikzcd}$$

5 A class may 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 { /* ... */ };
```

Here class AA has two sub-objects of class B: Z's B and the virtual B shared by X and Y, as shown below.

$$\begin{tikzcd} & B & & B \\ X \arrow[ur] & & Y \arrow[ul] & Z \arrow[u] \\ & AA \arrow[ul] \arrow[ur] \arrow[urr] & & \end{tikzcd}$$

10.2 Member Name Lookup [class.member.lookup]

1 Member name lookup determines the meaning of a name (*id-expression* or *qualified-id*) in a class scope. 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 (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 declarations from distinct sub-objects, there is an ambiguity and the program is ill-formed. Otherwise that set is the result of the lookup.

3 For 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
        }
```

If the name of an overloaded function is unambiguously found overloading resolution also takes place before access control. Ambiguities can often be resolved by qualifying a name with its class name. For example,

```
        class A {
        public:
            int f();
        };

        class B {
        public:
            int f();
        };

        class C : public A, public B {
            int f() { return A::f() + B::f(); }
        };
```

The definition of ambiguity allows a nonstatic object to be found in more than one sub-object. When virtual base classes are used, two base classes can share a common sub-object. For 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’
        }
```

When virtual base classes are used, a hidden declaration may 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. For 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 g(); };
```

$$\begin{tikzcd}
W & & V & & W \\
& B \arrow[ul] \arrow[ur] & & C \arrow[ul] \arrow[ur] & \\
& & D \arrow[ul] \arrow[ur] & &
\end{tikzcd}$$

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::g()
{
    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()
}
```

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 must unambiguously refer to a unique object representing the base class. For 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
}
```

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*[46)]

[46)] A function with the same name but a different parameter list (see 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.

`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.

3 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. See 5.2.2. For 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
}
```

4 That is, 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 refe rence denoting that object (the static type). See 5.2.2.

5 The `virtual` specifier implies membership, so a virtual function cannot be a global (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 can be declared a `friend` in another class. A virtual function declared in a class must be defined or declared pure (10.4) in that class.

6 Following are some examples of virtual functions used 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`.

7 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.

8 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
}
```

9 Explicit qualification with the scope operator (5.1) suppresses the virtual call mechanism. For 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 may 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* (which may 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). For 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
    // ...
};
```

An abstract class may not be used as an parameter type, as a function return type, or as the type of an explicit conversion. Pointers and references to an abstract class may be declared. For example,

```
shape x;           // error: object of abstract class
shape* p;          // ok
shape f();         // error
void g(shape);     // error
shape& h(shape&);  // ok
```

3 Pure virtual functions are inherited as pure virtual functions. For 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(); // must be defined somewhere
};
```

would make class `circle` nonabstract and a definition of `circle::draw()` must be provided.

4 An abstract class may 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 of an abstract class; the effect of calling a pure virtual function directly or indirectly for the object being created from such a constructor is undefined.

10.5 Summary of scope rules [class.scope]

1 The scope rules for C++ programs can now be summarized. These rules apply uniformly for all names (including *typedef-names* (7.1.3) and *class-names* (9.1)) wherever the grammar allows such names in the context discussed by a particular rule. This section discusses lexical scope only; see 3.4 for an explanation of linkage issues. The notion of point of declaration is discussed in (3.3).

2 Any use of a name must be unambiguous (up to overloading) in its scope (_class.ambig_). Only if the name is found to be unambiguous in its scope are access rules considered (11). Only if no access control errors are found is the type of the object, function, or enumerator named considered.

3 A name used outside any function and class or prefixed by the unary scope operator `::` (and *not* qualified by the binary `::` operator or the `->` or `.` operators) must be the name of a global object, function, or enumerator.

4 A name specified after `X::`, after `obj.`, where `obj` is an `X` or a reference to `X`, or after `ptr->`, where `ptr` is a pointer to `X` must be the name of a member of class `X` or be a member of a base class of `X`. In addition, `ptr` in `ptr->` may be an object of a class `Y` that has `operator->()` declared so `ptr->operator->()` eventually resolves to a pointer to `X` (13.4.6).

5 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 must be declared before its use in the block in which it occurs or in an enclosing block or globally. The declaration of a local name hides previous declarations of the same name in enclosing blocks and at file scope. In particular, no overloading occurs of names in different scopes (13.4).

6 A name that is not qualified in any of the ways described above and that is used in a function that is a nonstatic member of class `X` must be declared in the block in which it occurs or in an enclosing block, be a member of class `X` or a base class of class `X`, or be a global name. The declaration of a local name hides declarations of the same name in enclosing blocks, members of the function's class, and global names. The declaration of a member name hides declarations of the same name in base classes and global names.

7 A name that is not qualified in one of the ways described above and is used in a static member function of a class `X` must be declared in the block in which it occurs, in an enclosing block, be a static member of class `X`, or a base class of class `X`, or be a global name.

8 A function parameter name in a function definition (8.4) is in the scope of the outermost block of the function (in particular, it is a local name). A function parameter name in a function declaration (8.3.5) that is not a function definition is in a local scope that disappears immediately after the function declaration. A default argument is in the scope determined by the point of declaration (3.3) of its parameter, but may not access local variables or nonstatic class members; it is evaluated at each point of call (8.3.6).

9 A *ctor-initializer* (12.6.2) is evaluated in the scope of the outermost block of the constructor it is specified for. In particular, it can refer to the constructor's parameter names.

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 and friends of the class in which it is declared.

— `protected`; that is, its name can be used only by member functions and friends of the class in which it is declared and by member functions and friends of classes derived from this class (see 11.5).

— `public`; that is, its name can be used by any function.

2 Members of a class declared with the keyword `class` are `private` by default. Members of a class declared with the keywords `struct` or `union` are `public` by default. For example,

```
class X {
    int a;  // X::a is private by default
};

struct S {
    int a;  // S::a is public by default
};
```

11.1 Access specifiers [class.access.spec]

1 Member declarations may 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. For 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
};
```

Any number of access specifiers is allowed and no particular order is required. For 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
};
```

2 The order of allocation of data members with separate *access-specifier* labels is implementation dependent (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 (but see 13.1.1). 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[47].

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`. For 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`.

3 Because of the rules on pointer conversion (4.10), a static member of a private base class may 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 the same as 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 and bp2->mi is the same as this->mi
}
```

[47] 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.

4 A base class is said to be accessible if its public members are 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). 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 public or protected member of a private or protected base class can be restored to the same level in the derived class by mentioning its *qualified-id* in the public (for public members of the base class) or protected (for protected members of the base class) part of a derived class declaration. Such mention is called an *access declaration*.

2 For 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.

3 An access declaration may not be used to restrict access to a member that is accessible in the base class, nor may it be used to enable access to a member that is not accessible in the base class. For example,

```
class A {
public:
    int z;
};

class B : private A {
public:
    int a;
    int x;
private:
    int b;
protected:
    int c;
};

class D : private B {
public:
    B::a;  // make ‘a’ a public member of D
    B::b;  // error: attempt to grant access
           // can’t make ‘b’ a public member of D
    A::z;  // error: attempt to grant access
protected:
    B::c;  // make ‘c’ a protected member of D
    B::x;  // error: attempt to reduce access
           // can’t make ‘x’ a protected member of D
};

class E : protected B {
public:
    B::a;  // make ‘a’ a public member of E
};
```

The names `c` and `x` are protected members of `E` by virtue of its protected derivation from `B`. An access declaration for the name of an overloaded function adjusts the access to all functions of that name in the base class. For example,

```
class X {
public:
    f();
    f(int);
};

class Y : private X {
public:
    X::f;  // makes X::f() and X::f(int) public in Y
};
```

4 The access to a base class member cannot be adjusted in a derived class that also defines a member of that name. For example,

```
class X {
public:
    void f();
};

class Y : private X {
public:
    void f(int);
    X::f;  // error: two declarations of f
};
```

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. 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);
}
```

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`. For example,

```
class Y {
    friend char* X::foo(int);
    // ...
};
```

All the functions of a class `X` can be made friends of a class `Y` by a single declaration using an *elaborated-type-specifier*[48] (9.1):

```
class Y {
    friend class X;
    // ...
};
```

Declaring a class to be a friend also implies that private and protected names from the class granting friendship can be used in the class receiving it. For 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
};
```

[48] Note that the *class-key* of the *elaborated-type-specifier* is required.

3 For a class or function mentioned as a friend and not previously declared, see 7.3.1.

4 A function first declared in a friend declaration has external linkage (3.4). Otherwise, it retains its previous linkage (7.1.1).

Box 54

Is the following friend declaration well-formed: "friend static void f();" ?

5 A function of namespace scope may 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.

6 Friend declarations are not affected by *access-specifiers* (9.2).

7 Friendship is neither inherited nor transitive. For 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
    }
};
```

11.5 Protected member access [class.protected]

1 A friend or a member function of a derived class can access a protected static member of a base class. A friend or a member function of a derived class can access a protected nonstatic member of one of its base classes only through a pointer to, reference to, or object of the derived class itself (or any class derived from that class). When a protected member of a base class appears in a *qualified-id* in a friend or a member function of a derived class, the *nested-name-specifier* must name the derived class. For example,

```
class B {
protected:
    int i;
};

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)
    int B::*   pmi_B = &B::i;    // illegal
    int D2::*  pmi_D2 = &D2::i;  // ok
}

void D2::mem(B* pb, D1* p1)
{
    pb->i = 1;  // illegal
    p1->i = 2;  // illegal
    i = 3;      // ok (access through ‘this’)
}

void g(B* pb, D1* p1, D2* p2)
{
    pb->i = 1;  // illegal
    p1->i = 2;  // illegal
    p2->i = 3;  // illegal
}
```

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. For example,

```
class B {
public:
    virtual f();
};

class D : public B {
private:
    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
}
```

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. For 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.

12 Special member functions [special]

1 Some member functions are special in that they affect the way objects of a class are created, copied, and destroyed, and how values may be converted to values of other types. Often such special functions are called implicitly. Also, the compiler may generate instances of these functions when the programmer does not supply them. Compiler-generated special functions may be referred to in the same ways that programmer-written functions are.

2 These member functions obey the usual access rules (11). For example, declaring a constructor `protected` ensures that only derived classes and friends can create objects using it.

12.1 Constructors [class.ctor]

1 A member function with the same name as its class is called a constructor; it is used to construct values of its class type. An object of class type will be initialized before any use is made of the object; see 12.6.

2 A constructor can be invoked for a `const` or `volatile` object.[49] A constructor may not be declared `const` or `volatile` (9.4.1). A constructor may not be `virtual`. A constructor may not be `static`.

3 Constructors are not inherited. Default constructors and copy constructors, however, are generated (by the compiler) where needed (12.8). Generated constructors are `public`.

4 A *default constructor* for a class `X` is a constructor of class `X` that can be called without an argument. If no constructor has been declared for class `X`, a default constructor is implicitly declared. The definition for an implicitly-declared default constructor is generated only if that constructor is called. An implicitly-declared default constructor is non-trivial if and only if either the class has direct virtual bases or virtual functions or if the class has direct bases or members of a class (or array thereof) requiring non-trivial initialization (12.6).

5 A *copy constructor* for a class `X` is a constructor whose first parameter is of type `X&` or `const X&` and whose other parameters, if any, all have defaults, so that it can be called with a single argument of type `X`. For example, `X::X(const X&)` and `X::X(X&,`int=0) are copy constructors. If no copy constructor is declared in the class definition, a copy constructor is implicitly declared[50]. The definition for an implicitly-declared copy constructor is generated only if that copy constructor is called.

Box 55
Do we need a definition for trivial implicitly-declared copy constructors?

[49] Volatile semantics might or might not be used.
[50] Thus the class definition

```
        struct X {
            X(const X&, int);
        };
```

causes a copy constructor to be generated and the member function definition

```
        X::X(const X& x, int i =0) { ... }
```

is ill-formed because of ambiguity.

6 A constructor for a class `X` whose first parameter is of type `X` or `const X` (*not* reference types), is not a copy constructor, and must have other parameters. For example, `X::X(X)` is ill-formed.

7 Constructors for array elements are called in order of increasing addresses (8.3.4).

8 If a class has base classes or member objects with constructors, their constructors are called before the constructor for the derived class. The constructors for base classes are called first. See 12.6.2 for an explanation of how arguments can be specified for such constructors and how the order of constructor calls is determined.

9 An object of a class with a constructor cannot be a member of a union.

10 No return type (not even `void`) can be specified for a constructor. A `return` statement in the body of a constructor may not specify a return value. It is not possible to take the address of a constructor.

11 A constructor can be used explicitly to create new objects of its type, using the syntax

class-name `(` *expression-list*$_{opt}$ `)`

For example,

```
complex zz = complex(1,2.3);
cprint( complex(7.8,1.2) );
```

An object created in this way is unnamed (unless the constructor was used as an initializer for a named variable as for `zz` above), with its lifetime limited to the expression in which it is created; see 12.2.

12 Member functions may be called from within a constructor; see 12.7.

12.2 Temporary objects [class.temporary]

1 In some circumstances it may be necessary or convenient for the compiler to generate a temporary object. Precisely when such temporaries are introduced is implementation dependent. When a compiler introduces a temporary object of a class that has a constructor it must ensure that a constructor is called for the temporary object. Similarly, the destructor must be called for a temporary object of a class where a destructor is declared. For 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, one might use a temporary in which to construct `X(2)` before passing it to `f()` by `X(X&)`; alternatively, `X(2)` might be constructed in the space used to hold the argument for the first call of `f()`. Also, a temporary might be used to hold the result of `f(X(2))` before copying it to `b` by `X(X&)`; alternatively, `f()`'s result might be constructed in `b`. On the other hand, for many functions `f()`, 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`. Even if the copy constructor is not called, all the semantic restrictions, such as accessibility, must be satisfied.

2 The compiler must ensure that every temporary object is destroyed. Ordinarily, temporary objects are destroyed as the last step in evaluating the full-expression (1.7) that (lexically) contains the point where they were created. This is true even if that evaluation ends in throwing an exception. |
*

3 The only context in which temporaries are destroyed at a different point is when an expression appears as a declarator initializer. In that context, the temporary that holds the result of the expression must persist at least until the initialization implied by the declarator is complete. If the declarator declares a reference, all temporaries in the expression persist until the end of the scope in which the reference is declared. Otherwise, the declarator defines an object that is initialized from a copy of the temporary; during this copying, an implementation may call the copy constructor many times; the temporary is destroyed as soon as it has been copied. In all cases, temporaries are destroyed in reverse order of creation.

Another form of temporaries is discussed in 8.5.3.

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). For 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. 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 (_class.ambig_, 12.3.2). Conversions obey the access control rules (11). As ever access control is applied after ambiguity resolution (10.5).

4 See 13.2 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 with a single parameter specifies a conversion from its parameter type to the type of its class. For 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))
}
```

When no constructor for class X accepts the given type, no attempt is made to find other constructors or conversion functions to convert the assigned value into a type acceptable to a constructor for class X. For example,

```
class X { /* ... */ X(int); };
class Y { /* ... */ Y(X); };
Y a = 1;                 // illegal: Y(X(1)) not tried
```

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 may not be declared in the *type-specifier-seq*. Neither parameter types nor return type may 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). If *conversion-type-id* is `void` or cv-qualified `void`, the program is ill-formed.

2 Here is an example:

```
class X {
    // ...
public:
    operator int();
};

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()`. User-defined conversions are not restricted to use in assignments and initializations. For example,

```
void g(X a, X b)
{
    int i = (a) ? 1+a : 0;
    int j = (a&&b) ? a+b : i;
    if (a) { // ...
    }
}
```

3 The *conversion-type-id* in a *conversion-function-id* is the longest possible sequence of *conversion-declarator*s. This prevents ambiguities between the declarator operator * and its expression counterparts. For 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.

4 Conversion operators are inherited.

5 Conversion functions can be virtual.

6 At most one user-defined conversion (constructor or conversion function) is implicitly applied to a single value. For 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()
```

7 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. For example,

```
class X {
public:
    // ...
    operator int();
};

class Y : public X {
public:
    // ...
    operator void*();
};

void f(Y& a)
{
    if (a) {    // error: ambiguous
        // ...
    }
}
```

12.4 Destructors [class.dtor]

1 A member function of class `cl` named `~cl` is called a destructor; it is used to destroy values of type `cl` immediately before the object containing them is destroyed. 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` or `volatile` object.[51] A destructor may not be declared `const` or `volatile` (9.4.1). A destructor may not be `static`.

2 Destructors are not inherited. If a base or a member of a class has a destructor and no destructor is declared for the class itself a default destructor is generated.

Box 56
A default destructor should be generated if the class has a deallocation function.

This generated destructor calls the destructors for bases and members of its class. Generated destructors are `public`.

[51] Volatile semantics might or might not be used.

3 The body of a destructor is executed before the destructors for member or base objects. Destructors for nonstatic member objects are executed in reverse order of their declaration before the destructors for base classes. Destructors for nonvirtual base classes are executed in reverse order of their declaration in the derived class before destructors for virtual base classes. Destructors for virtual base classes are executed in the reverse order of their appearance in a depth-first left-to-right traversal of the directed acyclic graph of base classes; "left-to-right" is the order of appearance of the base class names in the declaration of the derived class. Destructors for elements of an array are called in reverse order of their construction.

4 A destructor may be declared `virtual` or pure `virtual`. In either case if any objects of that class or any derived class are created in the program the destructor must be defined.

5 Member functions may be called from within a destructor; see 12.7.

6 An object of a class with a destructor cannot be a member of a union.

7 Destructors are invoked implicitly (1) when an automatic variable (3.6) or temporary (12.2, 8.5.3) object goes out of scope, (2) for constructed static (3.6) objects at program termination (3.5), 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 dealloation function (5.3.5, 12.5). For 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
}
```

8 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
}
```

9 Invocation of destructors is subject to the usual rules for member functions, e.g., an object of the appropriate type is required (except invoking `delete` on a null pointer has no effect). When a destructor is invoked for an object, the object no longer exists; if the destructor is explicitly invoked again for the same object the behavior is undefined. For 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.

10 The notation for explicit call of a destructor may be used for any simple type name. For example,

```
int* p;
// ...
p->int::~int();
```

Using the notation for a type that does not have a destructor has no effect. Allowing this enables people to write code without having to know if a destructor exists for a given type.

11 The effect of destroying an object more than once is undefined. This implies that that explicitly destroying a local variable causes undefined behavior on exit from the block, because exiting will attempt to destroy the variable again. This is true even if the block is exited because of an exception.

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 (3.6.3.1).

2 When a non-array object 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 For 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
}
```

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.

7 When an object is deleted by a *delete-expression*, the deallocation function is looked up in the scope of 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.[52)]

9 Any deallocation function for a class `X` is a static member (even if not explicitly declared `static`). For example,

[52)] 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.

```
class X {
    // ...
    void operator delete(void*);
    void operator delete[](void*, size_t);
};

class Y {
    // ...
    void operator delete(void*, size_t);
    void operator delete[](void*);
};
```

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. For 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 Access to the deallocation function is checked statically. Thus even though a different one may actually be executed, the statically visible deallocation function must be accessible. Thus in the example above, if `B::operator delete()` had been `private`, the delete expression would have been ill-formed.

12.6 Initialization [class.init]

1 A class having a user-defined constructor or having a non-trivial implicitly-declared default constructor is said to require non-trivial initialization.

2 An object of a class (or array thereof) with no private or protected non-static data members and that does not require non-trivial initialization can be initialized using an initializer list; see 8.5.1. An object of a class (or array thereof) with a user-declared constructor must either be initialized or have a default constructor (12.1) (whether user- or compiler-declared). The default constructor is used if the object (or array thereof) is not explicitly initialized.

12.6.1 Explicit initialization [class.expl.init]

1 Objects of classes with constructors (12.1) can be initialized with a parenthesized expression list. This list is taken as the argument list for a call of a constructor doing the initialization. Alternatively a single value is specified as the initializer using the `=` operator. This value is used as the argument to a copy constructor. Typically, that call of a copy constructor can be eliminated. For 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’
```

Overloading of the assignment operator `=` has no effect on initialization.

2 The initialization that occurs in argument passing and function return is equivalent to the form

```
T x = a;
```

The initialization that occurs in `new` expressions (5.3.4) and in base and member initializers (12.6.2) is equivalent to the form

```
T x(a);
```

3 Arrays of objects of a class with constructors use constructors in initialization (12.1) just as do individual objects. If there are fewer initializers in the list than elements in the array, a default constructor (12.1) must be declared (whether by the compiler or the user), and it is used; otherwise the *initializer-clause* must be complete. For example,

```
complex cc = { 1, 2 }; // error; use constructor
complex v[6] = { 1,complex(1,2),complex(),2 };
```

Here, `v[0]` and `v[3]` are initialized with `complex::complex(double)`, `v[1]` is initialized with `complex::complex(double,double)`, and `v[2]`, `v[4]`, and `v[5]` are initialized with `complex::complex()`.

4 An object of class `M` can be a member of a class `X` only if (1) `M` has a default constructor, or (2) `X` has a user-declared constructor and if every user-declared constructor of class `X` specifies a *ctor-initializer* (12.6.2) for that member. In case 1 the default constructor is called when the aggregate is created. If a member of an aggregate has a destructor, then that destructor is called when the aggregate is destroyed.

5 Constructors for nonlocal static objects are called in the order they occur in a file; destructors are called in reverse order. See also 3.5, 6.7, 9.5.

12.6.2 Initializing bases and members [class.base.init]

1 The definition of a constructor can specify initializers for direct and virtual base classes and for members not inherited from a base class. This is most useful for class objects, constants, and references where the semantics of initialization and assignment differ. A *ctor-initializer* has the form

ctor-initializer:
: *mem-initializer-list*

mem-initializer-list:
mem-initializer
mem-initializer , *mem-initializer-list*

mem-initializer:
`::`$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name* (*expression-list*$_{opt}$)
identifier (*expression-list*$_{opt}$)

The argument list is used to initialize the named nonstatic member or base class object. This (or for an aggregate (8.5.1), initialization by a brace-enclosed list) is the only way to initialize nonstatic `const` and reference members. For 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);
```

First, the base classes are initialized in declaration order (independent of the order of *mem-initializer*s), then the members are initialized in declaration order (independent of the order of *mem-initializer*s), then the body of `D::D()` is executed (12.1). The declaration order is used to ensure that sub-objects and members are destroyed in the reverse order of initialization.

2 Virtual base classes constitute a special case. Virtual bases are constructed before any nonvirtual bases and in the order they appear on a depth-first left-to-right traversal of the directed acyclic graph of base classes; “left-to-right” is the order of appearance of the base class names in the declaration of the derived class.

3 The class of a *complete object* (1.5) is said to be the *most derived* class for the sub-objects representing base classes of the object. All sub-objects for virtual base classes are initialized by the constructor of the most derived class. If a constructor of the most derived class does not specify a *mem-initializer* for a virtual base class then that virtual base class must have a default constructor. Any *mem-initializer*s for virtual classes specified in a constructor for a class that is not the class of the complete object are ignored. For 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()
```

4 In a ctor-initializer, the effect of calling a non-static member function of a class object whose base classes have not all been initialized is undefined. For example,

```
class A {
public:
    A(int x);
};

class B : public A {
public:
    int f();
    B() : A(f()) {}     // undefined: calls B::f() but B’s A not yet initialized
};
```

A *mem-initializer* is evaluated in the scope of the constructor in which it appears. For example,

```
class X {
    int a;
public:
    const int& r;
    X(): r(a) {}
};
```

initializes `X::r` to refer to `X::a` for each object of class `X`.

5 The identifier of a *ctor-initializer*’s *mem-initializer* in a class’ constructor is looked up in the scope of the class. It must denote a nonstatic data member or the type of a direct or virtual base class. For the purpose of this name lookup, the name, if any, of each class is considered a nested class member of that class. A constructor’s *mem-initializer-list* can initialize a base class using any name that denotes that base class type; the name used may differ from the class definition. The type shall not designate both a direct non-virtual base class and an inherited virtual base class. For example:

```
struct A { A(); };
struct B: public virtual A { };
struct C: public A, public B { C(); };

C::C(): A() { }           // ill-formed: which A?
```

12.7 Constructors and destructors [class.cdtor]

1 Member functions may be called in constructors and destructors. This implies that virtual functions may be called (directly or indirectly). The function called will be the one defined in the constructor's (or destructor's) own class or its bases, but *not* any function overriding it in a derived class. This ensures that unconstructed parts of objects will not be accessed in the body of the constructor or destructor. For example,

```
class X {
public:
    virtual void f();
    X() { f(); }   // calls X::f()
    ~X() { f(); }  // calls X::f()
};

class Y : public X {
    int& r;
public:
    void f()
    {
        r++;  // disaster if ‘r’ is uninitialized
    }
    Y(int& rr) :r(rr) {} // calls X::X() which calls X::f()
};
```

2 The effect of calling a pure virtual function directly or indirectly for the object being constructed from a constructor, except using explicit qualification, is undefined (10.4).

12.8 Copying class objects [class.copy]

1 A class object can be copied in two ways, by assignment (5.17) and by initialization (12.1, 8.5) including function argument passing (5.2.2) and function value return (6.6.3). Conceptually, for a class `X` these two operations are implemented by an assignment operator and a copy constructor (12.1). If not declared by the programmer, they will if possible be automatically defined ("synthesized") as memberwise assignment and memberwise initialization of the base classes and non-static data members of `X`, respectively. An explicit declaration of either of them will suppress the synthesized definition.

2 If all bases and members of a class `X` have copy constructors accepting `const` parameters, the synthesized copy constructor for `X` will have a single parameter of type `const X&`, as follows:

```
X::X(const X&)
```

Otherwise it will have a single parameter of type `X&`:

```
X::X(X&)
```

and programs that attempt initialization by copying of `const X` objects will be ill-formed.

3 Similarly, if all bases and members of a class `X` have assignment operators accepting `const` parameters, the synthesized assignment operator for `X` will have a single parameter of type `const X&`, as follows:

```
X& X::operator=(const X&)
```

Otherwise it will have a single parameter of type `X&`:

```
X& X::operator=(X&)
```

and programs that attempt assignment by copying of `const X` objects will be ill-formed. The synthesized assignment operator will return a reference to the object for which is invoked.

4 Objects representing virtual base classes will be initialized only once by a generated copy constructor. Objects representing virtual base classes will be assigned only once by a generated assignment operator.

5 Memberwise assignment and memberwise initialization implies that if a class `X` has a member or base of a class `M`, `M`'s assignment operator and `M`'s copy constructor are used to implement assignment and initialization of the member or base, respectively, in the synthesized operations. The default assignment operation cannot be generated for a class if the class has:

- — a non-static data member that is a `const` or a reference,
- — a non-static data member or base class whose assignment operator is inaccessible to the class, or
- — a non-static data member or base class with no assignment operator for which a default assignment operation cannot be generated.

Similarly, the default copy constructor cannot be generated for a class if a non-static data member or a base of the class has an inaccessible copy constructor, or has no copy constructor and the default copy constructor cannot be generated for it.

6 The default assignment and copy constructor will be declared, but they will not be defined (that is, a function body generated) unless needed. That is, `X::operator=()` will be generated only if no assignment operation is explicitly declared and an object of class `X` is assigned an object of class `X` or an object of a class derived from `X` or if the address of `X::operator=` is taken. Initialization is handled similarly.

7 If implicitly declared, the assignment and the copy constructor will be public members and the assignment operator for a class `X` will be defined to return a reference of type `X&` referring to the object assigned to.

8 If a class `X` has any `X::operator=()` that has a parameter of class `X`, the default assignment will not be generated. If a class has any copy constructor defined, the default copy constructor will not be generated. For example,

```
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);
```

9 Assignment of class objects `X` is defined in terms of `X::operator=(const X&)`. This implies (12.3) that objects of a derived class can be assigned to objects of a public base class. For example,

```
class X {
public:
    int b;
};

class Y : public X {
public:
    int c;
};
```

```
void f()
{
    X x1;
    Y y1;

    x1 = y1;    // ok
    y1 = x1;    // error
}
```

Here `y1.b` is assigned to `x1.b` and `y1.c` is not copied.

10 Copying one object into another using the default copy constructor or the default assignment operator does not change the structure of either object. For example,

```
struct s {
    virtual f();
    // ...
};

struct ss : public s {
    f();
    // ...
};

void f()
{
    s a;
    ss b;
    a = b;      // really a.s::operator=(b)
    b = a;      // error
    a.f();      // calls s::f
    b.f();      // calls ss::f
    (s&)b = a;  // assign to b’s s part
                // really ((s&)b).s::operator=(a)
    b.f();      // still calls ss::f
}
```

The call `a.f()` will invoke `s::f()` (as is suitable for an object of class `s` (10.3)) and the call `b.f()` will call `ss::f()` (as is suitable for an object of class `ss`). *

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, 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 13.2. For example,

```
double abs(double);
int abs(int);

abs(1);       // call abs(int);
abs(1.0);     // call abs(double);
```

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 that cannot be distinguished by overload resolution cannot be overloaded:

— Since for any type “T,” a parameter of type “T” and a parameter of type ‘‘reference to T” accept the same set of initializer values, function declarations with parameter types differing only in this respect cannot be overloaded.

Box 57

This restriction is hard to check across translation units. Moreover, ambiguities can be detected just fine at call time. Perhaps we should remove it.

For example,

```
int f(int i)
{
    // ...
}

int f(int& r)  // error: function types
               // not sufficiently different
{
    // ...
}
```

It is, however, possible to distinguish between “reference to `const T`,” “reference to `volatile T`,” and plain “reference to `T`” so function declarations that differ only in this respect can be overloaded. Similarly, it is possible to distinguish between “pointer to `const T`,” “pointer to `volatile T`,” and plain “reference to `T`” so function declarations that differ only in this respect can be overloaded.

— Function declarations that differ only in the return type cannot be overloaded.

— Member function declarations that differ only in that one is a `static` member and the other isn't cannot be overloaded (9.5).

3 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). For 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)
```

Enumerations, on the other hand, are distinct types and can be used to distinguish overloaded function declarations. For example,

```
enum E { a };

void f(int i) { /* ... */ }
void f(E i)   { /* ... */ }
```

— 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). Note that only the second and subsequent array dimensions are significant in parameter types (8.3.4).

```
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]);
```

— 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. For 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)
```

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 may be used to distinguish overloaded function declarations. In particular, for any type `T`, inter 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 initialization are equivalent. Consider the following example

```
void f (int i, int j);
void f (int i, int j = 99);         // Ok: redeclaration of f (int, int)
void f (int i = 88, int j = 99);    // 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 ()?
}
```

13.1.1 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. For 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
}
```

A locally declared function is not in the same scope as a function in a containing scope.

```
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
    }
)
```

2 Different versions of an overloaded member function may be given different access rules. For 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]; }
    // ...
};
```

13.2 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 may 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, and certain other properties of the candidate function. The function selected by overload resolution is not guaranteed to be appropriate for the context. Other restrictions, such as the accessibility of the function, may 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 (5.2.2)

— 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 (13.2.1.1)

— Invocation of the operator referenced in an expression (5)

— Invocation of a constructor during initialization of a class object via a parenthesized expression list (12.6.1)

— Invocation of a user-defined conversion during initialization from an expression (8.5, 8.5.3)

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 (13.2.3.1) 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.

13.2.1 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 may 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. For 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 type of the implicit object parameter is "reference to `const volatile X`" to be compatible with any object of type `X`.

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 must 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

— 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.

13.2.1.1 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, an object of class type, or a set of pointers-to-function.

2 Subclauses 13.2.1.1.1 and 13.2.1.1.2, 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[53]. 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 13.2.1.1.1. (The resolution of `&F` in other contexts is described in 13.3.)

13.2.1.1.1 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.

[53] If `F` names a non-static member function, `&F` is a pointer-to-member, which cannot be used with the function call syntax.

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[54]. 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. 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`. 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. Because of the usual name hiding rules, these will all be declared in the same block or namespace. The argument list is the same as the *expression-list* in the call. If the name resolves to a 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, 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[55]. The call is ill-formed, however, if overload resolution selects one of the non-static member functions of `T` in this case.

13.2.1.1.2 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()`. 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`.

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 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[56].

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.

[54] Note that cv-qualifiers on the type of objects are significant in overload resolution for both lvalue and rvalue objects.
[55] 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.
[56] 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.

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)`. 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 funtion, 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.

13.2.1.2 Operators in expressions [over.match.oper]

1 If no operand of the operator 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. For 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.
}
```

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 may 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 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 12 (where @ denotes one of the operators covered in the specified subclause).

Table 12—relationship between operator and function call notation

Subclause	Expression	As member function	As non-member function
13.4.1	@a	(a).operator@ ()	operator@ (a)
13.4.2	a@b	(a).operator@ (b)	operator@ (a, b)
13.4.3	a=b	(a).operator= (b)	
13.4.5	a[b]	(a).operator[](b)	
13.4.6	a->	(a).operator-> ()	
13.4.7	a@	(a).operator@ (0)	operator@ (a, 0)

3 Three sets of candidate functions are constructed as follows:

— If the first operand of the operator is an object or reference to an object of class `X`, the operator could be implemented by a member operator function of `X`. The expression is transformed to a qualified function call per column 3 of Table 12 and a set of candidate functions is constructed for the transformed call according to the rules in 13.2.1.1.1. This set is designated the *member candidates* .

— If the operator is either a unary or binary operator (13.4.1, 13.4.2, or 13.4.7), the operator could be implemented by a non-member operator function. The expression is transformed to an unqualified function call per column 4 of Table 12. The operator name is looked up in the context of the expression following the usual rules for name lookup except that all member functions are ignored. Thus, if the operator name resolves to any declaration, it will be to a non-member function declaration. That function and its overloaded declarations constitute the set of candidate functions designated the *non-member candidates*. Because of the name hiding rules, these will all be declared in the same block or namespace[57].

Box 58

A motion is expected in Valley Forge that would eliminate all name hiding when resolving non-member operator names so that the non-member candidates would include all operators of the same name with a declaration in any enclosing block or namespace.

— In any case, a set of candidate functions, called the `built-in` candidates, is constructed. For the binary operator `,` or the unary operator &, the built-in candidates set is empty. For all other operators, the built-in candidates include all of the built-in operators defined in 13.5 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 13.2.3.1.

4 For the built-in assignment operators, conversions of the left operand are restricted as follows:

— no temporaries may be introduced to hold the left operand

— no user-defined conversions may be applied to achieve a type match with it

5 For all other operators, no such restrictions apply.

6 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. 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.

7 If the operator is the binary operator ,or the unary operator & and overload resolution is unsuccessful, then the operator is assumed to be the built-in operator and interpreted according to clause 5.

[57] Note that 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+
}
```

13.2.1.3 Initialization by user-defined conversions [over.match.user]

1 Under the conditions specified in 8.5 and 8.5.3, a user-defined conversion may be invoked to convert the *assignment-expression* of an *initializer-clause* to the type of the object being initialized (which may be a temporary in the reference case). Overload resolution is used to select the user-defined conversion to be invoked. Assuming that "*cv* T" is the type of the object 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 *assignment-expression* is a class type "*cv* `S`", the conversion functions of `S` and its accessible base classes are considered. Those that are not hidden within `S` and yield type `T` or a type that can be converted to type `T` via a standard conversion sequence (13.2.3.1.1) are candidate functions

2 In both cases, the argument list has one argument, which is the *assignment-expression* of the *initializer-clause*. 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 (13.2.3, 13.2.3.1).

13.2.1.4 Initialization by constructor [over.match.ctor]

1 When objects of classes with constructors are initialized with a parenthesized *expression-list* (12.6.1), 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.

13.2.2 Viable functions [over.match.viable]

1 From the set of candidate functions constructed for a given context (13.2.1), a set of viable functions is chosen, from which the best function will be selected by comparing argument conversion sequences for the best fit (13.2.3). 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 must 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, its parameter list is extended to the right with ellipses so that there are exactly *m* parameters.

— A candidate function having more than *m* parameters is viable only if the *(m+1)*–st parameter has a default initializer (8.3.6). 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 must exist for each argument an *implicit conversion sequence* (13.2.3.1) 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 13.2.3.1.4).

13.2.3 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 13.2.3.1 defines the implicit conversion sequences and subclause 13.2.3.2 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,

Box 59
This implies a strong preference for non-template functions. Please consider this carefully.

— the context is an initialization by user-defined conversion (_over.match.init_) and the standard conversion sequence from the return type of `F1` to the target type is a better conversion sequence than the standard conversion sequence from the return type of `F2` to the target type.

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[58].

3 Examples:

```
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
```

13.2.3.1 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 and 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 may be defined for a given argument-parameter pair, the conversion from the argument to the parameter may still be ill-formed in the final analysis.

3 Except in the context of an initialization by user-defined conversion (13.2.1.3), a well-formed implicit conversion sequence is one of the following forms:

— a *standard conversion sequence* (13.2.3.1.1),

— a *user-defined conversion sequence* (13.2.3.1.2), or

58) 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 may 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.

— an *ellipsis conversion sequence* (13.2.3.1.3).

4 In the context of an initialization by user-defined conversion (i.e., when considering the argument of a user-defined conversion function; see 13.2.1.3), 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 13.2.3.1.4.

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 (13.2.3.1.1).

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 13.2.3.2[59]. If that conversion sequence in not better than all the rest and a function that uses such an 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.

13.2.3.1.1 Standard conversion sequences [over.ics.scs]

1 Table 13 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 13 also has an associated rank (Exact Match, Promotion, or Conversion). These are used to rank standard conversion sequences (13.2.3.2). 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

[59] This rule prevents a function from becoming non-viable because of an ambiguous conversion sequence for one of its parameters. Consider this example,

```
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)`.

(13.2.3.1.4). 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.

Table 13—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

13.2.3.1.2 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 13.2.3 and 13.2.3.1)

3 It should be noted that 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.

13.2.3.1.3 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.

13.2.3.1.4 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 13.2.1). 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. For 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 is one exception: 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.

13.2.3.2 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 13.2.3.1)

— A standard conversion sequence (13.2.3.1.1) is a better conversion sequence than a user-defined conversion sequence or an ellipsis conversion sequence

— A user-defined conversion sequence (13.2.3.1.2) is a better conversion sequence than an ellipsis conversion sequence (13.2.3.1.3)

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 dominant conversion of `S1` is better than the dominant conversion 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 (see the definition of *reference-compatible with added qualification* in 8.5.3).

— User-defined conversion sequence `U1` is a better conversion sequence than another user-defined conversion sequence U2 if the second standard conversion sequence of `U1` is better than the second standard conversion sequence of `U2`.

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:

— If class `B` is derived directly or indirectly from class `A`, conversion of `B*` to `A*` 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::*`

13.3 Address of overloaded function [over.over]

1 A use of a function name without arguments selects, among all functions of that name that are in scope, the (only) function that exactly matches the target. The target may 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 (13.4)

— the return value of a function, operator function, or conversion (6.6.3)

— an explicit type conversion (5.2.3, 5.4)

2 Non-member functions match targets of type "pointer-to-function;" member functions match targets of type "pointer-to-member-function."

3 Note that 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)`.

4 For 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.

5 Note also that there are no standard conversions (4) of one pointer-to-function type or pointer-to-member-function into another (4.10). 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
```

6 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. For 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
```

13.4 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[]
+    -    *    /    %    ^    &    |    ~
!    =    <    >    +=   -=   *=   /=   %=
^=   &=   |=   <<   >>   >>=  <<=  ==   !=
<=   >=   &&   ||   ++   --   ,    ->*  ->
()   []
```

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 (13.4.1 - 13.4.7). They can be explicitly called, though. For example,

```
complex z = a.operator+(b);  // complex z = a+b;
void* p = operator new(sizeof(int)*n);
```

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 must either be a non-static member function or 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, may be changed for specific types by defining operator functions that implement these operators. Except for `operator=`, operator functions are inherited; see 12.8 for the rules for `operator=`.

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).

9 Operators not mentioned explicitly below in 13.4.3 to 13.4.7 act as ordinary unary and binary operators obeying the rules of section 13.4.1 or 13.4.2.

13.4.1 Unary operators [over.unary]

1 A prefix unary operator may 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 13.2.1.2 determine which, if any, interpretation is used. See 13.4.7 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. Consequently, a unary operator can hide a binary operator from an enclosing scope, and vice versa.

13.4.2 Binary operators [over.binary]

1 A binary operator may 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 13.2.1.2 determines which, if any, interpretation is used.

13.4.3 Assignment [over.ass]

1 The assignment function `operator=` must be a non-static member function with exactly one parameter. It implements the assigment operator, `=`. It is not inherited (12.8). Instead, unless the user defines `operator=` for a class `X`, `operator=` is defined, by default, as memberwise assignment of the members of class `X`.

```
X& X::operator=(const X& from)
{
    // copy members of X
}
```

13.4.4 Function call [over.call]

1 `operator()` must be a non-static member function. 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,arg2,arg3)` is interpreted as `x.operator()(arg1,arg2,arg3)` 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 (13.2).

13.4.5 Subscripting [over.sub]

1 `operator[]` must be a non-static member function. 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 (13.2).

13.4.6 Class member access [over.ref]

1 `operator->` must 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 (13.2). It follows that `operator->` must return either a pointer to a class that has a member `m` or an object of or a reference to a class for which `operator->` is defined.

13.4.7 Increment and decrement [over.inc]

1 The prefix and postfix increment operators can 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 parameter, it defines the prefix increment operator `++` for objects of that class. If the function is a member function with one parameter (which must be of type `int`) or a non-member function with two parameters (the second must be of type `int`), it defines the postfix increment operator `++` for objects of that class. When the postfix increment is called, the `int` argument will have value zero. For 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++;
}
```

2 The prefix and postfix decrement operators `--` are handled similarly.

13.5 Built-in operators [over.built]

1 The built-in operators (5) participate in overload resolution (13.2.1.2) as though declared as specified in this section. For `operator,` and unary `operator&`, a built-in operator is selected only if there are no user-defined operator candidates. For all other built-in operators, since they take only operands with non-class type, and operator overload resolution occurs only when an operand expression originally has class type, operator overload resolution can resolve to a built-in operator only when an operand has a class type which has a user-defined conversion to a non-class type appropriate for the operator.

2 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` but excluding e.g. `char`). Similarly, the term *promoted arithmetic type* refers to promoted integral types plus floating types.

3 For every pair (*T*, *VQ*), where *T* is an arithmetic type, and *VQ* is either `volatile` or empty, there exist

```
VQ T&    operator++(VQ T&);
VQ T&    operator--(VQ T&);
T        operator++(VQ T&, int);
T        operator--(VQ T&, int);
```

4 For every pair (*T*, *VQ*), where *T* is a cv-qualified or unqualified complete object type, and *VQ* is either `volatile` or empty, there exist *

```
T*VQ&   operator++(T*VQ&);
T*VQ&   operator--(T*VQ&);
T*      operator++(T*VQ&, int);
T*      operator--(T*VQ&, int);
```

5 For every cv-qualified or unqualified complete object type *T*, there exists

```
T&      operator*(T*);
```

6 For every function type *T*, there exists

```
T&      operator*(T*);
```

7 For every type *T*, there exist

```
T*      operator&(T&);
T*      operator+(T*);
```

8 For every promoted arithmetic type *T*, there exist

```
T       operator+(T);
T       operator-(T);
```

9 For every promoted integral type *T*, there exists

```
T       operator~(T);
```

10 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 exists

```
CV12 T&  operator->*(CV1 C*, CV2 T C::*);
```

where *CV12* is the union of *CV1* and *CV2*.

11 For every pair of promoted arithmetic types *L* and *R*, there exist

```
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*.

12 For every pair of types *T* and *I*, where *T* is a cv-qualified or unqualified complete object type and *I* is a promoted integral type, there exist

```
T*      operator+(T*, I);
T&      operator[](T*, I);
T*      operator-(T*, I);
T*      operator+(I, T*);
T&      operator[](I, T*);
```

13 For every triple (*T*, *CV1*, *CV2*), where *T* is a complete object type, and *CV1* and *CV2* are *cv-qualifier-seq*s, there exists

```
ptrdiff_t operator-(CV1 T*, CV2 T*);
```

14 For every triple (*T*, *CV1*, *CV2*), where *T* is any type, and *CV1* and *CV2* are *cv-qualifier-seq*s, there exist

```
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*);
```

15 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

```
bool    operator==(CV1 T C::*, CV2 T C::*);
bool    operator!=(CV1 T C::*, CV2 T C::*);
```

16 For every pair of promoted integral types *L* and *R*, there exist

```
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*.

17 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

```
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);
```

18 For every pair (*T*, *VQ*), where *T* is any type and *VQ* is either `volatile` or empty, there exists

```
T*VQ&   operator=(T*VQ&, T*);
```

19 For every triple (*T*, *VQ*, *I*), where *T* is a cv-qualified or unqualified complete object type, *VQ* is either `volatile` or empty, and *I* is a promoted integral type, there exist

```
T*VQ&   operator+=(T*VQ&, I);
T*VQ&   operator-=(T*VQ&, I);
```

20 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

```
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);
```

21 For every pair of types *L* and *R*, there exists

```
R       operator,(L, R);
```

22 There also exist

```
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. For 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. For 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* must 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 must be exactly one definition for each template in a program. There can be many declarations. Multiple definitions of a template in a single compilation unit is a required diagnostic. Multiple definitions of a template in different compilation units is a nonrequired diagnostic.

3 The names of a template obey 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 A vector class template might be declared like this:

```
template<class T> class vector {
    T* v;
    int sz;
public:
    vector(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, `vector` is a parameterized type with `T` as its parameter. A class template definition specifies how individual classes can be constructed much as a class definition specifies how individual objects can be constructed.

6 A member template can be defined within its class or separately. For example:

```
template<class T> class string {
public:
        template<class T2> compare(const T2&);
        template<class T2> string(const string<T2>& s) { /* ... */ }
        // ...
};

template<class T> template<class T2> string<T>::compare(const T2& s)
{
        // ...
}
```

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. For example:

```
vector<int> v(10);
vector<int>* p = &v;
```

*Template-id*s that name functions are discussed in 14.9.

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. For 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>
```

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. For 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
```

Box 60
Should we bless a hack allowing `Y<X<1>>`? (yes, that would help users)

6 The name of a class template shall not be declared to refer to any other template, class, function, object, namespace, value, or type in the same scope. Unless explicitly specified to have internal linkage, a template in namespace scope has externaml linkage (3.4). A global template name shall be unique in a program.

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 in the template itself before its use. For example:

```
// no B declared here

class X;

template<class T> class Y {
        class Z; // forward declaration of member class
        typedef T::A; // A is a type name

        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
                T::A* a5; // declare pointer to T’s A
                B* a6;    // B is not a type name:
                          // multiply B by a6
        }
};
```

2 The construct:

type-name-declaration:
`typedef` *qualified-name* `;`

is a *declaration* that states that *qualified-name* must name a type, but gives no clue to what that type might be. The *qualified-name* must include a qualifier containing a template parameter or a template class name.

3 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. For 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
        }
};
```

Box 61

There is a potentially serious problem with the rule that a name is a non-type name unless it has been explicitly specified to be the name of a type. In some contexts, there is no way of indicating that a name is a type name before using it. For example:

```
template <class Predicate> class unary_negate
        : unary_function<Predicate::argument_type, int>, // syntax error!
          restrictor<Predicate>
{
        // ...
};
```

There are two alternative solutions: (1) Allow the compiler to assume that a qualified name is the name of a type in ‘‘such contexts,’’ thus making the example above well-formed as written. (2) Introduce a keyword to allow the programmer to state that a name is a type name as part of using that name. For example:

```
template <class Predicate> class unary_negate
        : unary_function<typename Predicate::argument_type, int>,
          restrictor<Predicate>
{
        // ...
};
```

Solution (2) would replace the *type-name-declaration* construct.

4 Three kinds of names can be used within a template definition:

— The name of the template itself, the names of the *template-parameter*s (14.6), and names declared within the template itself.

— Names from the scope of the template definition.

— Names dependent on a *template-argument* (14.7) from the scope of a template instantiation.

5 For example:

```
#include<iostream.h>

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';
        }
        // ...
};
```

When looking for the declaration of a name used in a template definition the usual lookup rules (9.3) are first applied. Thus, 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.h`. 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, the `operator<<` needed to print `p[i]` cannot be known until it is known what type `T` is (14.2.3).

14.2.1 Locally declared names [temp.local]

1 Within the scope of a template or a specialization of a template the name of the template is equivalent to the name of the template qualified by the *template-parameter*. Thus, 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. For 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>
};
```

See 14.6 for the scope of *template-parameter*s.

2 A template *type-parameter* can be used in an *elaborated-type-specifier*. For 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)
        // ...
}
```

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. For 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
```

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. For 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)
```

In this, a template definition behaves exactly like other definitions. For 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)
```

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). For 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 the actual template type were missing from the program. Examples of 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 (14.9.2). For example: `f(T)`, `f(Vector<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` is a `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
}
```

Box 62

It has been suggested that a full list of cases would be a better definition than the general rule we decided on in San Jose. I strongly prefer a general rule, but we should be open to clarifications if people feel the need for them.

3 This ill-formed template instantiation uses a function that does not depend on a *template-argument*s:

```
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 can hide the name of a *template-parameter*. For example:

```
struct A {
        struct B { /* ... */ };
        int a;
};

template<class B, class a> struct X : A {
        B b;  // A's B
        a b;  // error: A's a isn't a type name
};
```

However, a name from a template argument cannot hide a name from the template or its enclosing scopes. For 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<A> ya;
```

The members `A::B` and `A::a` of the template argument A do not affect the binding of names in `Y<A>`.

6 A name of a member can hide the name of a *template-parameter*. For 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
}
```

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. However, such declarations must match declarations in scope at the point of their declaration or instantiation. For example:

```
void f();
// no Y, Z, or g here

template<class T> class X {
        friend class Y; // error: No Y in scope
        class Z * p;    // error: No Z in scope
        friend T operator+(const T&, const T&); // checking delayed
        friend void f(); // ok
        friend void f(T); // checking delayed
};

class C {
        friend C operator+(const C&, const C&);
};
void f(C);

class D { };
```

```
        void g()
        {
                X<C> c;  // ok: operator+(const C&, const C&) and f(C) in scope
                X<D> d;  // error: no operator+(const D&, const D&) or f(D)
        }
```

Box 63

In #94-0116/N503 Barton and Nackman argue strongly that some form of name injection is essential so that this rule should be relaxed. See also Spicer Issue 2.23.

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 The act of generating a class, function, or static data member from a template is commonly referred to as template instantiation.

14.3.1 Making template definitions available [temp.avail]

1 A template can be instantiated for a set of template arguments only if the definition of the template is available.

Box 64

The definition of ‘‘available’’ has yet to be determined. Two main lines of approach are being considered: (1) Make translation units containing template definitions available to the compilation system in some extra-linguistic way; see §4 of #94-0024/N0413. (2) Make template definitions available through some form of explicit template directives.

2 Here is the template directive variant proposed at the Waterloo meeting:

3 A template that has been used in a way that requires a specialization of its definition may be explicitly specialized (14.5) or explicitly instantiated (14.4) within the current translation unit, otherwise the specialization will be implicitly generated if the definition has either been previously made available through a template-directive or has been previously defined in the current translation unit, else the template shall be explicitly instantiated within the program.

4 The syntax for a template directive is:

template-directive:
    `template` *string-literal* `;`

A template-directive nominates a translation unit containing definitions of templates that have been declared in the current translation unit. There shall be an implementation- defined mapping between the string-literal and an external source file name that is used to produce the nominated translation unit. The nominated translation unit is produced using the same environment as for the current translation unit. Names declared in the current translation unit are not visible within the nominated translation unit. Names declared within the nominated translation unit do not become visible within the current translation unit. After a template-directive has been processed, the template definitions within the nominated translation unit are available for use in instantiation of the declared templates if required. Definitions of objects or functions with external linkage within the nominated translation unit shall be well-formed but shall otherwise be treated as declarations only (i.e., no storage will be allocated and no code will be generated). A template-directive shall not appear in class scope or local scope. The treatment of definitions of objects or functions with internal linkage will be revisited when linkage issues are resolved.

5 In order to explicitly instantiate the specialization, the definition shall either be made available through a template-directive or shall have been previously defined in the current translation unit. For explicit instantiation of a class, the definition of all members shall be made available.

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. 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)
        }
}
```

Box 65
This was discussed, but not voted on in Waterloo. The previous version was plain wrong.

2 Each compilation unit in which the definition of a template is used has a point of instantiation for the template. If this causes names used in the template definition to bind to different names in different compilations, the one-definition rule has been violated and any use of the template is ill-formed. Such violation does not require a diagnostic.

3 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. For 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.

4 If a virtual function is instantiated, its point of instantiation is immediately following the point of instantiation for its class.

Box 66

There has been no vote on the point of instantiation for a virtual function.

5 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.

```
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");
        // ...
}
```

This allows instantiation of a used template to be done before instantiation of its user.

6 The rules specifying where definitions are required (above) prevent mutually dependent definitions from causing mutually recursive instantiations.

7 Implicitly generated template classes, functions, and static data members are placed in the namespace where the template was defined. For example, a call of `lt.get()` from `Map<char*,int>::get()` would place `List<int>::get()` in `N` rather than in the global space.

Box 67

There has been no vote on the point of instantiation for an indirectly generated template specialization. This rule gets interesting only if template instantiation can cause name injection (14.2.4).

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. For 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>*
}
```

9 If an instantiation of a class template is required and the template is declared but not defined, the program is ill-formed. For example:

```
template<class T> class X;

A<char> ch; // error: definition of X required
```

10 Recursive instantiation is possible. For example:

```
template<int i> int fac() { return i>1 ? i*fac<i-1>() : 1; }

int fac<0>() { return 1; }

int f()
{
        return fac<17>();
}
```

11 There shall be an implementation quantity that specifies the limit on the depth of recursive instantiations.

Box 68

Put limit with deafult 17 in Annex B.

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. For 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 ...
};
```

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 must be in the namespace in which the template was defined. For 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<char*> /* ... */ }; // ok: specialization
                               //     in same namespace
```

15 A member function of an explicitly specialized class cannot be implicitly generated from the general template. Instead, the member function must itself be explicitly specialized. For 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() { /* ... */ };
```

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 must be provided in some compilation unit.

14.3.3 Instantiation of `operator->`

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. For 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
}
```

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` *inst* `;`

inst:
    class-key template-id
    type-specifier-seq template-id `(` *parameter-declaration-clause* `)`

Box 69

Syntax WG: please check this grammar. It ought to allow any declaration that is not a definition of a class or function with a *template-id* as the name being declared.

For example:

```
template class vector<char>;

template void sort<char>(vector<char>&);
```

3 A declaration of the template must be in scope and the definition of the template must be available at the point of explicit instantiation.

Box 70

Exactly what ‘‘must be available’’ means depends on the compilation model we adopt for templates.

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. For example:

```
// instantiate sort(vector<int>&):
// deduce template-argument:
template void sort<>(vector<int>&);
```

5 The explicit instantiation of a class implies the instantiation of all of its members not previously explicitly specialized in the compilation unit containing the explicit instantiation.

14.5 Template specialization [temp.spec]

1 A specialized template function, template class, or static member of a template can be declared by a declaration where the declared name is a *template-id*, that is:

specialization:
  declaration

Box 71

Syntax WG: please check this grammar. Should the fact that a *template-id* must be the name declared be made explicit in the grammar?

For example:

```
template<class T> class stream;

class stream<char> { /* ... */ };

template<class T> void sort(vector<T>& v) { /* ... */ }

void sort<char*>(vector<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 `vector<char*>`; other `vector` types will be sorted by functions generated from the template.

2 A declaration of the template being specialized must be in scope at the point of declaration of a specialization. For example:

```
class X<int> { /* ... */ }; // error: X not a template

template<class T> class X { /* ... */ };

class X<char*> { /* ... */ }; // fine: X is a template
```

3 If a template is explicitly specialized then that specialization must be declared before the first use of that specialization in every translation unit in which it is used. For example:

```
template<class T> void sort(vector<T>& v) { /* ... */ }

void f(vector<String>& v)
{
        sort(v); // use general template
                 // sort(vector<T>&), T is String
}

void sort<String>(vector<String>& v); // error: specialize after use
void sort<>(vector<char*>& v); // fine sort<char*> not yet used
```

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 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 (14.9.4).

14.6 Template parameters [temp.param]

1 The syntax for *template-parameter*s is:

template-parameter:
  type-parameter
  parameter-declaration

type-parameter:
  `class` *identifier*$_{opt}$
  `class` *identifier*$_{opt}$ `=` *type-id*
  `typedef` *identifier*$_{opt}$
  `typedef` *identifier*$_{opt}$ `=` *type-name*
  `template <` *template-parameter-list* `> class` *identifier*$_{opt}$
  `template <` *template-parameter-list* `> class` *identifier*$_{opt}$ `=` *template-name*

For example:

```
template<class T> myarray { /* ... */ };

template<class K, class V, template<class T> class C = myarray>
class Map {
        C<K> key;
        C<V> value;
        // ...
};
```

Box 72

This grammar leaves out namespace *template-parameter*s. See §2 of ANSI X3J16/94-0026, ISO WG21/N0413.

Box 73

Should this grammar be modified to accept `struct` as well as `class` for template *template-parameter*s?

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. For 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
```

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. For 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 `typedef` 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 (14.7). 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. For 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
}
```

6 A non-type *template-parameter* cannot be of floating type because only integral constant expressions (5.19) are considered as *template-arguments* for non-type template parameters and standard conversions are not applied to *template-arguments*. For example:

```
template<double d> class X;    // error
template<double* pd> class X;  // ok
template<double& rd> class X;  // ok
```

7 A default *template-argument* is a type or a value specified after = in a *template-parameter*. A default *template-argument* can be specified in a template declaration or a template definition. A function template shall not have default *template-arguments*. The set of default *template-arguments* available for use with a template declaration or definition is obtained by merging the default arguments from the definition (if in scope) and all declarations in scope in the same way default function arguments are (8.3.6). For example:

```
template<class T1, class T2 = int> class A;
template<class T1 = int, class T2> class A;
```

is equivalent to

```
template<class T1 = int, class T2 = int> class A;
```

If a *template-parameter* has a default argument all subsequent *template-parameter*s must have a default argument supplied in the same or previous declarations of the template. For example:

```
template<class T1 = int, class T2> class B; // error
```

A *template-parameter* shall not be given default arguments by two different declarations in the same scope.

```
template<class T = int> class X;
template<class T = int> class X { /*... */ }; // error
```

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. For example:

```
template<class T, T* p, class U = T> class X { /* ... */ };
template<class T> void f(T* p = new T);
```

A *template-parameter* cannot be used in preceding *template-parameters* or their default arguments.

8 A *template-parameter* can be used in the specification of base classes. For example:

```
template<class T> class X : public vector<T> { /* ... */ };
template<class T> class Y : public T { /* ... */ };
```

Note that 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.

14.7 Template arguments [temp.arg]

1 The types of the *template-argument*s specified in a *template-id* must match the types specified for the template in its *template-parameter-list*. For example, `vector`s as defined in 14 can be used like this:

```
vector<int> v1(20);
vector<complex> v2(30);

typedef vector<complex> cvec;   // make cvec a synonym
                                // for vector<complex>
cvec v3(40);  // v2 and v3 are of the same type

v1[3] = 7;
v2[3] = v3.elem(4) = complex(7,8);
```

2 A non-type non-reference *template-argument* must be a *constant-expression*, the address of an object or a non-overloaded function with external linkage, or a non-overloaded pointer to member. The address of an object or function must be expressed as `&f`, plain `f`, or `&X::f` where `f` is the function or object name. In the case of `&X::f`, `X` must be a (possibly qualified) name of a class and `f` the name of a static member of `X`. A pointer to member must be expressed as `&X::m` where `X` is a (possibly qualified) name of a class and `m` is the member name. 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. For 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
```

3 Similarly, addresses of array elements and non-static class members are not acceptable as `template-arguments`. For 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
X<&S::s,p> x6; // ok: address of static member
```

4 Nor is a local type or an type with no linkage name an acceptable *template-argument*. For example:

```
void f()
{
        struct S { /* ... */ };

        X<S,p> x3; // error: local type used as template-argument
}
```

5 Similarly, a reference *template-parameter* cannot be be bound to a temporary:

```
template<const int& CRI) struct B { /* ... */ };

B<1> b2; // error: temporary required for template argument

int c = 1;
B<c> b1; // ok
```

6 A template has no special access rights to its *template-argument* types. However, often a template doesn't need any. For example:

```
class Y {
private:
        struct S { /* ... */ };
        X<S> x; // most operations by X on S do not lead to errors
};

X<Y::S> y; // most operations by X on Y::S leads to errors
```

The template `X` can use `Y::S` without violating any access rules as long as it uses only the access through a *template-argument* that does not explicitly mention `Y`.

7 An argument for a *template-parameter* of reference type must be an object or function with external linkage, or a static class member. A temporary object is not an acceptable argument to a *template-parameter* of reference type.

8 When default *template-arguments* are used, a *template-argument* list can be empty. In that case the empty `<>` brackets must still be used. For example:

```
template<class T = char> class String;
String<>* p; // ok: String<char>
String* q;   // syntax error
```

The notion of ''array type decay'' does not apply to *template-parameter*s. For example:

```
template<int a[5]> struct S { /* ... */ };
int v[5];
int* p = v;
S<v> x; // fine
S<p> y; // error
```

14.8 Type equivalence [temp.type]

1 Two *template-id*s refer to the same class or function if their *template* names are identical and their arguments have identical values. For 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`.

14.9 Function templates [temp.fct]

1 A function template specifies how individual functions can be constructed. A family of sort functions, for example, might be declared like this:

```
template<class T> void sort(vector<T>);
```

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.

14.9.1 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. For example:

```
void f(vector<complex>& cv, vector<int>& ci)
{
    sort<complex>(cv);  // sort(vector<complex>)
    sort<int>(ci);      // sort(vector<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)
}
```

Implicit conversions (4) are accepted for a function argument for which the parameter has been fixed by explicit specification of a *template-argument*. For example:

```
template<class T> void f(T);

class complex {
        // ...
        complex(double);
};

void g()
{
        f<complex>(1); // ok, means f<complex>((complex(1))
}
```

14.9.2 Template argument deduction [temp.deduct]

1 Template arguments that can be deduced from the function arguments of a call need not be explicitly specified. For example,

```
void f(vector<complex>& cv, vector<int>& ci)
{
    sort(cv);   // call sort(vector<complex>)
    sort(ci);   // call sort(vector<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)
}
```

2 A template type argument `T` or a template non-type argument `i` can be deduced from a function argument composed from these elements:

`T`
cv-list `T`
`T*`
`T&`
`T[`*integer-constant*`]`
class-template-name`<T>`
type`(*)(T)`
type `T::*`
`T(*)()`
type`[i]`
class-template-name`<i>`

where the `T` in argument list form

type `(*)(T)`

includes argument lists with more than one argument where at least one argument contains a `T`. Also, these forms can be used in the same way as `T` is for further composition of types. For example,

`X<int>(*)(v[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 `v[6]`.

3 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. For 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
}
```

Template arguments shall not be deduced from function arguments involving other constructs.

4

Box 74
Can a template *template-parameter* be deduced? and if so how? Spicer issue 3.19.

5 Template arguments of an explicit instantiation or explicit specialization are deduced (14.4, 14.5) according to these rules specified for deducing function arguments.

6 Note that 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][10]);

void g(int v[10][10])
{
        f1(v);     // ok: i deduced to be 10
        f1<10>(v); // ok
        f2(v);     // error: cannot deduce template-argument i
        f2<10>(v); // ok
}
```

7 Nontype parameters shall not be used in expressions in the function declaration. The type of the function *template-parameter* must match the type of the *template-argument* exactly. For 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
```

8 Every *template-parameter* specified in the *template-parameter-list* must be either explicitly specified or deduced from a function argument. If function *template-argument*s are specified in a call they are specified in declaration order. Trailing arguments can be left out of a list of explicit *template-argument*s. For 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

}
```

9 A *template-parameter* cannot be deduced from a default function argument. For 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)
}
```

10 If a template parameter can be deduced from more than one function argument the deduced template parameter must the same in each case. For example:

```
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
}
```

14.9.3 Overload resolution [temp.over]

1 A template function can be overloaded either by (other) functions of its name or by (other) template functions of that same name. Overloading resolution for template functions and other functions of the same name is done in the following three steps:

1) Look for an exact match (13.2) on functions; if found, call it.

2) Look for a function template from which a function that can be called with an exact match can be generated; if found, call it.

3) Look for match with conversions. For arguments to ordinary functions and for arguments to a template function that corresponds to parameters whose type does not depend on a deduced *template-parameter*, the ordinary best match rules apply. For template functions, only the following conversions listed below applies. After the best matches are found for individual arguments, the intersection rule (_over.match.args_) is used to look for a best match; if found, call it.

Box 75

Rephrase to match Clause 13.

2 For arguments that correspond to parameters whose type depends on a deduced template parameter, the following conversions are allowed:

— For a parameter of the form `B<params>`, where `params` is a template parameter list containing one or more deduced parameters, an argument of type ‘‘class derived from `B<params>`’’ can be converted to `B<params>`. Additionally, for a parameter of the form `B<params>*`, an argument of type ‘‘pointer to class derived from `B<params>`’’ can be converted to `B<params>*`. Similarly for references. Also, for a parameter of the form `T` an argument of type ‘‘`T B::*` where B is a base of `D<params>`’’ can be converted to `T D<params>::*`.

— A pointer (reference) can be converted to a more qualified pointer (reference) type, according to the rules in 4.10 (_conv.ref_).

— ‘‘array of `T`’’ to ‘‘pointer to `T`.’’

— ‘‘function ...’’ to ‘‘pointer to function to ... .’’

Box 76

The pointer to member case added editorially.

3 If no match is found the call is ill-formed. In each case, if there is more than one alternative in the first step that finds a match, the call is ambiguous and is ill-formed.

4 A match on a template (step (2)) implies that a specific template function with parameters that exactly match the types of the arguments will be generated (14.3). Not even trivial conversions (13.2) will be applied in this case.

Box 77

This maybe too strict. See the proposal for a more general overloaded mechanism in N0407/94–0020 (issue 3.9).

5 The same process is used for type matching for pointers to functions (13.3) and pointers to members.

6 Here is an 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)
}
```

7 For example, 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`.

8 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 )
}
```

9 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))
}
```

10 The template definition is needed to generate specializations of a template. However, only a function template declaration is needed to call a specialization. For example,

```
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 the declaration of `f`, and the program will be ill-formed unless a definition of `f` is present in some translations unit.

11 In case a call has explicitly qualified *template-argument*s and requires overload resolution, the explicit qualification is used first to determine the set of overloaded functions to be considered and overload resolution then takes place for the remaining arguments. For example:

```
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*)
}
```

14.9.4 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. For 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; }
```

Such an overloaded function is a specialization but not an explicit specialization. The declaration simply guides the overload resolution. 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);
```

2 Defining a function with the same type as a template specialization that is called is ill-formed. For 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. For example:

```
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)
}
```

14.10 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. For example,

```
template<class T> class vector {
    T* v;
    int sz;
public:
    vector(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& vector<T>::operator[](int i)
{
    if (i<0 || sz<=i) error("vector: range error");
    return v[i];
}
```

2 The *template-argument* for `vector<T>::operator[]()` will be determined by the vector to which the subscripting operation is applied.

```
vector<int> v1(20);
vector<complex> v2(30);

v1[3] = 7;              // vector<int>::operator[]()
v2[3] = complex(7,8);   // vector<complex>::operator[]()
```

14.11 Friends [temp.friend]

1 A friend function of a template can be a template function or a non-template function. For 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) { /* ... */ }
```

2 A friend template shall not be defined within a class. For example:

```
class A {
        friend template<class T> B;    // ok
        friend template<class T> f(T); // ok

        friend template<class T> BB { /* ... /* }; // error
        friend template<class T> ff(T){ /* ... /* } // error
};
```

14.12 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. For 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. For 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);
            f(&b);
        }
```

Here `f(int*)` has a static member `s` of type `int` and `f(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*

handler-seq:
  handler handler-seq$_{opt}$

handler:
  `catch (` *exception-declaration* `)` *compound-statement*

exception-declaration:
  type-specifier-seq declarator
  type-specifier-seq abstract-declarator
  type-specifier-seq
  `...`

throw-expression:
  `throw` *assignment-expression*$_{\texttt{opt}}$

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*."

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. For 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*.

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. For 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
}
```

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 The operand of a `throw` shall be of a type with no ambiguous base classes. That is, it shall be possible to convert the value thrown unambiguously to each of its base classes.[60)]

4 A *throw-expression* initializes a temporary object of the static type of the operand of `throw` and uses that temporary to initialize the appropriately-typed variable named in the handler. If the static type of the expression thrown is a class or a pointer or reference to a class, there shall be an unambiguous conversion from that class type to each of its accessible base classes. Except for that restriction and forthe restrictions on type matching mentioned in 15.3 and the use of a temporary variable, the operand of `throw` is treated exactly as a function argument in a call (5.2.2) or the operand of a `return` statement.

5 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 may be initialized directly with the argument of the throw expression.

6 A *throw-expression* with no operand rethrows the exception being handled without copying it. For example, code that must be executed because of an exception yet cannot completely handle the exception can be written like this:

60) If the value thrown has no base classes or is not of class type, this condition is vacuously satisfied.

```
try {
    // ...
}
catch (...) {  // catch all exceptions

    // respond (partially) to exception

    throw;     // pass the exception to some
               // other handler
}
```

7 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.

8 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 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 an accessible (4.10) base class of `E` at the throw point, 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) at the throw point.

3 For 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`.

4 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.

5 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 must be the last handler for its *try-block*.

6 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*. If no matching handler is found in a program, the function `terminate()` (15.5.1) is called.

7 An exception is considered handled upon entry to a handler. The stack will have been unwound at that point.

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.

Box 78
Should it be possible to use more general types than *type-id*s in *exception-specification*s?

exception-specification:
    `throw (` *type-id-list*$_{opt}$ `)`

type-id-list:
    type-id
    type-id-list `,` *type-id*

An *exception-specification* shall appear only in a context that causes it to apply directly to a declaration or definition of a function or member function. For example:

```
extern void f() throw(int);           // OK
extern void (*fp) throw (int);        // ill-formed
extern void g(void f() throw(int));   // ill-formed
```

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 must have an *exception-specification* at least as restrictive as that in the base class. For example:

```
struct B {
    virtual void f() throw (int, double);
    virtual void g();
};

struct D: B {
    void f();                       // ill-formed
    void g() throw (int);           // OK
};
```

The declaration of `D::f` is ill-formed because it allows all exceptions, whereas `B::f` allows only `int` and `double`.

2 Types may not be defined in *exception-specification*s.

3 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 classes with ambiguous base classes, even though throwing objects of such classes is ill-formed (15.1). An exception specification can also include identifiers that represent incomplete types.[61)]

4 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 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 derived from `Y*`.

Box 79

This still needs to deal with `const` and `volatile`

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. For 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()
}
```

5 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. For 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.

61) This makes sense, for example, in declaring a function that is defined elsewhere. It probably does not make sense in a function definition, because the type would have to be completed before an object of that type could be constructed and thrown.

6 A function with no *exception-specification* allows all exceptions. A function with an empty *exception-specification*, `throw()`, does not allow any exceptions.

7 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. These functions are declared in `<exception>` and `<exception.ns>` (_lib.header.exception_).

15.5.1 The `terminate()` function [except.terminate]

1 Occasionally, exception handling must be abandoned for less subtle error handling techniques. For example,

— when a exception handling mechanism, after completing evaluation of the object to be thrown, calls a user function that exits via an uncaught exception,[62)]

— when the exception handling mechanism cannot find a handler for a thrown exception,

— when the exception handling mechanism finds the stack corrupted, or

— when a destructor called during stack unwinding caused by an exception tries to exit using an exception.

2 In such cases,

```
void terminate();
```

is called; `terminate()` calls the function given on the most recent call of `set_terminate()`:

```
typedef void(*PFV)();
PFV set_terminate(PFV);
```

3 The previous function given to `set_terminate()` will be the return value; this enables users to implement a stack strategy for using `terminate()`. The default function called by `terminate()` is `abort()`.

4 The function given as argument to `set_terminate`, if called, shall not return to its caller. It should either terminate execution by explicitly calling `exit()` or `abort()` or loop infinitely. The effect of such a function trying to return to its caller, either by executing a `return` statement or throwing an exception, is undefined.

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; `unexpected()` calls the function given on the most recent call of `set_unexpected()`:

```
typedef void(*PFV)();
PFV set_unexpected(PFV);
```

The previous function given to `set_unexpected()` will be the return value; this enables users to implement a stack strategy for using `unexpected()`. The default function called by `unexpected()` is `terminate()`. Since the default function called by `terminate()` is `abort()`, this leads to immediate and precise detection of the error.

[62)] 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 The `unexpected()` function shall not return, but it can throw (or re-throw) an exception. Handlers for this exception will be looked for starting at the call of the function whose *exception-specification* was violated. Thus an *exception-specification* does not guarantee that only the listed classes will be thrown. For example,

```
void  pass_through() { throw; }
void  f(PFV pf) throw()   // f claims to throw no exceptions
{
        (*pf)();          // but the argument function might
}
void  g(PFV pf)
{
        set_unexpected(&pass_through);
        f(pf);
}
```

After the call in `g()` to `set_unexpected()`, `f()` behaves as if it had no *exception-specification* at all.

15.6 Exceptions and access [except.access]

1 The parameter of a catch clause obeys the same access rules as a parameter of the function in which the catch clause occurs.

2 An object may 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.[63)]

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*

63) 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 other-wise 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;[64] 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.

[64] 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 <<<<<???>>>>>>, except that `int` and `unsigned int` act as if they have the same representation as, respectively, `long` and `unsigned long`. This includes interpreting character constants, which may involve converting escape sequences into execution character set members. Whether the numeric value for these character constants matches the value obtained when an identical character constant occurs in an expression (other than within a `#if` or `#elif` directive) is implementation-defined.[65] Also, whether a single-character character constant 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.[66]

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.

---

[65] 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)
```

[66] 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.[67] 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.

Box 80
Does this restriction still make sense for C++?

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 (see <<<<???>>>>).

7 The most common uses of `#include` preprocessing directives are as in the following:

```
#include <stdio.h>
#include "myprog.h"
```

8 This example illustrates 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
```

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.

[67] 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.

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.

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[68)] 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

[68)] Since, by macro-replacement time, all character constants 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.

constants: a \ character is inserted before each " and \ character of a character constant 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.

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 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 */
```

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 dependent.

`__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 defines the contents of the *Standard C++ 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 Standard C++ library contains components for: language support, predefined exceptions, strings, locales, bit sets, bit strings, dynamic arrays, complex numbers, and iostreams. The language support components are required by certain parts of the C++ language, such as memory allocation (5.3.4, 5.3.5) and exception processing (_except.intro_).

3 The strings and other containers provide commonly used data types not directly defined in the C++ language. The predefined exceptions provide support for uniform error reporting from the components in the library. Complex numbers extend support for numeric processing. The iostreams components are the primary mechanism for C++ program input/output.

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

5 This subclause describes the scope (_lib.scope_), references (_lib.references_), definitions (17.1), and method of description (17.2) for the library. Subclauses (17.3) and (18) through (27) specify the contents of the library, and library requirements and constraints on both well-formed C++ programs and conforming implementations.

17.1 Definitions [lib.definitions]

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

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

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

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

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

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

— **Observer function:** A class member function (9.4) accesses the state of an object of the class, but does not alter that state. Observer functions are specified as `const` member functions (9.4.1).

— **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 (3.4).

— **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 Standard C++ 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 [lib.description]

1 This subclause describes the conventions used throughout clause 17 to describe the Standard C++ library. It describes the structures of the normative subclauses 18 through 27 (17.2.1), conventions used to specify constraints on template arguments(17.2.2), conventions to describe types defined by an implementation (17.2.3), and other editorial conventions (17.2.4).

17.2.1 Structure of each subclause [lib.structure]

1 Subclause 17.3.1 provides a summary of the Standard C++ library's contents. Other Library clauses provide detailed specifications for each of the components in the library, as shown in Table 14:

Table 14—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:

— Summary

— Detailed specifications

— References to the Standard C library

3 The Summary provides a synopsis of the category, listing the headers specified in the subclause and the library entities provided in each header. The summary and the detailed specifications are presented in the order:

— Macros

— Values

— Types

— Classes

— Functions

— Objects

4 The detailed specifications each contain the following elements:[69)]

— Name and brief description

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

— Restrictions on template arguments, if any

— Decription of class invariants

— Description of function semantics

5 Descriptions of class member functions follow the order (as appropriate):[70)]

— Constructor(s) and destructor

— Copying & assignment functions

— Comparison functions

— Modifier functions

— Observer functions

— Operators and other non-member functions

6 Descriptions of function semantics contain the following elements (as appropriate):[71)]

— Requires, the preconditions for calling the function

— Effects, the actions performed by the function, and/or the postconditions 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

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

17.2.2 Constraints on template arguments [lib.template.constraints]

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

2 To ensure that the different components in a library work together, they must satisfy some basic requirements. Requirements should be as general as possible, so instead of saying ‘‘class X has to define a member function `operator++()`,’’ we say ‘‘for any object `x` of class `X`, `++x` is defined.’’ (It is unspecified whether the operator is a member or a global function.) Requirements are stated in terms of well-defined expressions, which define valid terms of the types that satisfy the requirements. For every set of requirements there is a table that specifies an initial set of the valid expressions and their semantics. Any generic algorithm that uses the requirements has to be written in terms of the valid expressions for its formal type parameters.

69) The form of these specifications was designed to follow the conventions established by existing C++ library vendors.
70) 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.
71) 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.

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

4 In some cases we present the semantic requirements using 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 (although in some cases the code given is unambiguously the optimum implementation).

17.2.2.1 Traits [lib.trait.types]

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

Box 81

At the Kitchener meeting, the Library WG decided to change the names used from ‘‘`baggage`’’ to ‘‘`traits`’’. However, Tom Plum objected to doing this without a formal proposal and vote. These names are therefore unresolved, and subject to change.

17.2.2.2 Iterator types [lib.iterator.types]

1 Iterators are a generalization of pointers that allow a programmer to work with different data structures (containers) in a uniform manner. To be able to construct template algorithms that work correctly and efficiently on different types of data structures, we need to formalize not just the interfaces but also the semantics and complexity assumptions of iterators. Iterators are objects that have `operator*` returning a value of some class or built-in type `T` called a *value type* of the iterator. For every iterator type `X` for which equality is defined, there is a corresponding signed integral type called the *distance type* of the iterator.

2 Since iterators are a generalization of pointers, their semantics is a generalization of the semantics of pointers in C++. This assures that every template function that takes iterators works with regular pointers. Depending on the operations defined on them, there are five categories of iterators: *input iterators*, *output iterators*, *forward iterators*, *bidirectional iterators* and *random access iterators*, as shown in Table 15.

Table 15—Relations among iterator categories

Random access	→ **Bidirectional**	→ **Forward**	→ **Input**
			→ **Output**

3 Forward iterators satisfy all the requirements of the input and output iterators and can be used whenever either kind is specified. Bidirectional iterators satisfy all the requirements of the forward iterators and can be used whenever a forward iterator is specified. Random access iterators satisfy all the requirements of bidirectional iterators and can be used whenever a bidirectional iterator is specified. There is an additional attribute that forward, bidirectional and random access iterators might have, that is, they can be *mutable* or *constant* depending on whether the result of the `operator*` behaves as a reference or as a reference to a constant. Constant iterators do not satisfy the requirements for output iterators.

4 Just as a regular pointer to an array guarantees that there is a pointer value pointing past the last element of the array, so for any iterator type there is an iterator value that points past the last element of a corresponding container. These values are called *past-the-end* values. Values of the iterator for which the `operator*` is defined are called *dereferenceable.* The library never assumes that past-the-end values are dereferenceable. Iterators might also have singular values that are not associated with any container. For example, after the declaration of an uninitialized pointer `x` (as with `int* x;`), `x` should 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.

5 An iterator `j` is called *reachable* from an iterator `i` if there is a finite sequence of applications of `operator++` to `i` that makes `i == j`. If `j` is reachable from `i`, they refer to the same container.

6 Most of the library's algorithmic templates that operate on data structures have interfaces that use ranges. A *range* is a pair of iterators that designate the beginning and end of the computation. A range `[i, i)` is an empty range; in general, a range `[i, j)` refers to the elements in the data structure starting with the one pointed to by `i` and up to but not including the one pointed to by `j`. Range `[i, j)` is valid if and only if `j` is reachable from `i`. The result of the application of the algorithms in the library to invalid ranges is undefined.

7 All the categories of iterators require only those functions that are realizable for a given category in constant time (amortized). Therefore, requirement tables for the iterators do not have a complexity column.

8 In the following sections, we assume: `a` and `b` are values of `X`, `n` is a value of the distance type `Distance`, `u`, `tmp`, and `m` are identifiers, `r` is a value of `X&`, `t` is a value of value type `T`.

17.2.2.2.1 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 16:

Table 16—Input iterator requirements

expression	return type	operational semantics	assertion/note pre/post-condition
`X(a)`			`a == X(a)`. note: a destructor is assumed.

`X u(a);`
`X u = a;`
			post: `u == a`.

`a == b`	convertible to `bool`		`==` is an equivalence relation.
`a != b`	convertible to `bool`	`!(a == b)`	
`*a`	convertible to `T`		pre: `a` is dereferenceable. `a == b` implies `*a == *b`.
`++r`	`X&`		pre: `r` is dereferenceable. post: `r` is dereferenceable or `r` is past-the-end. `&r == &++r`.

`r++`	`X`	`{ X tmp = r;`
		`  ++r;`
		`  return tmp; }`	

2 NOTE: For input iterators, `a == b` does not imply `++a == ++b`. (Equality does not guarantee the substitution property or referential transparency.) Algorithms on input iterators should never attempt to pass through the same iterator twice. They should be *single pass* algorithms. *Value type T is not required to be an lvalue type.* These algorithms can be used with istreams as the source of the input data through the `istream_iterator` class.

17.2.2.2.2 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 17:

Table 17—Output iterator requirements

expression	return type	operational semantics	assertion/note pre/post-condition
`X(a)`			`a = t` is equivalent to `X(a) = t`. note: a destructor is assumed.
`X u(a);` `X u = a;`			
`*a = t`	result is not used		pre: `a` is dereferenceable.
`++r`	`X&`		pre: `r` is dereferenceable. post: `r` is dereferenceable or `r` is past-the-end. `&a == &++a`.
`r++`	`X`	`{ X tmp = r; ++r; return tmp; }`	

2 NOTE: The only valid use of an `operator*` is on the left side of the assignment statement. *Assignment through the same value of the iterator happens only once.* Algorithms on output iterators should never attempt to pass through the same iterator twice. They should be *single pass* algorithms. Equality and inequality might not be defined. Algorithms that take output iterators can be used with ostreams as the destination for placing data through the `ostream_iterator` class as well as with insert iterators and insert pointers.

17.2.2.2.3 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 18:

Table 18—Forward iterator requirements

expression	return type	operational semantics	assertion/note pre/post-condition
`X u;`			note: `u` might have a singular value. note: a destructor is assumed.
`X()`			note: `X()` might be singular.
`X(a)`			`a == X(a)`.

`X u(a);`
`X u = a;`
		`X u; u = a;`	post: `u == a`.

`a == b`	convertible to `bool`		`==` is an equivalence relation.
`a != b`	convertible to `bool`	`!(a == b)`	
`r = a`	X&		post: r == a.
`*a`	convertible to `T`		pre: `a` is dereferenceable. `a == b` implies `*a == *b`. If `X` is mutable, `*a = t` is valid.
`++r`	`X&`		pre: `r` is dereferenceable. post: `r` is dereferenceable or `r` is past-the-end. `r == s` and `r` is dereferenceable implies `++r == ++r`. `&a == &++a`.

`r++`	`X`	`{ X tmp = r;`
		`  ++r;`
		`  return tmp; }`	

2 NOTE: The condition that `a == b` implies `++a == ++b` (which is not true for input and output iterators) and the removal of the restrictions on the number of the assignments through the iterator (which applies to output iterators) allows the use of multi-pass one-directional algorithms with forward iterators.

17.2.2.2.4 Bidirectional iterators [lib.bidirectional.iterators]

1 A class or a built-in type `X` satisfies the requirements of a bidirectional iterator if to the table that specifies forward iterators we add the following lines, as shown in Table 19:

Table 19—Bidirectional iterator requirements (in addition to forward iterator)

expression	return type	operational semantics	assertion/note pre/post-condition
`--r`	X&		pre: there exists `s` such that `r == ++s`. post: `s` is dereferenceable. `--(++r) == r`. `--r == --r` implies `r == s`. `&r == &--r`.

`r--`	X	`{ X tmp = r;`
		`  --r;`
		`  return tmp; }`	

2 NOTE: Bidirectional iterators allow algorithms to move iterators backward as well as forward.

17.2.2.2.5 Random access iterators [lib.random.access.iterators]

1 A class or a built-in type `X` satisfies the requirements of a random access iterator if to the table that specifies bidirectional iterators we add the following lines, as shown in Table 20:

Table 20—Random access iterator requirements (in addition to bidirectional iterator)

expression	return type	operational semantics	assertion/note pre/post-condition
`r += n`	`X&`	`{ Distance m = n; if (m >= 0) while (m--) ++r; else while (m++) --r; return r; }`	
`a + n` `n + a`	`X`	`{ X tmp = a; return tmp += n; }`	`a + n == n + a`.
`r -= n`	`X&`	`return r += -n;`	
`a - n`	`X`	`{ X tmp = a; return tmp -= n; }`	
`b - a`	`Distance`	`{ X tmp = a; Distance m = 0; while (tmp != b) ++tmp, ++m; return m; }`	pre: there exists a value `n` of `Distance` such that `a + n == b`. `b == a + (b - a)`.
`a[n]`	convertible to `T`	`*(a + n)`	
`a < b`	convertible to `bool`	`b - a > 0`	`<` is a total ordering relation
`a > b`	convertible to bool	`b < a`	`>` is a total ordering relation opposite to `<`.
`a >= b`	convertible to bool	`!(a < b)`	
`a <= b`	convertible to bool	`!(a > b)`	

17.2.2.3 Allocator types [lib.allocator.types]

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

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

3 In the following Table 21, we assume `X` is an allocator class for objects of type `T`, `a` is a value of `X`, `n` is of type `X::size_type`, `p` is of type `X::pointer` which was obtained from `X`.

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

Table 21—Allocator requirements

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

17.2.2.4 Container types [lib.container.types]

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 22, we assume `X` is a container class containing objects of type `T`, `a` and `b` are values of `X`, `u` is an identifier and `r` is a value of `X&`.

Table 22—Container requirements

expression	return type	assertion/note pre/post-condition	complexity
`X::value_type`	`T`		compile time
`X::iterator`	iterator type pointing to `T`	any iterator category except output iterator.	compile time
`X::const_ iterator`	iterator type pointing to `const T`	any iterator category except output iterator.	compile time
`X::difference_type`	signed integral type	is identical to the distance type of `X::iterator` and `X::const_iterator`	compile time
`X:: size_type`	unsigned integral type	size_type can represent any non-negative value of `difference_type`	compile time
`X u;`		post: `u.size() == 0`.	constant
`X()`		`X().size() == 0`.	constant
`X(a)`		`a == X(a)`.	linear

`X u(a);`
`X u = a;`
		post: `u == a`.
		Equivalent to: `X u; u = a;`
			linear

`(&a)->`
` ~X()`
	result is not used	post: `a.size() == 0`.
		note: the destructor is applied to every element of `a`, all the memory is returned.
			linear

`a.begin()`	`iterator`; `const_iterator` for constant `a`		constant
`a.end()`	`iterator`; `const_iterator` for constant `a`		constant
`a == b`	convertible to `bool`	`==` is an equivalence relation. `a.size()==b.size() && equal(a.begin(), a.end(), b.begin())`	linear
`a != b`	convertible to `bool`	Equivalent to: `!(a == b)`	linear

expression	return type	operational semantics	assertion/note pre/post-condition	complexity

`r = a`	`X&`	`if (&r != &a) {`
		`(&r)->X::~X();`
		`new (&r) X(a);`
		`return r; }`
			post: `r == a`.	linear

`a.size()`	`size_type`	`a.end() - a.begin()`		constant
`a.max_ size()`	`size_type`		`size()` of the largest possible container.	constant
`a.empty()`	convertible to `bool`	`a.size() == 0`		constant
`a < b`	convertible to `bool`	`lexicographical_ compare (a.begin(), a.end(), b.begin(), b.end())`	pre: `<` is defined for values of `T`. `<` is a total ordering relation.	linear
`a > b`	convertible to `bool`	`b < a`		linear
`a <= b`	convertible to `bool`	`!(a > b)`		linear
`a >= b`	convertible to `bool`	`!(a < b)`		linear

Notes:

`equal` and `lexicographical_compare` are defined in Clause (_lib.algorithms_).

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.

17.2.2.4.1 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 23, `X` is a sequence class, `a` is value of `X`, `i` and `j` satisfy input iterator requirements, `[i, j)` is a valid range, `n` is a value of `X::size_type`, `p` is a valid iterator to `a`, `q`, `q1`, `q2` are valid dereferenceable iterators to `a`, `[q1, q2)` is a valid range, `t` is a value of `X::value_type`.

3 The complexities of the expressions are sequence dependent.

Table 23—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 24:

Table 24—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.

17.2.2.4.2 Associative containers [lib.associative.containers]

1 Associative containers provide an ability for fast retrieval of data based on keys. The library provides four basic kinds of associative containers: `set`, `multiset`, `map` and `multimap`.

2 All of them are parameterized on `Key` and an ordering relation `Compare` that induces a total ordering on elements of `Key`. In addition, `map` and `multimap` associate an arbitrary type `T` with the `Key`. The object of type `Compare` is called the *comparison object* of a container.

3 In this section when we talk about equality of keys we mean the equivalence relation imposed by the comparison and *not* the `operator==` on keys. That is, two keys `k1` and `k2` are considered to be equal if for the comparison object `comp`, `comp(k1, k2) == false && comp(k2, k1) == false`.

4 An associative container supports *unique keys* if it may contain at most one element for each key. Otherwise, it supports *equal keys*. `set` and `map` support unique keys. `multiset` and `multimap` support equal keys.

5 For `set` and `multiset` the value type is the same as the key type. For `map` and `multimap` it is equal to `pair<const Key, T>`.

6 `iterator` of an associative container is of the bidirectional iterator category.

7 In the following Table 25, `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 25—Associative container requirements (in addition to container)

expression	return type	assertion/note pre/post-condition	complexity
`X::key_type`	`Key`		compile time
`X::key_compare`	`Compare`	defaults to `less<key_type>`.	compile time
`X::value_compare`	a binary predicate type	is the same as `key_compare` for `set` and `multiset`; is an ordering relation on pairs induced by the first component (i.e. `Key`) for `map` and `multimap`.	compile time

`X(c)`
`X a(c);`
		constructs an empty container;
		uses `c` as a comparison object.
			constant

`X()`
`X a;`
		constructs an empty container;
		uses `Compare()` as a comparison object.
			constant

`X(i,j,c);`
`X a(i,j,c);`
		constructs an empty container and inserts elements from the range `[i, j)` into it; uses `c` as a comparison object.	`NlogN` in general (`N` is the distance from `i` to `j`); linear if `[i, j)` is sorted with `value_comp()`

`X(i, j)`
`X a(i, j);`
		same as above, but uses `Compare()` as a comparison object.	same as above

`a.key_comp()`	`X::key_compare`	returns the comparison object out of which `a` was constructed.	constant
`a.value_comp()`	`X::value_compare`	returns an object of `value_compare` constructed out of the comparison object.	constant
`a_uniq.insert(t)`	`pair<iterator, bool>`	inserts `t` if and only if there is no element in the container with key equal to the key of `t`. The `bool` component of the returned pair indicates whether the insertion takes place and the `iterator` component of the pair points to the element with key equal to the key of `t`.	logarithmic

expression	return type	assertion/note pre/post-condition	complexity
`a_eq.insert(t)`	`iterator`	inserts `t` and returns the iterator pointing to the newly inserted element.	logarithmic
`a.insert(p, t)`	`iterator`	inserts `t` if and only if there is no element with key equal to the key of `t` in containers with unique keys; always inserts `t` in containers with equal keys. always returns the iterator pointing to the element with key equal to the key of `t`. iterator `p` is a hint pointing to where the insert should start to search.	logarithmic in general, but amortized constant if `t` is inserted right after `p`.
`a.insert(i, j)`	result is not used	inserts the elements from the range `[i, j)` into the container.	`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.
```

17.2.3 Implementation types [lib.implementation.types]

1 Certain types defined in this clause are based on other types, but with added constraints.

17.2.3.1 Enumerated types [lib.enumerated.types]

1 Several types defined in this clause are *enumerated types*. Each enumerated type can be implemented as an enumeration or as a synonym for an enumeration. The enumerated type `enumerated` can be written:

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

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

17.2.3.2 Bitmask types [lib.bitmask.types]

1 Several types defined in this clause are *bitmask types*. Each bitmask type can be implemented as an enumerated type that overloads certain operators. The bitmask type `bitmask` can be written:

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

2 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.

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

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

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

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

17.2.3.3 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.[72)]

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

— A *character sequence* is an array object *A* that can be declared as *T A*`[`*N*`]`, where *T* is any of the types `char`, `unsigned char`, or `signed char`, 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.

17.2.3.3.1 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).[73)]

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.[74)]

17.2.3.3.2 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.[75)]

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

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

that can be declared as *T A*`[`*N*`]`, where *T* is type `wchar_t`, 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.

1 A *null-terminated wide-character string,* or *NTWCS,* is a wide-character sequence whose highest-addressed element with defined content has the value zero.[76)]

2 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.

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

73) Many of the objects manipulated by function signatures declared in `<cstring>` 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.

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

75) 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.

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

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

4 A *static* NTWCS is an NTWCS with static storage duration.[77)]

17.2.4 Other conventions [lib.res.and.conventions]

1 This subclause describes several editorial conventions used to describe the contents of the Standard C++ library. These conventions are for describing handler functions (17.2.4.1), class and class template member functions (17.2.4.2), private members (17.2.4.3), and convenient names (17.2.4.4).

Box 82
ISSUE: this subclause needs more discussion by the Library WG.

17.2.4.1 Handler functions [lib.global.pointers]

1 Certain handler functions are determined by the values stored in pointer objects within the Standard C++ library. Initially, these pointer objects store null pointers or designate functions defined in the Standard C++ library. Other functions, however, when executed at run time, permit the program to alter these stored values to point at functions defined in the program.

Box 83
ISSUE: this overspecification creates a problem for reentrancy

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

1 For the sake of exposition, this clause describes no copy constructors, assignment operators, or (non-virtual) destructors with the same apparent semantics as those that can be generated by default. It is unspecified whether the implementation provides explicit definitions for such member function signatures, or for virtual destructors that can be generated by default.

2 For the sake of exposition, this clause repeats in a derived class declarations for all the virtual member functions inherited from a base class. All such declarations are enclosed in a comment that ends with ***inherited***, as in:

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

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

Box 84
Editorial Proposal: Eliminate the specious ‘‘definitions’’ that say ‘‘Behaves the same as ...’’

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

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

[77)] A wide string literal, such as `L"abc"`, is a static NTWCS.

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

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

17.2.4.4 Convenient names [lib.unreserved.names]

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

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

3 For example:

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

```
FILE* file;
```

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

```
struct _Filet* _Fname;
```

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

17.3 Library-wide requirements [lib.requirements]

1 This subclause specifies requirements that apply to the entire Standard C++ library. Subclauses 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 Standard C++ 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 Standard C++ library provides definitions for the following types of entities:

— Macros

— Values

— Types

— Templates

— Classes

— Functions

— Objects

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

3 The Standard C++ library provides the following standard macros, as shown in Table 26. The header names (enclosed in < and >) indicate that the macro may be defined in more than one header. All such definitions shall be equivalent (3.2).

Table 26—Standard Macros

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

4 The Standard C++ library provides the following standard values, as shown in Table 27:

Table 27—Standard Values

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

5 The Standard C++ library provides the following standard types, as shown in Table 28:

Table 28—Standard Types

`capacity`	`mbstate_t`	`size_t <ctime>`	`wctrans_t`
`clock_t`	`new_handler`	`size_t <stddef>`	`wctype_t`
`div_t`	`ptrdiff_t <stddef>`	`streamoff`	`wint_t <cwchar>`
`FILE`	`ptrdiff_t<cstddef>`	`string`	`wint_t <cwctype>`
`fpos_t`	`sig_atomic_t`	`terminate_handler`	`wint_t <stddef>`
`fvoid_t`	`size_t <cstddef>`	`unexpected_handler`	`wstring`
`jmp_buf`	`size_t <cstdio>`	`va_list`	
`ldiv_t`	`size_t <cstring>`	`wchar_t`	

6 The Standard C++ library provides the following standard template classes, as shown in Table 29:

Table 29—Standard Template Classes

`allocator`	`locale::ctype_byname`
`back_insert_iterator`	`locale::moneypunct`
`basic_convbuf`	`locale::moneypunct_byname`
`basic_filebuf`	`locale::money_get`
`basic_ifstream`	`locale::money_put`
`basic_imanip`	`locale::msg`
`basic_ios`	`locale::msg_byname`
`basic_istream`	`locale::numpunct`
`basic_istringstream`	`locale::num_get`
`basic_istrstream`	`locale::num_put`
`basic_ofstream`	`locale::time_get`
`basic_omanip`	`locale::time_get_byname`
`basic_ostream`	`locale::time_put`
`basic_ostringstream`	`locale::time_put_byname`
`basic_ostrstream`	`map`
`basic_smanip`	`mask_array`
`basic_stdiobuf`	`multimap`
`basic_streambuf`	`multiset`
`basic_string`	`omanip`
`basic_stringbuf`	`ostreambuf_iterator`
`basic_strstreambuf`	`ostream_iterator`
`binary_negate`	`pointer_to_binary_function`
`binder1st`	`pointer_to_unary_function`
`binder2nd`	`priority_queue`
`bits`	`queue`
`deque`	`raw_storage_iterator`
`dyn_array`	`restrictor`
`front_insert_iterator`	`reverse_bidirectional_iterator`
`gslice_array`	`reverse_iterator`
`imanip`	`set`
`indirect_array`	`slice_array`
`insert_iterator`	`smanip`
`istreambuf_iterator`	`stack`
`istream_iterator`	`unary_negate`
`list`	`valarray`
`locale::codecvt`	`vector`
`locale::codecvt_byname`	`wimanip`
`locale::collate`	`womanip`
`locale::collate_byname`	`wsmanip`
`locale::ctype`	

7 The Standard C++ library provides the following standard template structures, as shown in Table 30:

Table 30—Standard Template Structs

`bidirectional_iterator`	`ios_char_baggage`	`negate`
`binary_function`	`ios_conv_baggage`	`not_equal_to`
`conv_baggage`	`ios_pos_baggage`	`pair`
`divides`	`less`	`plus`
`equal_to`	`less_equal`	`random_access_iterator`
`forward_iterator`	`logical_and`	`times`
`greater`	`logical_not`	`unary_function`
`greater_equal`	`logical_or`	
`input_iterator`	`minus`	
`ios_baggage`	`modulus`	

8 The Standard C++ library provides the following standard template operator functions, as shown in Table 31. 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 31—Standard Template Operators

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

9 The Standard C++ library provides the following standard template functions, as shown in Table 32:

Table 32—Standard Template Functions

abs (valarray)	iterator_category [5]
accumulate [2]	lexicographical_compare [2]
acos (valarray)	log (valarray)
adjacent_difference [2]	log10(valarray)
adjacent_find [2]	lower_bound [2]
advance	make_heap [2]
allocate	make_pair
asin (valarray)	max [2]
atan (valarray)	max_element [2]
atan2(valarray) [3]	merge [2]
back_inserter	min [2]
basic_dec	min_element [2]
basic_ends	mismatch [2]
basic_fixed	next_permutation [2]
basic_flush	not1
basic_hex	not2
basic_internal	nth_element [2]
basic_left	objconstruct
basic_noshowbase	objcopy
basic_noshowpoint	objdestroy
basic_noskipws	objmove
basic_nouppercase	partial_sort [2]
basic_nowshowpos	partial_sort_copy [2]
basic_oct	partial_sum [2]
basic_resetiosflags	partition
basic_right	pop_heap [2]
basic_scientific	pow (valarray) [3]
basic_setbase	prev_permutation [2]
basic_setfill	ptr_fun [2]
basic_setiosflags	push_heap [2]
basic_setprecision	random_shuffle [2]
basic_showbase	remove
basic_showpoint	remove_copy
basic_showpos	remove_copy_if
basic_skipws	remove_if
basic_uppercase	replace
basic_ws	replace_copy
binary_search [2]	replace_copy_if
bind1st	replace_if
bind2nd	reverse
construct	reverse_copy
copy	rotate
copy_backward	rotate_copy
cos (valarray)	search [2]
cosh (valarray)	set_difference [2]
count	set_intersection [2]
count_if	set_symmetric_difference [2]
deallocate	set_union [2]

`destroy [2]`	`sin (valarray)`
`distance`	`sinh (valarray)`
`distance_type (istreambuf_iterator)`	`sort [2]`
`distance_type [5]`	`sort_heap [2]`
`equal [2]`	`sqrt (valarray)`
`equal_range [2]`	`stable_partition`
`exp (valarray)`	`stable_sort [2]`
`fill`	`swap`
`fill_n`	`swap_ranges`
`find`	`tan (valarray)`
`find_if`	`tanh (valarray)`
`for_each`	`transform [2]`
`front_inserter`	`uninitialized_copy`
`generate`	`uninitialized_fill_n`
`generate_n`	`unique [2]`
`get_temporary_buffer`	`unique_copy [2]`
`includes [2]`	`unititialized_fill`
`inner_product [2]`	`upper_bound [2]`
`inplace_merge [2]`	`value_type (istreambuf_iterator)`
`inserter`	`value_type (ostreambuf_iterator)`
`iterator_category (istreambuf_iter)`	`value_type [5]`
`iterator_category (ostreambuf_iter)`	

10 The Standard C++ library provides the following standard classes, as shown in Table 33. Type names (enclosed in < and >) indicate that these are specific instances of templates.

Table 33—Standard Classes

`bad_alloc`	`float_complex`	`range_error`
`bad_cast`	`gslice`	`runtime_error`
`bad_typeid`	`ifstream`	`slice`
`basic_filebuf<char>`	`invalid_argument`	`stdiobuf`
`basic_filebuf<wchar_t>`	`ios`	`streambuf`
`basic_ifstream<char>`	`istdiostream`	`streampos`
`basic_ifstream<wchar_t>`	`istream`	`stringbuf`
`basic_ios<char>`	`istringstream`	`strstreambuf`
`basic_ios<wchar_t>`	`istrstream`	`type_info`
`basic_istream<char>`	`length_error`	`vector<bool,allocator>`
`basic_istream<wchar_t>`	`locale`	`wfilebuf`
`basic_ofstream<char>`	`locale::ctype<char>`	`wifstream`
`basic_ofstream<wchar_t>`	`logic_error`	`wistream`
`basic_ostream<char>`	`long_double_complex`	`wistringstream`
`basic_ostream<wchar_t>`	`ofstream`	`wofstream`
`basic_streambuf<char>`	`ostdiostream`	`wostream`
`basic_streambuf<wchar_t>`	`ostream`	`wostringstream`
`bitstring`	`ostringstream`	`wstreambuf`
`domain_error`	`ostrstream`	`wstringbuf`
`double_complex`	`out_of_range`	
`exception`	`overflow_error`	
`filebuf`	`ptr_dyn_array`	

11 The Standard C++ library provides the following standard structures, as shown in Table 34:

Table 34—Standard Structs

`bidirectional_iterator_tag`	`ios_pos_baggage<streampos>`
`conv_baggage<wchar_t>`	`ios_pos_baggage<wstreampos>`
`empty`	`lconv`
`forward_iterator_tag`	`locale::ctype_base`
`input_iterator_tag`	`output_iterator`
`ios_baggage<char>`	`output_iterator_tag`
`ios_baggage<wchar_t>`	`random_access_iterator_tag`
`ios_char_baggage<char>`	`tm <ctime>`
`ios_char_baggage<wchar_t>`	`tm <cwchar>`
`ios4baggage<wstreampos>`	

12 The Standard C++ library provides the following standard operator functions, as shown in Table 35:

Table 35—Standard Operator functions

operator delete	operator/ (double_complex) [3]
operator delete[]	operator/ (float_complex) [3]
operator new	operator/ (long_double_complex) [3]
operator new (void*)	operator/= (double_complex)
operator new[]	operator/= (float_complex)
operator new[] (void*)	operator/= (long_double_complex)
operator!= (basic_string) [5]	operator< (empty)
operator!= (double_complex) [3]	operator< (vector<bool,allocator>)
operator!= (float_complex) [3]	operator<< (basic_string)
operator!= (long_double_complex) [3]	operator<< (bits)
operator& (bits)	operator<< (bit_string)
operator& (bit_string)	operator<< (double_complex)
operator* (double_complex) [3]	operator<< (float_complex)
operator* (float_complex) [3]	operator<< (locale)
operator* (long_double_complex) [3]	operator<< (long_double_complex)
operator*= (double_complex)	operator== (basic_string) [5]
operator*= (float_complex)	operator== (double_complex) [3]
operator*= (long_double_complex)	operator== (empty)
operator+ (basic_string) [5]	operator== (float_complex) [3]
operator+ (bit_string)	operator== (long_double_complex) [3]
operator+ (double_complex) [4]	operator== (vector<bool,allocator>)
operator+ (dyn_array) [3]	operator>> (basic_string)
operator+ (float_complex) [4]	operator>> (bits)
operator+ (long_double_complex) [4]	operator>> (bit_string)
operator+ (ptr_dyn_array) [3]	operator>> (double_complex)
operator+= (double_complex)	operator>> (float_complex)
operator+= (float_complex)	operator>> (locale)
operator+= (long_double_complex)	operator>> (long_double_complex)
operator- (double_complex) [4]	operator^ (bits)
operator- (float_complex) [4]	operator^ (bit_string)
operator- (long_double_complex) [4]	operator| (bits)
operator-= (double_complex)	operator| (bit_string)
operator-= (float_complex)	
operator-= (long_double_complex)	

13 The Standard C++ library provides the following standard functions, as shown in Table 36:

Table 36—Standard Functions

abort	noshowpos
abs	noskipws
abs (double_complex)	nouppercase
abs (float_complex)	oct
abs (long_double_complex)	perror
acos	polar(double_complex)
arg (double_complex)	polar(float_complex)
arg (float_complex)	polar(long_double_complex)
arg (long_double_complex)	pow
asctime	pow (double_complex)
asin	pow (float_complex)
atan	pow (long_double_complex)
atan2	printf
atexit	putc
atof	puts
atoi	putwc
atol	putwchar
bsearch	qsort
btowc	raise
calloc	rand
ceil	real (double_complex)
clearerr	real (float_complex)
clock	real (long_double_complex)
conj (double_complex)	realloc
conj (float_complex)	remove
conj (long_double_complex)	rename
cos	resetiosflags
cos (double_complex)	rewind
cos (float_complex)	right
cos (long_double_complex)	scanf
cosh	scientific
cosh (double_complex)	setbase
cosh (float_complex)	setbuf
cosh (long_double_complex)	setfill
ctime	setiosflags
dec	setlocale
difftime	setprecision
div	setvbuf
endl	setw
ends	set_new_handler
exit	set_terminate
exp	set_unexpected
exp (double_complex)	showbase
exp (float_complex)	showpoint
exp (long_double_complex)	showpos
fabs	signal
fclose	sin
feof	sin (double_complex)

ferror	sin (float_complex)
fflush	sin (long_double_complex)
fgetc	sinh
fgetpos	sinh (double_complex)
fgets	sinh (float_complex)
fgetwc	sinh (long_double_complex)
fgetws	skipws
fixed	sprintf
floor	sqrt
flush	sqrt (double_complex)
fmod	sqrt (float_complex)
fopen	sqrt (long_double_complex)
fprintf	srand
fputc	sscanf
fputs	strcat
fputwc	strchr
fputws	strcmp
fread	strcoll
free	strcpy
freopen	strcspn
frexp	strerror
fscanf	strftime
fseek	strlen
fsetpos	strncat
ftell	strncmp
fwide	strncpy
fwprintf	stroul
fwrite	strpbrk
fwscanf	strrchr
getc	strspn
getchar	strstr
getenv	strtod
getline	strtok
gets	strtol
getwc	strxfrm
getwchar	swprintf
gmtime	swscanf
hex	system
imag (double_complex)	tan
imag (float_complex)	tanh
imag (long_double_complex)	terminate
internal	time
isalnum	tmpfile
isalpha	tmpfile
iscntrl	tmpnam
isdigit	tolower
isgraph	toupper
islower	towctrans
isprint	towlower
ispunct	towupper
isspace	unexpected
isupper	ungetc

iswalnum	ungetwc
iswalpha	uppercase
iswcntrl	vfwprintf
iswctype	vprintf
iswdigit	vprintf
iswgraph	vsprintf
iswlower	vswprintf
iswprint	vwprintf
iswpunct	wcrtomb
iswspace	wcscat
iswupper	wcschr
iswxdigit	wcscmp
isxdigit	wcscoll
iterator_category(output_iterator)	wcscpy
labs	wcscspn
ldexp	wcsftime
ldiv	wcslen
left	wcsncat
localeconv	wcsncmp
localtime	wcsncpy
log	wcspbrk
log (double_complex)	wcsrchr
log (float_complex)	wcsrtombs
log (long_double_complex)	wcsspn
log10	wcsstr
longjmp	wcstod
malloc	wcstok
mblen	wcstol
mbrlen	wcstombs
mbrtowc	wcstoul
mbsinit	wcsxfrm
mbsrtowcs	wctob
mbstowcs	wctomb
mbtowc	wctrans
memchr	wctype
memcmp	wmemchr
memcpy	wmemcmp
memmove	wmemcpy
memset	wmemmove
mktime	wmemset
modf	wprintf
norm (double_complex)	ws
norm (float_complex)	wscanf
norm (long_double_complex)	_double_complex
noshowbase	_float_complex
noshowpoint	

14 The Standard C++ library provides the following standard objects, as shown in Table 37:

Table 37—Standard Objects

`cerr`	`cin`	`clog`	`cout`	`errno`

17.3.1.2 Headers [lib.headers]

1 The elements of the Standard C++ library are declared or defined (as appropriate) in a *header*.[78)]

2 The Standard C++ library provides the following *C++ headers*, as shown in Table 38:

Box 85
Header names were no specified as part of the STL proposal.

Therefore, header names of the form `<stl` *`xxx`* `(TBD)>` are temporary placeholders, and subject to change pending Library WG discussion.

Table 38—C++ Library Headers

`<bits>`	`<ios>`	`<sstream>`	`<stl memory (TBD)>`
`<bitstring>`	`<iostream>`	`<stddef>`	`<stl numerics (TBD)>`
`<complex>`	`<istream>`	`<stdexcept>`	`<streambuf>`
`<cstream>`	`<locale>`	`<stl algorithms (TBD)>`	`<string>`
`<dynarray>`	`<memory>`	`<stl containers (TBD)>`	`<strstream>`
`<exception>`	`<new>`	`<stl core (TBD)>`	`<typeinfo>`
`<fstream>`	`<ostream>`	`<stl functional (TBD)>`	`<valarray>`
`<iomanip>`	`<ptrdynarray>`	`<stl iterators (TBD)>`	

3 The facilities of the Standard C Library are provided in 18 additional headers, as shown in Table 39:

Table 39—C++ Headers for C Library Facilities

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

Box 86
Header `<ciso646>` is not needed: see 2.8, Table 4

4 The contents of each header `c`*name* shall be the same as that of the corresponding header *name*`.h`, as specified in ISO C (Clause 4), or Amendment 1, (Clause 7), as appropriate. In this C++ library, however, the declarations and definitions are within namespace scope (3.3.4) of the namespace `std`. Subclause C.4 describes the effects of using the *name*`.h` (C header) form in a C++ program.

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

17.3.1.3 Freestanding implementations [lib.compliance]

1 Two kinds of implementations are defined: *hosted* and *freestanding* (1.6). For a hosted implementation, this International Standard defines the set of available headers.

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

Box 87
TBS. Requires a definition of library support for a freestanding environment.

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

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

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

17.3.2 Using the library [lib.using]

1 This subclause describes how a C++ program gains access to the facilities of the Standard C++ 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 Standard C++ library are defined in headers, whose contents are made available to a translation unit when it contains the appropriate `#include` preprocessing directive (16.2).

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

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

17.3.2.2 Linkage [lib.using.linkage]

1 Entities in the Standard C++ library have external linkage (3.4). Unless otherwise specified, objects and functions have the default `extern "C++"` linkage (7.5).

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

SEE ALSO: replacement functions (17.3.3.4), run-time changes (17.3.3.5).

17.3.3 Constraints on programs [lib.constraints]

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

17.3.3.1 Reserved names [lib.reserved.names]

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

— Macros

— Global names

— Names with external linkage

2 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.[79)]

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.[80)]

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. [81)]

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.

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 Standard C++ 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.

79) It is not permissible to remove a library macro definition by using the `#undef` directive.
80) The list of such reserved names includes `errno`, declared or defined in `<cerrno>`.
81) 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>`.

17.3.3.3 Derived classes [lib.derived.classes]

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

17.3.3.4 Replacement functions [lib.replacement.functions]

1 This clause describes the behavior of numerous functions defined by the Standard C++ 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 four dynamic memory allocation function signatures declared in `<new>` (3.6.3, 18):[82)]

— `operator new(size_t)`

— `operator new[](size_t)`

— `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, 3.5).

17.3.3.5 Handler functions [lib.handler.functions]

1 The Standard C++ 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`

3

17.3.3.6 Function arguments [lib.res.on.arguments]

1 Each of the following statements applies to all arguments to functions defined in the Standard C++ library, unless explicitly stated otherwise in this clause.

— 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 Standard C++ 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), access specifiers (17.3.4.6), class derivation (17.3.4.7), and exceptions (17.3.4.8).

Box 88
ISSUE - all of these have to be discussed by the Library WG

17.3.4.1 Headers [lib.res.on.headers]

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

2 None of the C headers includes any of the other headers, except that each C header includes its corresponding C++ header, as described above. None of the C++ headers includes any of the C headers. However, any of the C++ headers can include any of the other C++ headers, and must include a C++ header that contains any needed definition.[82)]

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

1 Only the names or global function signatures described in subclause _lib.TBD_ are reserved to the implementation.[83)]

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 A call to a global function signature described in this clause behaves the same as if the implementation declares no additional global function signatures.[84)]

17.3.4.4 Member functions [lib.member.functions]

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

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

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

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

2 A call to a member function signature described in this clause behaves the same as if the implementation declares no additional member function signatures.[85)]

17.3.4.5 Reentrancy [lib.reentrancy]

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

82) 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.

83) 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.*

84) 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.

85) 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.

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

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

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

17.3.4.7 Derived classes [lib.derivation]

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

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

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

2 In any case:

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

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

— Unless explicitly stated otherwise, types with distinct names in this clause are distinct types.[86)]

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

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

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

Box 89
ISSUE: aren't these two statements conveyed by exception specifications?

3 None of the functions defined in the Standard C++ library catch any exceptions, unless explicitly stated otherwise. A function can catch an exception not documented in this clause provided it rethrows the exception.

Box 90
ISSUE: this prevents implementations from using their own exceptions within the library

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

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 (18.1), characteristics of the predefined types (18.2), functions supporting start and termination of a C++ program (18.3), support for dynamic memory management (18.4), support for dynamic type identification (18.5), support for exception processing (18.6), and other runtime support (18.7).

18.1 Types [lib.support.types]

1 Common definitons.

2 Required by subclauses 5.3.3 and 12.5.

3 Headers:

— `<stddef>`

— `<cstddef>`

4 Table 40:

Table 40—Header `<stddef>` synopsis

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

5 Table 41:

Table 41—Header `<cstddef>` synopsis

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

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

18.1.1 Values [lib.stddef.values]

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

1 which is the largest representable value of type `size_t`.

18.1.2 Types [lib.stddef.types]

Box 91
This is not used anywhere. Remove it.

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

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

Box 92
This is redundant with `<cstddef>`. Remove it.

```
typedef T ptrdiff_t;
```

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

Box 93
This is redundant with `<cstddef>`. Remove it.

```
typedef T size_t;
```

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

Box 94
This is redundant with `<cwchar>`. Remove it.

```
typedef T wint_t;
```

4 The type `wint_t` is a synonym for *T*, the implementation-defined integral type, unchanged by integral promotions, that can hold any value of type `wchar_t` as well as at least one value that does not correspond to the code for any member of the extended character set.[87)]

```
typedef T capacity;
static const capacity default_size;
static const capacity reserve;
```

5 The type `capacity` is an enumerated type (indicated here as *T*), with the elements:

— `default_size`, as an argument value indicates that no reserve capacity argument is present in the argument list;

— `reserve`, as an argument value indicates that the preceding argument specifies a reserve capacity.

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

SEE ALSO: ISO C subclause 7.1.6.

18.2 Implementation properties [lib.support.limits]

1 Limits.

2 Required by subclause 3.7.1.

3 Headers:

— `<climits>`

— `<cfloat>`

4 Table 42:

Table 42—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`	

5 Table 43:

Table 43—Header `<cfloat>` synopsis

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

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

SEE ALSO: subclause 2.8, 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 Required by subclauses 3.5, 3.5.3.

2 Headers:

— `<cstdlib>` (partial)

3 Table 44:

Table 44—Header `<cstdlib>` synopsis

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

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

18.3.0.1 `atexit` [lib.atexit]

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

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

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

18.3.0.2 `exit` [lib.exit]

`exit(int` *`status`*`)`

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

— First, all functions *`f`* registered by calling `atexit(`*`f`*`)`, are called, in the reverse order of their registration.[88)]

— Next, all static objects are destroyed in the reverse order of their construction. (Automatic objects are not destroyed as a result of calling `exit`.)[89)]

— 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.[90)]

— 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.[91)]

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

SEE ALSO: ISO C subclause 7.10.4.

18.4 Dynamic memory management [lib.support.dynamic]

1 Required by subclauses 1.5, 3.6.3, 5.3.4, 5.3.5, 12.5.

2 Headers:

— `<new>`

88) A function is called for every time it is registered. The function signature `atexit(void (*)())`, is declared in `<cstdlib>`.
89) 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`.
90) Any C streams associated with `cin`, `cout`, etc (_lib.header.iostream_) are flushed and closed when static objects are destroyed in the previous phase. The function signature `tmpfile()` is declared in `<cstdio>`.
91) The macros `EXIT_FAILURE` and `EXIT_SUCCESS` are defined in `<cstdlib>`.

3 Table 45:

Table 45—Header `<new>` synopsis

Type	Name(s)
Type:	`new_handler`
Class:	`bad_alloc`
Operator functions:	
`operator delete`	`operator new (void*)`
`operator delete[]`	`operator new[]`
`operator new`	`operator new[] (void*)`
Function:	`set_new_handler`

4 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.

SEE ALSO: subclause 20.3.

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

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

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

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

1 The *allocation function* (3.6.3.1) called by a *new-expression* (5.3.4) to allocate *`size`* bytes of storage suitably aligned to represent any object of that size.

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

3 Required behavior: return a pointer to dynamically allocated storage (3.6.3).

4 Default behavior:

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

— Returns a pointer to the allocated storage if the attempt is successful. Otherwise, if the last argument to *`set_new_handler()`* was a null pointer, the result is implementation-defined.[92)]

— Otherwise, the function calls the current *`new_handler`* (_lib.new.handler_). If the called function returns, the loop repeats.

— The loop terminates when an attempt to allocate the requested storage is successful or when a called *`new_handler`* function does not return.

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

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

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

1 The *deallocation function* (3.6.3.2) called by a *delete-expression* to render the value of *ptr* invalid.

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

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

4 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(size_t)`.[93] For such a non-null value of *ptr*, reclaims storage allocated by the earlier call to the default `operator new(size_t)`.

5 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>`.

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

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

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

1 The *allocation function* (3.6.3.1) 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.[94]

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

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

4 Default behavior: returns `operator new(`*size*`)`.

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

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

1 The *deallocation function* (3.6.3.2) called by the array form of a *delete-expression* to render the value of *ptr* invalid.

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

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

4 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)`.[95] For such a non-null value of *ptr*, reclaims storage allocated by the earlier call

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

[94] 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.

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

to the default `operator new[](size_t)`.

5 It is unspecified under what conditions part or all of such reclaimed storage is allocated by a subsequent call to `operator new(size_t)` or any of `calloc`, `malloc`, or `realloc`, declared in `<cstdlib>`.

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

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

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

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

1 Returns *ptr*.

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

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

1 Returns *ptr*.

18.4.2 Storage allocation errors [lib.alloc.errors]

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

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

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

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

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

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

```
bad_alloc();
```

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

18.4.2.1.2 `bad_alloc` destructor [lib.des.bad.alloc]

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

1 Destroys an object of class `alloc`.

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

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

1 Returns an implementation-defined value.[96)]

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

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

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

2 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()`;

3 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);
```

1 Establishes the function designated by *`new_p`* as the current *`new_handler`*.

2 Returns the previous *`new_handler`*.

18.5 Type identification [lib.support.rtti]

1 Required by subclauses 5.2.6, 5.2.7.

2 Headers:

— `<typeinfo>`

3 Table 46:

Table 46—Header `<typeinfo>` synopsis

Type	Name(s)		
Classes:	`bad_cast`	`bad_typeid`	`typeid`

4 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.

18.5.1 Type information [lib.rtti]

18.5.1.1 Class `typeid` [lib.typeid]

Box 95
TO BE SPECIFIED

96) A possible return value is `&`*`alloc_msg`*.

18.5.1.2 Class `type_info` [lib.type.info]

```
    class type_info {
    public:
        virtual ~type_info();
        bool operator==(const type_info& rhs) const;
        bool operator!=(const type_info& rhs) const;
        bool before(const type_info& rhs) const;
        const char* name() const;
    private:
        type_info(const type_info& rhs);
        type_info& operator=(const type_info& rhs);
//      const char* name;      exposition only
//      T desc;                exposition only
    };
```

1 The class `type_info` describes type information generated within the program by the implementation. Objects of this class effectively store a pointer to a name for the type, and an encoded value suitable for comparing two types for equality or collating order. The names, encoding rule, and collating sequence for types are all unspecified and may differ between programs.

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

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

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

18.5.1.2.1 `type_info` destructor [lib.des.type.info]

```
    virtual ~type_info();
```

1 Destroys an object of type `type_info`.

18.5.1.2.2 `type_info::operator==` [lib.type.info::op==]

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

1 Compares the value stored in *`desc`* with *`rhs.desc`*.

2 Returns `true` if the two values represent the same type.

18.5.1.2.3 `type_info::operator!=` [lib.type.info::op!=]

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

1 Returns `true` if `!(*this ==` *`rhs`*`)`.

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

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

1 Compares the value stored in *`desc`* with *`rhs.desc`*.

2 Returns `true` if `*this` precedes *`rhs`* in the collation order.

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

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

1 Returns *name*.

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

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

1 Constructs an object of class `type_info` and initializes *name* to *rhs.name* and *desc* to *rhs.desc*.[97)]

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

2 Assigns *rhs.name* to *name* and *rhs.desc* to *desc*.

3 Returns `*this`.

18.5.2 Type identification errors [lib.rtti.errors]

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

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

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

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

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

1 Constructs an object of class `bad_cast`, initializing the base class with `logic_error(`*what_arg*.

18.5.2.1.2 `bad_cast` destructor [lib.des.bad.cast]

```
virtual ~bad_cast();
```

1 Destroys an object of class `bad_cast`.

18.5.2.1.3 `bad_cast::what` [lib.bad.cast::what]

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

1 Behaves the same as `exception::what()`.

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

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

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

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

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

```
bad_typeid();
```

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

18.5.2.2.2 `bad_typeid` destructor [lib.des.bad.type.id]

```
virtual ~bad_typeid();
```

1 Destroys an object of class `bad_typeid`.

18.6 Exception handling [lib.support.exception]

1 Required by subclause 15.5.

2 Headers:

— `<exception>`

3 Table 47:

Table 47—Header `<exception>` synopsis

Type	Name(s)	
Types:	`terminate_handler`	`unexpected_handler`
Functions:	`set_terminate` `terminate`	`set_unexpected` `unexpected`

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

18.6.1 Abnormal termination [lib.exception.terminate]

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

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

1 The type of a *handler function* to be called by `terminate()` when terminating exception processing.

2 Required behavior: a *`terminate_handler`* shall terminate execution of the program without returning to the caller.

3 Default behavior: the implementation's default *`terminate_handler`* calls `abort()`.

18.6.1.2 `set_terminate` [lib.set.terminate]

```
terminate_handler set_terminate(terminate_handler new_p);
```

1 Establishes the function designated by *new_p* as the current handler function for terminating exception processing.

2 *new_p* shall not be a null pointer.

3 Returns the previous *`terminate_handler`*.

18.6.1.3 `terminate` [lib.terminate]

```
void terminate();
```

1 Called by the implementation when exception handling must be abandoned for any of several reasons (15.5.1).

2 Calls the current *`terminate_handler`* handler function (_lib.terminate.handler_).

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

18.6.2.1 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.*

2 Required behavior: an *`unexpected_handler`* shall either throw an exception or terminate execution of the program without returning to the caller.

3 An *`unexpected_handler`* may perform any of the following:

— rethrow the exception;

— throw another exception;

— call `terminate()`;

— call either `abort()` or `exit()`;

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

18.6.2.2 `set_unexpected` [lib.set.unexpected]

```
unexpected_handler set_unexpected(unexpected_handler new_p);
```

1 Establishes the function designated by *`new_p`* as the current *`unexpected_handler`*.

2 *`new_p`* shall not be a null pointer.

3 Returns the previous *`unexpected_handler`*.

18.6.2.3 `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).

2 Calls the current *`unexpected_handler`* handler function (_lib.unexpected.handler_).

18.7 Other runtime support [lib.support.runtime]

1 Headers:

— `<cstdarg>` Variable arguments

— `<csetjmp>` Nonlocal jumps

— `<ctime>` system clock `clock(), time()`

— `<csignal>` Signal handling

— `<cstdlib>` Runtime environment `getenv(), system()`

2 Table 48:

Table 48—Header `<cstdarg>` synopsis

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

3 Table 49:

Table 49—Header `<csetjmp>` synopsis

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

4 Table 50:

Table 50—Header `<ctime>` synopsis

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

5 Table 51:

Table 51—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`		

6 Table 52:

Table 52—Header `<cstdlib>` synopsis

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

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

8 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 (19.1), documenting program assertions (19.2), and a global variable for error number codes (19.3).

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.

4 Headers:

— `<stdexcept>`

5 Table 53:

Table 53—Header `<stdexcept>` synopsis

Type	Name(s)	
Classes:		
`domain_error`	`length_error`	`overflow_error`
`exception`	`logic_error`	`range_error`
`invalid_argument`	`out_of_range`	`runtime_error`

6 The header `<stdexcept>` defines several types of predefined exceptions for reporting errors in a C++ program. These exceptions are related via inheritance, as indicated in Table 54:

Table 54—Standard exceptions inheritance hierarchy

Base class	Derived class	Derived class
exception		
	logic_error	
		domain_error
		invalid_argument
		length_error
		out_of_range
	runtime_error	
		range_error
		overflow_error

19.1.1 Class `exception` [lib.exception]

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

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

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

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

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

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

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

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

```
    exception();
```

2 Constructs an object of class `exception` and initializes *`desc`* to an unspecified value and *`alloced`* to zero.[98)]

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

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

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

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

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

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

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

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

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

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

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

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

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

19.1.2.2 `logic_error` destructor [lib.logic.error.des]

```
    virtual ~logic_error();
```

1 Destroys an object of class `logic_error`.

19.1.2.3 `logic_error::what` [lib.logic.error::what]

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

1 Behaves the same as `exception::what()`.

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

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

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

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

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

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

19.1.3.2 `domain_error` destructor [lib.domain.error.des]

```
    virtual ~domain_error();
```

1 Destroys an object of class `domain`.

19.1.3.3 `domain::what` [lib.domain.error::what]

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

1 Behaves the same as `exception::what()`.

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

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

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

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

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

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

19.1.4.2 `invalid_argument` destructor [lib.invalid.argument.des]

```
    virtual ~invalid_argument();
```

1 Destroys an object of class `invalid_argument`.

19.1.4.3 `invalid_argument::what` [lib.invalid.argument::what]

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

1 Behaves the same as `exception::what()`.

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

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

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

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

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

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

19.1.5.2 `length_error` destructor [lib.length.error.des]

```
    virtual ~length_error();
```

1 Destroys an object of class `length_error`.

19.1.5.3 `length_error::what` [lib.length.error::what]

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

1 Behaves the same as `exception::what()`.

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

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

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

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

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

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

19.1.6.2 `out_of_range` destructor [lib.out.of.range.des]

```
    virtual ~out_of_range();
```

1 Destroys an object of class `out_of_range`.

19.1.6.3 `out_of_range::what` [lib.out.of.range::what]

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

1 Behaves the same as `exception::what()`.

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

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

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

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

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

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

```
    runtime_error();
```

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

19.1.7.2 `runtime_error` destructor [lib.runtime.error.des]

```
    virtual ~runtime_error();
```

1 Destroys an object of class `runtime`.

19.1.7.3 `runtime::what` [lib.runtime.error::what]

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

1 Behaves the same as `exception::what()`.

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

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

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

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

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

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

19.1.8.2 `range_error` destructor [lib.range.error.des]

```
    virtual ~range_error();
```

1 Destroys an object of class `range_error`.

19.1.8.3 `range_error::what` [lib.range.error::what]

```
//  virtual int what() const;     inherited
```

1 Behaves the same as `exception::what()`.

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

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

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

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

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

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

19.1.9.2 `overflow_error` destructor [lib.overflow.error.des]

```
    virtual ~overflow_error();
```

1 Destroys an object of class `overflow_error`.

19.1.9.3 `overflow_error::what` [lib.overflow.error::what]

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

1 Behaves the same as `exception::what()`.

19.2 Assertions [lib.assertions]

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

2 Headers:

— `<cassert>`

3 Table 55:

Table 55—Header `<cassert>` synopsis

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

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

SEE ALSO: ISO C subclause 7.2.

19.3 Error numbers [lib.errno]

1 Headers:

— `<cerrno>`

2 Table 56:

Table 56—Header `<cerrno>` synopsis

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

3 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 core components (20.1), function objects (20.2), dynamic memory management utilities (20.3), and date/time utilities (20.4).

20.1 Core components [lib.core]

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

2 Headers:

— `<stl core (TBD)>`

3 Table 57:

Table 57—Header `<stl core (TBD)>` synopsis

Type	Name(s)	
Template class:	`restrictor`	
Template struct:	`pair`	
Template operators:		
`operator!= (T)`	`operator<= (T)`	`operator> (T)`
`operator< (pair)`	`operator== (pair)`	`operator>= (T)`
`operator< (restrictor)`	`operator== (restrictor)`	
Template function:	`make_pair`	
Struct:	`empty`	
Operator functions:	`operator< (empty)`	`operator== (empty)`

20.1.1 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>
inline bool operator!=(const T& x, const T& y) {
  return !(x == y);
}

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

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

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

20.1.2 Tuples [lib.tuples]

1 The library includes templates for heterogeneous n-tuples for n equal to 0 and 2. For non-empty tuples the library provides matching template functions to simplify their construction.[99)]

2 For example, instead of saying,

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

3 one may say

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

20.1.2.1 Empty [lib.empty]

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

```
    struct empty {};

    inline bool operator==(const empty&, const empty&) { return true; }
    inline bool operator< (const empty&, const empty&) { return false; }
```

20.1.2.2 Pair [lib.pair]

```
    template <class T1, class T2>
    struct pair {
        T1 first;
        T2 second;

        pair(const T1& x, const T2& y) : first(x), second(y) {}
    };

    template <class T1, class T2>
    inline bool operator==(const pair<T1, T2>& x, const pair<T1, T2>& y) {
      return x.first == y.first && x.second == y.second;
    }

    template <class T1, class T2>
    inline bool operator<(const pair<T1, T2>& x, const pair<T1, T2>& y) {
      return x.first < y.first || (!(y.first < x.first) && x.second < y.second);
    }

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

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

20.1.3 Restrictor [lib.restrictor]

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

```
template <class T>
class restrictor {
friend bool operator==(const restrictor<T>& x, const restrictor<T>& y);
friend bool operator< (const restrictor<T>& x, const restrictor<T>& y);
protected:
    T value;
public:
    restrictor(const T& x) : value(x) {}
};

template <class T>
inline bool operator==(const restrictor<T>& x, const restrictor<T>& y) {
  return x.value == y.value;
}

template <class T>
inline bool operator<(const restrictor<T>& x, const restrictor<T>& y) {
  return x.value < y.value;
}
```

20.2 Function objects [lib.function.objects]

1 Headers:

— `<stl functional (TBD)>`

2 Table 58:

Table 58—Header `<stl functional (TBD)>` synopsis

Type	Name(s)		
Template classes:			
`binary_negate`	`pointer_to_binary_function`		
`binder1st`	`pointer_to_unary_function`		
`binder2nd`	`unary_negate`		
Template structs:			
`binary_function`	`less`	`minus`	`times`
`divides`	`less_equal`	`modulus`	`unary_function`
`equal_to`	`logical_and`	`negate`	
`greater`	`logical_not`	`not_equal_to`	
`greater_equal`	`logical_or`	`plus`	
Template functions:			
	`bind1st`	`not1`	`ptr_fun [2]`
	`bind2nd`	`not2`	

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

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

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

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

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

6 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.

20.2.1 Base [lib.base]

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

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

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

20.2.2 Arithmetic operations [lib.arithmetic.operations]

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

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

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

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

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

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

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

20.2.3 Comparisons [lib.comparisons]

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

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

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

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

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

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

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

20.2.4 Logical operations [lib.logical.operations]

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

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

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

20.2.5 Negators [lib.negators]

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

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

template <class Predicate>
unary_negate<Predicate> not1(const Predicate& pred) {
  return unary_negate<Predicate>(pred);
}

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

    bool operator()(const first_argument_type& x,
                    const second_argument_type& y) const
      {
        return !value(x, y);
      }
};

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

20.2.6 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`.

20.2.6.1 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)
      : op(x), value(y) {}
    result_type operator()(const argument_type& x) const {
      return op(value, x);
    }
};
```

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

```
    template <class Operation, class T>
    binder1st<Operation> bind1st(const Operation& op, const T& x) {
      return binder1st<Operation>(op, Operation::first_argument_type(x));
    }
```

20.2.6.3 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)
          : op(x), value(y) {}
        result_type operator()(const argument_type& x) const {
          return op(x, value);
      }
    };
```

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

```
    template <class Operation, class T>
    binder2nd<Operation> bind2nd(const Operation& op, const T& x) {
      return binder2nd<Operation>(op, Operation::second_argument_type(x));
    }
```

1 For example,

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

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

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

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

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

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

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

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

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

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

2 For example,

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

replaces each `C` with `C++` in sequence `v`.[100)]

20.3 Memory [lib.memory]

1 Headers:

— `<stl memory (TBD)>`

— `<memory>`

— `<cstdlib>`

— `<cstring>`

2 Table 59:

Table 59—Header `<stl memory (TBD)>` synopsis

Type	**Name(s)**	
Template classes:	`allocator`	`raw_storage_iterator`
Template functions:		
`allocate`	`destroy [2]`	`uninitialized_fill_n`
`construct`	`get_temporary_buffer`	`unititialized_fill`
`deallocate`	`uninitialized_copy`	

3 Table 60:

Table 60—Header `<memory>` synopsis

Type	**Name(s)**	
Template functions:	objconstruct	objdestroy
`objcopy`	`objmove`	

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

4 Table 61:

Table 61—Header `<cstdlib>` synopsis

Type	Name(s)	
Functions:	`calloc`	`malloc`
	`free`	`realloc`

5 Table 62:

Table 62—Header `<cstring>` synopsis

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

SEE ALSO: ISO C subclause 7.11.2.

20.3.1 The default allocator [lib.default.allocator]

```
template <class T>
class allocator {
public:
  typedef T*        pointer;
  typedef const T*  const_pointer;
  typedef T         value_type;
  typedef size_t    size_type;
  typedef ptrdiff_t difference_type;

  allocator();
 ~allocator();

  pointer allocate(size_type n);
  void    deallocate(pointer p);
  size_t  init_page_size();
  size_t  max_size();
};
```

[101]

20.3.2 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`.

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

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

    raw_storage_iterator<OutputIterator, T>& operator=(const T& element) {
      construct(&*value, element);
      return *this;
    }

    raw_storage_iterator<OutputIterator, T>& operator++() {
      ++value;
      return *this;
    }

    raw_storage_iterator<OutputIterator, T> operator++(int) {
      raw_storage_iterator<OutputIterator, T> tmp = *this;
      ++value;
      return tmp;
    }
};
```

20.3.3 Memory handling primitives [lib.memory.primitives]

20.3.3.1 `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>
inline T* allocate(ptrdiff_t n, T*); // n >= 0
```

2 The size (in bytes) of the allocated buffer is no less than `n*sizeof(T)`.[102)]

20.3.3.2 `deallocate` [lib.deallocate]

1 Also, the following functions are provided:

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

20.3.3.3 `construct` [lib.construct]

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

102) 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>
inline T huge* allocate(long long n, T*);
```

20.3.3.4 destroy [lib.destroy]

```
template <class T>
inline void destroy(T* pointer) {
  pointer->T::~T();
}

template <class ForwardIterator>
void destroy(ForwardIterator first, ForwardIterator last) {
  while (first != last) destroy(&*first++);
}
```

[103]

20.3.3.5 get_temporary_buffer [lib.get.temporary.buffer]

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

1 `get_temporary_buffer` finds the largest buffer not greater than `n*sizeof(T)`, and returns a pair consisting of the address and the capacity (in the units of `sizeof(T)`) of the buffer.[104]

20.3.4 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`.

20.3.4.1 uninitialized_copy [lib.uninitialized.copy]

```
template <class InputIterator, class ForwardIterator>
ForwardIterator uninitialized_copy(InputIterator first, InputIterator last,
  ForwardIterator result)
{
  while (first != last) construct(&*result++, *first++);
  return result;
}
```

20.3.4.2 uninitialized_fill [lib.uninitialized.fill]

```
template <class ForwardIterator, class T>
void uninitialized_fill(ForwardIterator first, ForwardIterator last,
  const T& x)
{
  while (first != last) construct(&*first++, x);
}
```

[103] 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.
[104] It is guaranteed that for every memory model that an implementation supports, there is a corresponding `get_temporary_buffer` template function defined which is overloaded on the corresponding signed integral type. For example, if a system supports `huge` pointers and their difference is of type `long long`, the following function has to be provided:

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

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

```
template <class ForwardIterator, class Size, class T>
void uninitialized_fill_n(ForwardIterator first, Size n, const T& x)
{
  while (n--) construct(&*first++, x);
}
```

1 The header `<memory>` defines several template functions that copy, construct, and destroy arrays of objects.

20.3.4.4 `objcpy` [lib.template.objcpy]

```
template<class T> T* objcpy(T* dest, const T* src, size_t n);
```

1 Assigns *`src`*`[`*`I`*`]` to *`dest`*`[`*`I`*`]` for all non-negative values of *`I`* less than *`n`*. The pointers *`dest`* and *`src`* shall designate the initial elements of non-overlapping arrays of *`n`* objects of type *`T`*. The order in which assignments take place is unspecified.

2 Returns *`dest`*.

```
template<class T> T* objcpy(void* dest, const T* src, size_t n);
```

3 Constructs `((`*`T`*`*)`*`dest`*`)[`*`I`*`]` by copying *`src`*`[`*`I`*`]` for all non-negative values of *`I`* less than *`n`*. The pointer *`dest`* shall designate a region of storage suitable for representing an array of *`n`* objects of type *`T`*. The pointer *`src`* shall designate the initial element of an array of *`n`* objects of type *`T`* that does not overlap the region designated by *`dest`*. The order in which elements are constructed is unspecified.

4 Returns `(`*`T`*`*)`*`dest`*.

20.3.4.5 `objmove` [lib.template.objmove]

```
template<class T> T* objmove(T* dest, T* src, size_t n);
```

1 Assigns *`src`*`[`*`I`*`]` to *`dest`*`[`*`I`*`]` for all non-negative values of *`I`* less than *`n`*. The pointers *`dest`* and *`src`* shall designate the initial elements of arrays of *`n`* objects of type *`T`*. If *`dest`*` == `*`src`*, no assignment occurs.

2 Otherwise, each element of *`dest`* is destroyed after it has been assigned to its corresponding element in *`src`*. An element of *`dest`* that is also an element of *`src`* is first assigned to its corresponding element in *`src`*, then destroyed, before it is assigned to.

3 The order in which elements are assigned or destroyed is otherwise unspecified.

4 Returns *`dest`*.

```
template<class T> T* objmove(void* dest, T* src, size_t n);
```

5 Constructs `((`*`T`*`*)`*`dest`*`)[`*`I`*`]` by copying *`src`*`[`*`I`*`]` for all non-negative values of *`I`* less than *`n`*. The pointer *`dest`* shall designate a region of storage suitable for representing an array of *`n`* objects of type *`T`*. The pointer *`src`* shall designate the initial element of an array of *`n`* objects of type *`T`*. If *`dest`*` == (void*)`*`src`*, no construction occurs.

6 Otherwise, each element of *`dest`* is destroyed after it has been copied to its corresponding element in *`src`*. An element of *`dest`* that is also an element of *`src`* is first copied to its corresponding element in *`src`*, then destroyed, before it is constructed.

7 The order in which elements are constructed or destroyed is otherwise unspecified.

8 Returns `(`*`T`*`*)`*`dest`*.

20.3.4.6 `objconstruct` [lib.template.objcons]

```
template<class T> T* objconstruct(void* dest, size_t n);
```

1 Constructs `((T*)dest)[I]` with the constructor `T()` for all non-negative values of *I* less than *n*. The pointer *dest* shall designate a region of storage suitable for representing an array of *n* objects of type *T*. The order in which elements are constructed is unspecified.

2 Returns `(T*)dest`.

20.3.4.7 `objdestroy` [lib.template.objdes]

```
template<class T> void* objdestroy(T* dest, size_t n);
```

1 Destroys `((T*)dest)[I]` for all non-negative values of *I* less than *n*. The pointer *dest* shall designate an array of *n* objects of type *T*. The order in which elements are destroyed is unspecified.

2 Returns `(void*)dest`.

20.3.5 C library changes [lib.c.malloc]

1
2 The contents of `<cstdlib>` 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`.

4 The function `free` does not attempt to deallocate storage by calling `operator delete`.

SEE ALSO: ISO C subclause 7.11.2.

20.4 Date and time [lib.date.time]

1 Headers:

— `<ctime>`

2 Table 63:

Table 63—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`

3 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 (21), and null-terminated sequence utilities (21.2).

21.1 String classes [lib.string.classes]

1 Headers:

— `<string>`

2 Table 64:

Table 64—Header `<string>` synopsis

Type	Name(s)
Types:	`string` `wstring`
Template class:	`basic_string`

Operator functions:
`operator!= (basic_string) [5]`
`operator+  (basic_string) [5]`
`operator<< (basic_string)`
	
	`operator== (basic_string) [5]`
	`operator>> (basic_string)`

Function:	`getline`

3 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.

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

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

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

21.1.1.1 Template class `string_char_baggage` [lib.string.char.baggage]

Box 96

At the Kitchener meeting, the Library WG decided to change the names used from ‘‘`baggage`’’ to ‘‘`traits`’’. However, Tom Plum objected to doing this without a formal proposal and vote. These names are therefore unresolved, and subject to change.

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

    // constraints

    static void assign(char_type& c1, const char_type& c2)
        { c1 = c2; }
    static bool eq(const char_type& c1, const char_type& c2)
        { return (c1 == c2); }
    static bool ne(const char_type& c1, const char_type& c2)
        { return !(c1 == c2); }
    static bool lt(const char_type& c1, const char_type& c2)
        { return (c1 < c2); }
    static char_type eos() { return char_type(); }
        // the null character
    static basic_istream<charT>&
      char_in(basic_istream<charT>& is, charT& a) {
        return is >> a; // extractor for a charT object
      }
    static basic_ostream<charT>&
      char_out(basic_ostream<charT>& os, charT a) {
        return os << a;  // inserter for a charT object
      }
    static bool is_del(charT a) { return isspace(a); }
      // characteristic function for delimiters of charT

    // speed-up functions

    static int compare(const char_type* s1, const char_type* s2,
      size_t n) {
        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) {
        size_t l = 0;
        while (ne(*s++, eos()) ++l;
        return l;
    }
    static char_type* copy(char_type* s1, const char_type* s2, size_t n) {
        char_type* s = s1;
        for (size_t i = 0; i < n; ++i) assign(*++s1, *++s2);
        return s;
    }
};
```

21.1.1.2 Template class `basic_string` [lib.basic.string]

```
template<class charT, class baggage = string_char_baggage<charT> >
class basic_string {
public:
    typedef charT char_type;
    typedef baggage baggage_type;
```

```
 basic_string();
 basic_string(size_t size, capacity cap);
 basic_string(const basic_string& str, size_t pos = 0, size_t n = NPOS);
 basic_string(const charT* s, size_t n);
 basic_string(const charT* s);
 basic_string(charT c, size_t rep = 1);
~basic_string();
 basic_string& operator=(const basic_string& str);
 basic_string& operator=(const charT* s);
 basic_string& operator=(charT c);

 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_t pos = 0, size_t n = NPOS);
 basic_string& append(const charT* s, size_t n);
 basic_string& append(const charT* s);
 basic_string& append(charT c, size_t rep = 1);

 basic_string& assign(const basic_string& str, size_t pos = 0, size_t n = NPOS);
 basic_string& assign(const charT* s, size_t n);
 basic_string& assign(const charT* s);
 basic_string& assign(charT c, size_t rep = 1);

 basic_string& insert(size_t pos1, const basic_string& str, size_t pos2 = 0,
                      size_t n = NPOS);
 basic_string& insert(size_t pos, const charT* s, size_t n);
 basic_string& insert(size_t pos, const charT* s);
 basic_string& insert(size_t pos, charT c, size_t rep = 1);

 basic_string& remove(size_t pos = 0, size_t n = NPOS);
 basic_string& replace(size_t pos1, size_t n1, const basic_string& str,
                       size_t pos2 = 0, size_t n2 = NPOS);
 basic_string& replace(size_t pos, size_t n1, const charT* s, size_t n2);
 basic_string& replace(size_t pos, size_t n1, const charT* s);
 basic_string& replace(size_t pos, size_t n, charT c, size_t rep = 1);

 charT get_at(size_t pos) const;
 void  put_at(size_t pos, charT c);
 charT  operator[](size_t pos) const;
 charT& operator[](size_t pos);

 const charT* c_str() const;
 const charT* data() const;
 size_t length() const:
 void resize(size_t n, charT c);
 void resize(size_t n);
 size_t reserve() const;
 void reserve(size_t res_arg);
 size_t copy(charT* s, size_t n, size_t pos = 0);

 size_t find(const basic_string& str, size_t pos = 0) const;
 size_t find(const charT* s, size_t pos, size_t n) const;
 size_t find(const charT* s, size_t pos = 0) const;
 size_t find(charT c, size_t pos = 0) const;
 size_t rfind(const basic_string& str, size_t pos = NPOS) const;
 size_t rfind(const charT* s, size_t pos, size_t n) const;
 size_t rfind(const charT* s, size_t pos = NPOS) const;
 size_t rfind(charT c, size_t pos = NPOS) const;
```

```
    size_t find_first_of(const basic_string& str, size_t pos = 0) const;
    size_t find_first_of(const charT* s, size_t pos, size_t n) const;
    size_t find_first_of(const charT* s, size_t pos = 0) const;
    size_t find_first_of(charT c, size_t pos = 0) const;
    size_t find_last_of (const basic_string& str, size_t pos = NPOS) const;
    size_t find_last_of (const charT* s, size_t pos, size_t n) const;
    size_t find_last_of (const charT* s, size_t pos = NPOS) const;
    size_t find_last_of (charT c, size_t pos = NPOS) const;

    size_t find_first_not_of(const basic_string& str, size_t pos = 0) const;
    size_t find_first_not_of(const charT* s, size_t pos, size_t n) const;
    size_t find_first_not_of(const charT* s, size_t pos = 0) const;
    size_t find_first_not_of(charT c, size_t pos = 0) const;
    size_t find_last_not_of (const basic_string& str, size_t pos = NPOS) const;
    size_t find_last_not_of (const charT* s, size_t pos, size_t n) const;
    size_t find_last_not_of (const charT* s, size_t pos = NPOS) const;
    size_t find_last_not_of (charT c, size_t pos = NPOS) const;

    basic_string substr(size_t pos = 0, size_t n = NPOS) const;
    int compare(const basic_string& str, size_t pos = 0, size_t n = NPOS) const;
    int compare(charT* s, size_t pos, size_t n) const;
    int compare(charT* s, size_t pos = 0) const;
    int compare(charT  c, size_t pos = 0, size_t rep = 1) const;
private:
    static charT eos() { return baggage::eos(); }
//  charT* ptr;          exposition only
//  size_t len, res;     exposition only
};
```

1 For a char-like type *charT*, the template class `basic_string<charT,baggage>` 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<charT,baggage>`.

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

— *charT* ptr*, points to the initial char-like object of the string;

— *size_t len*, counts the number of char-like objects currently in the string;

— *size_t res*, for an unallocated string, holds the estimated maximum size of the string, while for an allocated string, becomes the currently allocated size.

3 In all cases, *len <= res*.

4 The functions described in this clause can report two kinds of errors, each associated with a distinct exception:

— a `length` error is associated with exceptions of type `length_error`;

— an *out-of-range* error is associated with exceptions of type `out_of_range`.

21.1.1.3 `basic_string` member functions [lib.string.members]

21.1.1.3.1 `basic_string` constructors [lib.string.cons]

```
basic_string();
```

1 Constructs an object of class `basic_string<charT,baggage>`.

2 The postconditions of this function are indicated in Table 65:

Table 65—`basic_string()` effects

Element	Value
`ptr`	an unspecified value
`len`	0
`res`	an unspecified value

```
basic_string(size_t size, capacity cap);
```

3 Constructs an object of class `basic_string<charT,baggage>`. If *`cap`* is *`default_size`*, the function either throws `length_error` if *`size`* equals `NPOS` or initializes the string as indicated in Table 66:

Table 66—`basic_string(size_t,capacity)` effects

Element	Value
`ptr`	points at the first element of an allocated array of *`size`* elements, each of which is initialized to zero
`len`	*`size`*
`res`	a value at least as large as *`len`*

4 Otherwise, *`cap`* shall be reserve and the function initializes the string as indicated in Table 67:

Table 67—`basic_string(size_t,capacity)` effects

Element	Value
`ptr`	an unspecified value
`len`	0
`res`	*`size`*

```
basic_string(const basic_string<charT,baggage>& str,
             size_t pos = 0, size_t n = NPOS);
```

5 Throws `out_of_range` if *`pos > str.len`*. Otherwise, the function constructs an object of class `basic_string<charT,baggage>` and determines the effective length *`rlen`* of the initial string value as the smaller of *`n`* and *`str.len - pos`*. Thus, the function initializes the string as indicated in Table 68:

Table 68—basic_string(basic_string,size_t,size_t) effects

Element	Value
ptr	points at the first element of an allocated copy of *rlen* elements of the string controlled by *str* beginning at position *pos*
len	*rlen*
res	a value at least as large as *len*

```
basic_string(const charT* s, size_t n = NPOS);
```

6 Constructs an object of class basic_string<charT,baggage> and determines its initial string value from the array of *charT* of length *n* whose first element is designated by *s*. *s* shall not be a null pointer. Thus, the function initializes the string as indicated in Table 69:

Table 69—basic_string(const charT*,size_t) effects

Element	Value
ptr	points at the first element of an allocated copy of the array whose first element is pointed at by *s*
len	*n*
res	a value at least as large as *len*

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

7 Constructs an object of class basic_string<charT,baggage> and determines its initial string value from the array of *charT* of length baggage::length(s) whose first element is designated by *s*. *s* shall not be a null pointer. Thus, the function initializes the string as indicated in Table 70:

Table 70—basic_string(const charT*) effects

Element	Value
ptr	points at the first element of an allocated copy of the array whose first element is pointed at by *s*
len	*baggage::length(s)*
res	a value at least as large as *len*

8 Uses baggage::length().

```
basic_string(charT c, size_t rep = 1);
```

9 Throws length_error if *rep* equals NPOS. Otherwise, the function constructs an object of class basic_string<charT,baggage> and determines its initial string value by repeating the char-like object *c* for all *rep* elements. Thus, the function initializes the string as indicated in Table 71:

Table 71—basic_string(charT,size_t) effects

Element	Value
ptr	points at the first element of an allocated array of *rep* elements, each storing the initial value *c*
len	*rep*
res	a value at least as large as *len*

21.1.1.3.2 basic_string destructor [lib.string.des]

```
~basic_string();
```

1 Destroys an object of class `basic_string<charT,baggage>`.

21.1.1.3.3 basic_string::operator= [lib.string::op=]

```
basic_string<charT,baggage>& operator=(const basic_string<charT,baggage>& str);
```

1 Returns `*this = basic_string<charT,baggage>(str)`.

```
basic_string<charT,baggage>& operator=(const charT* s);
```

2 Returns `*this = basic_string<charT,baggage>(s)`.

3 Uses `baggage::length()`.

```
basic_string<charT,baggage>& operator=(charT c);
```

4 Returns `*this = basic_string<charT,baggage>(c)`.

21.1.1.3.4 basic_string::operator+= [lib.string::op+=]

```
basic_string<charT,baggage>& operator+=(const basic_string<charT,baggage>& rhs);
```

1 Returns `append(rhs)`.

```
basic_string<charT,baggage>& operator+=(const charT* s);
```

2 Returns `*this += basic_string<charT,baggage>(s)`.

3 Uses `baggage::length()`.

```
basic_string<charT,baggage>& operator+=(charT c);
```

4 Returns `*this += basic_string<charT,baggage>(c)`.

21.1.1.3.5 basic_string::append [lib.string::append]

```
basic_string<charT,baggage>& append(const basic_string<charT,baggage>& str,
                                    size_t pos = 0, size_t n = NPOS);
```

1 Throws `out_of_range` if *pos > str.len*. Otherwise, the function determines the effective length *rlen* of the string to append as the smaller of *n* and *str.len - pos*. The function then throws `length_error` if *len >= NPOS - rlen*.

2 Otherwise, the function replaces the string controlled by `*this` with a string of length *len + rlen* whose first *len* elements are a copy of the original string controlled by `*this` and whose remaining elements are a copy of the initial elements of the string controlled by *str* beginning at position *pos*.

3 Returns `*this`.

```
basic_string<charT,baggage>& append(const charT* s, size_t n);
```

4 Returns `append(basic_string(s, n))`.

```
basic_string<charT,baggage>& append(const charT* s);
```

5 Returns `append(basic_string<charT,baggage>(s))`.

6 Uses *baggage::length()*.

```
basic_string<charT,baggage>& append(charT c, size_t rep = 1);
```

7 Returns `append(basic_string<charT,baggage>(c, rep))`.

21.1.1.3.6 **basic_string::assign** [lib.string::assign]

```
basic_string<charT,baggage>& assign(const basic_string<charT,baggage>& str,
                                    size_t pos = 0, size_t n = NPOS);
```

1 Throws `out_of_range` if *pos > str.len*. Otherwise, the function determines the effective length *rlen* of the string to assign as the smaller of *n* and *str.len - pos*.

2 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*.

3 Returns `*this`.

```
basic_string<charT,baggage>& assign(const charT* s, size_t n);
```

4 Returns `assign(basic_string<charT,baggage>(s, n))`.

```
basic_string<charT,baggage>& assign(const charT* s);
```

5 Returns `assign(basic_string(s))`.

6 Uses `baggage::length()`.

```
basic_string<charT,baggage>& assign(charT c, size_t rep = 1);
```

7 Returns `assign(basic_string<charT,baggage>(c, rep))`.

21.1.1.3.7 **basic_string::insert** [lib.string::insert]

```
basic_string<charT,baggage>&
  insert(size_t pos1, const basic_string<charT,baggage>& str,
         size_t pos2 = 0, size_t n = NPOS);
```

1 Throws `out_of_range` if *pos1 > len* or *pos2 > str.len*. Otherwise, the function determines the effective length *rlen* of the string to insert as the smaller of *n* and *str.len - pos2*. Then throws `length_error` if *len >= NPOS - rlen*.

2 Otherwise, the function replaces the string controlled by `*this` with a string of length *len + 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`.

3 Returns `*this`.

```
basic_string<charT,baggage>& insert(size_t pos, const charT* s, size_t n);
```

4 Returns `insert(pos, basic_string<charT,baggage>(s, n))`.

```
basic_string<charT,baggage>& insert(size_t pos, const charT* s);
```

5 Returns `insert(pos, basic_string(s))`.

6 Uses *`baggage::length()`*.

```
basic_string<charT,baggage>& insert(size_t pos, charT c, size_t rep = 1);
```

7 Returns `insert(pos, basic_string<charT,baggage>(c, rep))`.

21.1.1.3.8 basic_string::remove [lib.string::remove]

```
basic_string<charT,baggage>& remove(size_t pos = 0, size_t n = NPOS);
```

1 Throws `out_of_range` if *`pos > len`*. Otherwise, the function determines the effective length *`xlen`* of the string to be removed as the smaller of *`n`* and *`len - pos`*.

2 The function then replaces the string controlled by `*this` with a string of length *`len - 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`*.

3 Returns `*this`.

21.1.1.3.9 basic_string::replace [lib.string::replace]

```
basic_string<charT,baggage>&
  replace(size_t pos1, size_t n1,
          const basic_string<charT,baggage>& str,
          size_t pos2 = 0, size_t n2 = NPOS);
```

1 Throws `out_of_range` if *`pos1 > len`* or *`pos2 > str.len`*. Otherwise, the function determines the effective length *`xlen`* of the string to be removed as the smaller of *`n1`* and *`len - pos1`*. It also determines the effective length *`rlen`* of the string to be inserted as the smaller of *`n2`* and *`str.len - pos2`*. Then throws `length_error` if *`len - xlen >= NPOS - rlen`*.

2 Otherwise, the function replaces the string controlled by `*this` with a string of length *`len - 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`*.

3 Returns `*this`.

```
basic_string<charT,baggage>& replace(size_t pos, size_t n1,
                                     const charT* s, size_t n2);
```

4 Returns `replace(pos, n1, basic_string<charT,baggage>(s, n2))`.

```
basic_string<charT,baggage>& replace(size_t pos, size_t n1, const charT* s);
```

5 Returns `replace(pos, n1, basic_string<charT,baggage>(s))`.

6 Uses `baggage::length()`.

```
basic_string<charT,baggage>& replace(size_t pos, size_t n,
                                     charT c, size_t rep = 1);
```

7 Returns `replace(pos, n, basic_string<charT,baggage>(c, rep))`.

21.1.1.3.10 `basic_string::get_at` [lib.string::get.at]

```
charT get_at(size_t pos) const;
```

1 Throws `out_of_range` if *`pos >= len`*. Otherwise, returns `ptr[pos]`.

21.1.1.3.11 `basic_string::put_at` [lib.string::put.at]

```
void put_at(size_t pos, charT c);
```

1 Throws `out_of_range` if *`pos > len`*. Otherwise, if *`pos == len`*, the function replaces the string controlled by `*this` with a string of length *`len + 1`* whose first *`len`* elements are a copy of the original string and whose remaining element is initialized to *`c`*. Otherwise, the function assigns *`c`* to *`ptr[pos]`*.

21.1.1.3.12 `basic_string::operator[]` [lib.string::op.array]

```
charT  operator[](size_t pos) const;
charT& operator[](size_t pos);
```

1 If *`pos < len`*, returns `ptr[pos]`. Otherwise, if *`pos == len`*, the const version returns zero. Otherwise, the behavior is undefined.

2 The reference returned by the non-const version is invalid after any subsequent call to *`c_str`* or any non-const member function for the object.

21.1.1.3.13 `basic_string::c_str` [lib.string::c.str]

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

1 Returns a pointer to the initial element of an array of length *`len + 1`* whose first *`len`* elements equal the corresponding elements of the string controlled by `*this` and whose last element is a null character specified by `eos()`. 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<charT,baggage>`* that designates the same object as *`this`*.

2 Uses `baggage::eos()`.

21.1.1.3.14 `basic_string::data` [lib.string::data]

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

1 Returns `ptr` if *`len`* is nonzero, otherwise a null pointer. 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 the class *`basic_string<charT,baggage>`* that designates the same object as *`this`*.

21.1.1.3.15 `basic_string::length` [lib.string::length]

```
size_t length() const:
```

1 Returns `len`.

21.1.1.3.16 `basic_string::resize` [lib.string::resize]

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

1 Throws `length_error` if *n* equals `NPOS`. Otherwise, the function alters the length of the string designated by `*this` as follows:

2 If *n <= len*, 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`.

3 If *n > len*, the function replaces the string designated by `*this` with a string of length *n* whose first *len* elements are a copy of the original string designated by `*this`, and whose remaining elements are all initialized to *c*.

```
void resize(size_t n);
```

4 Returns `resize(n, eos())`.

5 Uses `baggage::eos()`.

21.1.1.3.17 `basic_string::reserve` [lib.string::reserve]

```
size_t reserve() const;
```

1 Returns *res*.

```
void reserve(size_t res_arg);
```

2 If no string is allocated, the function assigns *res_arg* to *res*. Otherwise, whether or how the function alters *res* is unspecified.

21.1.1.3.18 `basic_string::copy` [lib.string::copy]

```
size_t copy(charT* s, size_t n, size_t pos = 0);
```

1 Throws `out_of_range` if *pos > len*. Otherwise, the function determines the effective length *rlen* of the string to copy as the smaller of *n* and *len - pos*. *s* shall designate an array of at least *rlen* elements.

2 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*.[105)]

3 Returns *rlen*.

21.1.1.3.19 `basic_string::find` [lib.string::find]

```
size_t find(const basic_string<charT,baggage>& str, size_t pos = 0) const;
```

1 Determines the lowest position *xpos*, if possible, such that both of the following conditions obtain:

— *pos <= xpos* and *xpos + str.len <= len*;

— `baggage::eq(ptr[xpos + I], str.ptr[I])` for all elements *I* of the string controlled by *str*.

2 Returns *xpos* if the function can determine such a value for*xpos*. Otherwise, returns `NPOS`.

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

105) The function does not append a null object to the string.

3 Returns `find(basic_string(s, n), pos)`.

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

4 Returns `find(basic_string<charT,baggage>(s), pos)`.

5 Uses `baggage::length()`.

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

6 Returns `find(basic_string<charT,baggage>(c), pos)`.

21.1.1.3.20 `basic_string::rfind` [lib.rfind]

```
size_t rfind(const basic_string<charT,baggage>& str, size_t pos = NPOS) const;
```

1 Determines the highest position *`xpos`*, if possible, such that both of the following conditions obtain:

— *`xpos <= pos`* and *`xpos + str.len <= len`*;

— `baggage::eq(ptr[xpos + I], str.ptr[I])` for all elements *`I`* of the string controlled by *`str`*.

2 Returns *`xpos`* if the function can determine such a value for *`xpos`*. Otherwise, returns `NPOS`.

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

3 Returns `rfind(basic_string<charT,baggage>(s, n), pos)`.

```
size_t rfind(const charT* s, size_t pos = NPOS) const;
```

4 Returns `rfind(basic_string(s), pos)`.

5 Uses `baggage::length()`.

```
size_t rfind(charT c, size_t pos = NPOS) const;
```

6 Returns `rfind(basic_string<charT,baggage>(c, n), pos)`.

21.1.1.3.21 `basic_string::find_first_of` [lib.string::find.first.of]

```
size_t find_first_of(const basic_string<charT,baggage>& str,
                     size_t pos = 0) const;
```

1 Determines the lowest position *`xpos`*, if possible, such that both of the following conditions obtain:

— *`pos <= xpos`* and *`xpos < len`*;

— `baggage::eq(ptr[xpos], str.ptr[I])` for some element *`I`* of the string controlled by *`str`*.

2 Returns *`xpos`* if the function can determine such a value for *`xpos`*. Otherwise, returns `NPOS`.

```
size_t find_first_of(const charT* s, size_t pos, size_t n) const;
```

3 Returns `find_first_of(basic_string(s, n), pos)`.

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

4 Returns `find_first_of(basic_string(s), pos)`.

5 Uses `baggage::length()`.

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

6 Returns `find_first_of(basic_string(c), pos)`.

21.1.1.3.22 `basic_string::find_last_of` [lib.string::find.last.of]

```
size_t find_last_of(const basic_string<charT,baggage>& str,
                    size_t pos = NPOS) const;
```

1 Determines the highest position *xpos*, if possible, such that both of the following conditions obtain:

— *xpos <= pos* and *pos < len*;

— `baggage::eq(ptr[xpos], str.ptr[I])` for some element *I* of the string controlled by *str*.

2 Returns *xpos* if the function can determine such a value for *xpos*. Otherwise, returns `NPOS`.

```
size_t find_last_of(const charT* s, size_t pos, size_t n) const;
```

3 Returns `find_last_of(basic_string(s, n), pos)`.

```
size_t find_last_of(const charT* s, size_t pos = NPOS) const;
```

4 Returns `find_last_of(basic_string(s), pos)`.

5 Uses `baggage::length()`.

```
size_t find_last_of(charT c, size_t pos = NPOS) const;
```

6 Returns `find_last_of(basic_string<charT,baggage>(c), pos)`.

21.1.1.3.23 `basic_string::find_first_not_of` [lib.string::find.first.not.of]

```
size_t find_first_not_of(const basic_string<charT,baggage>& str,
                         size_t pos = 0) const;
```

1 Determines the lowest position *xpos*, if possible, such that both of the following conditions obtain:

— *pos <= xpos* and *xpos < len*;

— `baggage::eq(ptr[xpos], str.ptr[I])` for no element *I* of the string controlled by *str*.

2 Returns *xpos* if the function can determine such a value for *xpos*. Otherwise, returns `NPOS`.

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

3 Returns `find_first_not_of(basic_string<charT,baggage>(s, n), pos)`.

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

4 Returns `find_first_not_of(basic_string<charT,baggage>(s), pos)`.

5 Uses `baggage::length()`.

```
size_t find_first_not_of(charT, size_t pos = 0) const;
```

6 Returns `find_first_not_of(basic_string(c), pos)`.

21.1.1.3.24 `basic_string::find_last_not_of` [lib.string::find.last.not.of]

```
size_t find_last_not_of(const basic_string<charT,baggage>& str,
                        size_t pos = NPOS) const;
```

1 Determines the highest position *xpos*, if possible, such that both of the following conditions obtain:

— *xpos <= pos* and *pos < len*;

— `baggage::eq(ptr[xpos], str.ptr[I])` for no element *I* of the string controlled by *str*.

2 Returns *xpos* if the function can determine such a value for *xpos*. Otherwise, returns `NPOS`.

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

3 Returns `find_last_not_of(basic_string(s, n), pos)`.

```
size_t find_last_not_of(const charT* s, size_t pos = NPOS) const;
```

4 Returns `find_last_not_of(basic_string<charT,baggage>(s), pos)`.

5 Uses `baggage::length()`.

```
size_t find_last_not_of(charT c, size_t pos = NPOS) const;
```

6 Returns `find_last_not_of(basic_string<charT,baggage>(c), pos)`.

21.1.1.3.25 `basic_string::substr` [lib.string::substr]

```
basic_string<charT,baggage> substr(size_t pos = 0,
                                   size_t n = NPOS) const;
```

1 Throws `out_of_range` if *pos > len*. Otherwise, the function determines the effective length *rlen* of the string to copy as the smaller of *n* and *len - pos*.

2 Returns `basic_string(ptr + pos, rlen)`.

21.1.1.3.26 `basic_string::compare` [lib.string::compare]

```
int compare(const basic_string<charT,baggage>& str,
            size_t pos = 0, size_t n = NPOS)
```

1 Throws `out_of_range` if *pos > len*. Otherwise, the function determines the effective length *rlen* of the strings to compare as the smallest of *n*, *len - pos*, and *str.len*. The function then compares the two strings by calling `baggage::compare(ptr + pos, str.ptr, rlen)`.

2 Returns the nonzero result if the result of the comparison is nonzero. Otherwise, returns a value as indicated in Table 72:

Table 72—`compare()` results

Condition	Return Value
len - pos < str.len	a value less than 0
len - pos == str.len	0
len - pos > str.len	a value greater than 0

— if *len < rlen*, a value less than zero;

— if *len == rlen*, the value zero;

— if *len > rlen*, a value greater than zero.

3 Uses `baggage::compare()`.

```
int compare(const charT* s, size_t pos, size_t n) const;
```

4 Returns `compare(basic_string<charT,baggage>(s, n), pos)`.

5 Uses `baggage::compare()`.

```
int compare(const charT* s, size_t pos = 0) const;
```

6 Returns `compare(basic_string<charT,baggage>(s), pos)`.

7 Uses `baggage::length()` and `baggage::compare()`.

```
int compare(charT c, size_t pos = 0, size_t rep = 1) const;
```

8 Returns `compare(basic_string(c, rep), pos)`.

9 Uses `baggage::compare()`.

21.1.1.4 `basic_string` non-member functions [lib.string.nonmembers]

21.1.1.4.1 `operator+` [lib.string::op+]

```
basic_string<charT,baggage>
  operator+(const basic_string<charT,baggage>& lhs,
            const basic_string<charT,baggage>& rhs);
```

1 Returns `basic_string<charT,baggage>(lhs).append(rhs)`.

```
basic_string<charT,baggage>
  operator+(const charT* lhs, const basic_string<charT,baggage>& rhs);
```

2 Returns *`basic_string<charT,baggage>(lhs) + rhs`*.

3 Uses `baggage::length()`.

```
basic_string<charT,baggage>
  operator+(charT lhs, const basic_string<charT,baggage>& rhs);
```

4 Returns *`basic_string<charT,baggage>(lhs) + rhs`*.

```
basic_string<charT,baggage>
  operator+(const basic_string<charT,baggage>& lhs, const charT* rhs);
```

5 Returns `basic_string(lhs) + basic_string(rhs)`.

6 Uses `baggage::length()`.

```
basic_string<charT,baggage>
  operator+(const basic_string<charT,baggage>& lhs, charT rhs);
```

7 Returns `lhs + basic_string<charT,baggage>(rhs)`.

21.1.1.4.2 `operator==` [lib.string::operator==]

```
bool operator==(const basic_string<charT,baggage>& lhs,
                const basic_string<charT,baggage>& rhs);
```

1 Returns the value `true` if `lhs.compare(rhs)` is zero.

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

2 Returns *`basic_string<charT,baggage>(lhs) == rhs`*.

3 Uses `baggage::length()`.

```
bool operator==(charT lhs, const basic_string<charT,baggage>& rhs);
```

4 Returns *basic_string(lhs) == rhs*.

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

5 Returns `lhs == basic_string<charT,baggage>(rhs)`.

6 Uses `baggage::length()`.

```
bool operator==(const basic_string<charT,baggage>& lhs, charT rhs);
```

7 Returns `lhs == basic_string<charT,baggage>(rhs)`.

21.1.1.4.3 `operator!=` [lib.string::op!=]

```
bool operator!=(const basic_string<charT,baggage>& lhs,
                const basic_string<charT,baggage>& rhs);
```

1 Returns the value `true` if `lhs.compare(rhs)` is nonzero.

```
bool operator!=(const charT* lhs, const basic_string<charT,baggage>& rhs);
```

2 Returns *basic_string<charT,baggage>(lhs) != rhs*.

3 Uses `baggage::length()`.

```
bool operator!=(charT lhs, const basic_string<charT,baggage>& rhs);
```

4 Returns *basic_string<charT,baggage>(lhs) != rhs*.

```
bool operator!=(const basic_string<charT,baggage>& lhs, const charT* rhs);
```

5 Returns `lhs != basic_string<charT,baggage>(rhs)`.

6 Uses `baggage::length()`.

```
bool operator!=(const basic_string<charT,baggage>& lhs, charT rhs);
```

7 Returns `lhs != basic_string<charT,baggage>(rhs)`.

21.1.1.4.4 Inserters and extractors

1

```
template<class charT>
basic_istream<charT>&
  operator>>(basic_istream<charT>& is, basic_string<charT>& a);
```

2 Implement with `string_char_baggage<charT>::char_in` and `is_del()`.

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

3 Implement with `string_char_baggage<charT>::char_out()`.

21.1.2 Class `string` [lib.string]

```
struct string_char_baggage<char> {
    typedef char char_type;
    static void assign(char_type& c1, const char_type& c2)
        { c1 = c2; }
    static bool eq(const char_type& c1, const char_type& c2)
        { return (c1 == c2); }
    static bool ne(const char_type& c1, const char_type& c2)
        { return (c1 != c2); }
    static bool lt(const char_type& c1, const char_type& c2)
        { return (c1 < c2); }
    static char_type eos() { return 0; }

    static int compare(const char_type* s1, const char_type* s2,
      size_t n) {
        return memcmp(s1, s2, n);
    }

    static size_t length(const char_type* s) {
        return strlen(s);
    }

    static char_type* copy(char_type* s1, const char_type* s2, size_t n) {
        return memcpy(s1, s2, n);
    }
};

typedef basic_string<char> string;
```

21.1.3 Class `wstring` [lib.wstring]

```
struct string_char_baggage<wchar_t> {
    typedef wchar_t char_type;

    static void assign(char_type& c1, const char_type& c2)
        { c1 = c2; }
    static bool eq(const char_type& c1, const char_type& c2)
        { return (c1 == c2); }
    static bool ne(const char_type& c1, const char_type& c2)
        { return (c1 != c2); }
    static bool lt(const char_type& c1, const char_type& c2)
        { return (c1 < c2); }
    static char_type eos() { return 0; }

    static int compare(const char_type* s1, const char_type* s2, size_t n) {
        return wmemcmp(s1, s2, n);
    }

    static size_t length(const char_type* s) {
        return wcslen(s);
    }

    static char_type* copy(char_type* s1, const char_type* s2, size_t n) {
        return wmemcpy(s1, s2, n);
    }
};

typedef basic_string<wchar_t> wstring;
```

21.2 Null-terminated sequence utilities [lib.c.strings]

1 Headers:

— `<cctype>`

— `<cwctype>`

— `<cstring>`

— `<cwchar>`

— `<cstdlib>` multibyte conversions

— `<ciso646>`

2 Table 73:

Table 73—Header `<cctype>` synopsis

Type	Name(s)			
Functions:				
`isalnum`	`isdigit`	`isprint`	`isupper`	`tolower`
`isalpha`	`isgraph`	`ispunct`	`isxdigit`	`toupper`
`iscntrl`	`islower`	`isspace`		

3 Table 74:

Table 74—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`	

4 Table 75:

Table 75—Header `<cstring>` synopsis

Type	Name(s)			
Macro:	`NULL <cstring>`			
Type:	`size_t <cstring>`			
Functions:				
`strcoll`		`strlen`	`strpbrk`	`strtok`
`strcat`	`strcpy`	`strncat`	`strrchr`	`strxfrm`
`strchr`	`strcspn`	`strncmp`	`strspn`	
`strcmp`	`strerror`	`strncpy`	`strstr`	

5 Table 76:

Table 76—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`	

6 Table 77:

Table 77—Header `<cstdlib>` synopsis

Type	Name(s)		
Macros:	`MB_CUR_MAX`		
Functions:			
`atol`	`mblen`	`strtod`	`wctomb`
`atof`	`mbstowcs`	`strtol`	`wcstombs`
`atoi`	`mbtowc`	`stroul`	

7 The contents are the same as the Standard C library, with the following modifications:

8 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);
```

9 both of which have the same behavior as the original declaration.

10 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);
```

11 both of which have the same behavior as the original declaration.

12 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);
```

13 both of which have the same behavior as the original declaration.

14 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);
```

15 both of which have the same behavior as the original declaration.

16 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);
```

17 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 perform localization of strings. The locale facility includes internationalization support for character classification and collation, numeric and currency punctuation, date and time formatting, and message retrieval.

2 The following subclauses describe components for locales (22.1), standard facets (22.2), and facilities included from the ISO C library (22.3).

22.1 Locales [lib.locales]

1 Headers:

— `<locale>`

2 Table 78:

Table 78—Header `<locale>` synopsis

Type	Name(s)	
Template classes:		
`locale::codecvt`	`locale::msg`	
`locale::codecvt_byname`	`locale::msg_byname`	
`locale::collate`	`locale::numpunct`	
`locale::collate_byname`	`locale::num_get`	
`locale::ctype`	`locale::num_put`	
`locale::ctype_byname`	`locale::time_get`	
`locale::moneypunct`	`locale::time_get_byname`	
`locale::moneypunct_byname`	`locale::time_put`	
`locale::money_get`	`locale::time_put_byname`	
`locale::money_put`		
Classes:	`locale`	`locale::ctype<char>`
Struct:	`locale::ctype_base`	
Operator functions:	`operator<< (locale)`	`operator>> (locale)`

3 The header `<locale>` defines classes and several functions that encapsulate and manipulate the information peculiar to a locale.[106)]

22.1.1 Class `locale` [lib.locale]

106) In this subclause, the type name `struct tm` is an incomplete type that is defined in `<ctime>`.

```
class locale {
public:
  class id;
  class facet;
  class ctype_base;
  template <class charT> class ctype;
                         class ctype<char>;        // specialization
  template <class charT> class ctype_byname;
                         class ctype_byname<char>; // specialization
  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;
  template <class charT> class collate;
  template <class charT> class collate_byname;
  template <class fromT, class toT, class stateT> class codecvt;
  template <class fromT, class toT, class stateT> class codecvt_byname;
  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;
  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;
  template <class charT> class msg;
  template <class charT> class msg_byname;

  // member function template use<Facet>() returns a reference to
  // a facet.  Its result is guaranteed by locale’s value semantics
  // to last at least as long as the locale it came from.  If the
  // facet requested is not present in *this, it returns the facet
  // from the current global locale, if present, or throws an exception.

  template <class Facet>
  const Facet& use() const
  {
    int i = (const id&) Facet::id;    // verify is a locale::id.
    facet* f = (Facet*) 0;            // verify derived from facet.
    return static_cast<const Facet&>(
      (i < imp_->vec_.size() && (f = imp_->vec_[i])) ? *f : delegate(i));
  }

  // has<Facet>() simply reports whether a locale implements a particular
  //   facet.  If (loc.has<facet>() || locale().has<facet>()) is false,
  //   loc.use<facet>() would throw an exception.

  template <class Facet>
  bool has() const
  {
    facet* null = (Facet*) 0;          // verify derived from facet.
    size_t ix = (const id&) Facet::id; // verify is a locale::id.
    return (ix < imp_->vec_.size()) && imp_->vec_[ix];
  }
```

```
const basic_string<char>& name() const { return imp_->name_; }

bool operator==(const locale& other) const
{ return ((imp_ == other.imp_) ||
          (name() != "*" && name() == other.name()))
}

bool operator!=(const locale& other) const { return !(*this == other); }

// convenience interfaces
template <class charT> inline bool isspace(charT c) const;
template <class charT> inline bool isprint(charT c) const;
template <class charT> inline bool iscntrl(charT c) const;
template <class charT> inline bool isupper(charT c) const;
template <class charT> inline bool islower(charT c) const;
template <class charT> inline bool isalpha(charT c) const;
template <class charT> inline bool isdigit(charT c) const;
template <class charT> inline bool ispunct(charT c) const;
template <class charT> inline bool isxdigit(charT c) const;
template <class charT> inline bool isalnum(charT c) const;
template <class charT> inline bool isgraph(charT c) const;

template <class charT> inline charT toupper(charT c) const;
template <class charT> inline charT tolower(charT c) const;

// this template function satisfies requirements for a
//  comparator predicate template argument
template <class charT>
inline bool operator()(const basic_string<charT>& s1,
                       const basic_string<charT>& s2) const;

//********** constructors ****************
// default constructor: the current global locale
locale() : imp_(global_ ? global_ : init()) { imp_->add_ref(); }

locale(const locale& other)  { imp_ = other.imp_; imp_->add_ref(); }

const locale& operator=(const locale& other) const
{
  if (imp_ != other.imp_) {
    imp_->rem_ref(); imp_ = other.imp_; imp_->add_ref();
  }
  return *this;
}

// the following constructor makes a locale composed of "byname"
//  facets, and assigns a name.

locale(const char* std_name);  // using std C locale names, e.g. "POSIX"

typedef unsigned category;  // as defined in <clocale>, e.g. LC_CTYPE

// the following constructor copies its first argument except for a
// specified component, which is constructed on the spot.  if *other*
// has a name, the new locale does also.

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

```
  // the following constructor copies all facets but one from the first
  //  argument, installs the other argument as the remaining facet.
  //  The resulting locale has no name.

  template <class Facet>
  locale(const locale& other, Facet* f)  // to accrete or replace facet
  : imp_(new imp(*other.imp_, 1))
  {
    f->add_ref();
    install(f, Facet::id, "*");
  }

  // the following constructor copies all facets but one from the
  // first argument, and the remaining facet from the second argument.
  // The resulting locale has a name only if both argument locales do.

  template <class Facet>
  locale(const locale& other, const locale& one) // to replace a facet
  : imp_(new imp(*other.imp_, 1))
  {
    facet* f = (Facet*) one.imp_->vec[Facet::id];  // check derivation
    install(f, Facet::id, merge_names(Facet::id, one.imp_->name_));
  }

  ~locale() { imp_->rem_ref(); }  // non-virtual


  // static members:
  static locale global(const locale&);  // replaces ::setlocale(...)
  static const locale& classic();       // the "C" locale
  static const locale transparent()     // continuously updated global locale
    { return locale(new imp(1, 0, true)); }
};

template <class charT>
basic_ostream<charT>& operator<<(basic_ostream<charT>& s, const locale& l)
  { s << l.name() << endl; return s;}

template <class charT>
basic_istream<charT>& operator>>(basic_istream<charT>& s, locale& l);
  // read a line, construct a locale, throw exception if cannot.
```

1 A `locale` constructed from a name string (such as ""), or from parts of two named locales, or read from a stream, has a name; all others do not. Named locales may be compared for equality; an unnamed locale is equal only to (copies of) itself. For an unnamed locale, `locale::name()` returns the string `"*"`.

2 A facet has a dual role: in one sense, it's just a class interface; at the same time, it's an index into a locale's set of facets. Access to the facets of a `locale` is via two member function templates, `locale::use<facet>()` and `locale::has<facet>()`.

3 For example, the standard iostream `operator<<` might be implemented as:

```
  ostream& operator<<(ostream& s, double f)
  {
    locale l = s.rdloc();
    if (s.opfx()) {
      l.use< locale::num_put<char> >().put(s, s, l, f);
      s.osfx();
    }
    return s;
  }
```

In the call to `l.use<...>()`, the type argument chooses a facet, making available all members of the named type. If the named facet is not present in a locale (or, failing that, in the global locale), it throws the standard exception `bad_cast`. You can check if a locale implements a particular facet with the member `has<...>()`. User-defined facets may be bestowed on a locale, and used the same way as standard facets.

4 All locale semantics are accessed via `use<>()` and `has<>()`, with two exceptions. Convenient interfaces are provided for traditional ctype functions such as `isdigit()` and `isspace()`, so that given a locale object *loc* you can say `loc.isspace(c)`. These are provided to ease the conversion of existing extractors. Also, a member template function `operator()(basic_string<T>&, basic_string<T>&)` is provided so that a locale may be used as a predicate argument to the standard collections.

5 The static member `locale::global()`, which returned a snapshot of the current global locale, is replaced by the default constructor `locale()`, which is both shorter to type and useful in more places. The static member `locale::transparent()`, and any locale with similar behavior, is now documented as unsafe to imbue on an iostream or install as the global locale; the effect is undefined. All locales are now semi-transparent (translucent?), in that a locale which does not implement a facet delegates to the global locale.

Box 97
Effects TBS:

22.1.1.1 Type `locale::category` [lib.locale.category]

```
typedef unsigned category;  // as defined in <clocale>, e.g. LC_CTYPE
```

22.1.1.2 `locale` constructors [lib.locale.cons]

```
locale();

locale(const locale& other);

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

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);
```

22.1.1.3 `locale` destructor [lib.locale.des]

```
~locale();
```

22.1.1.4 `locale::use` [lib.locale.use]

```
template <class Facet> const Facet& use() const;
```

22.1.1.5 `locale::has` [lib.locale.has]

```
template <class Facet> bool has() const;
```

22.1.1.6 `locale::name` [lib.locale.name]

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

22.1.1.7 `locale::operator==` [lib.locale.op==]

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

22.1.1.8 `locale::operator!=` [lib.locale.op!=]

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

22.1.1.9 `locale::isspace` [lib.locale.isspace]

```
template <class charT> bool isspace(charT c) const;
```

22.1.1.10 `locale::isprint` [lib.locale.isprint]

```
template <class charT> bool isprint(charT c) const;
```

22.1.1.11 `locale::iscntrl` [lib.locale.iscntrl]

```
template <class charT> bool iscntrl(charT c) const;
```

22.1.1.12 `locale::isupper` [lib.locale.isupper]

```
template <class charT> bool isupper(charT c) const;
```

22.1.1.13 `locale::islower` [lib.locale.islower]

```
template <class charT> bool islower(charT c) const;
```

22.1.1.14 `locale::isalpha` [lib.locale.isalpha]

```
template <class charT> bool isalpha(charT c) const;
```

22.1.1.15 `locale::isdigit` [lib.locale.isdigit]

```
template <class charT> bool isdigit(charT c) const;
```

22.1.1.16 `locale::ispunct` [lib.locale.ispunct]

```
template <class charT> bool ispunct(charT c) const;
```

22.1.1.17 `locale::isxdigit` [lib.locale.isxdigit]

```
template <class charT> bool isxdigit(charT c) const;
```

22.1.1.18 `locale::isalnum` [lib.locale.isalnum]

```
template <class charT> bool isalnum(charT c) const;
```

22.1.1.19 `locale::isgraph` [lib.locale.isgraph]

```
template <class charT> bool isgraph(charT c) const;
```

22.1.1.20 `locale::toupper` [lib.locale.toupper]

```
template <class charT> charT toupper(charT c) const;
```

22.1.1.21 `locale::tolower` [lib.locale.tolower]

```
template <class charT> charT tolower(charT c) const;
```

22.1.1.22 `locale::operator()` [lib.locale.op()]

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

22.1.1.23 `locale::global` [lib.locale.global]

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

22.1.1.24 `locale::classic` [lib.locale.classic]

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

22.1.1.25 `locale::transparent` [lib.locale.transparent]

```
static const locale transparent();
```

22.1.1.26 `operator<<` [lib.locale.op<<]

```
template <class charT>
basic_ostream<charT>& operator<<(basic_ostream<charT>& s, const locale& l)
```

22.1.1.27 `operator>>` [lib.locale.op>>]

```
template <class charT>
basic_ostream<charT>& operator>>(basic_ostream<charT>& s, locale& l)
```

22.1.2 Locale classes [lib.locale.classes]

1 For some standard facets there is a ‘‘`_byname`’’ class derived from it that implements POSIX locale semantics; they are specified by name in the standard to allow users to derive from them. If there is no ‘‘`_byname`’’ version, the base class implements POSIX semantics itself, sometimes with the help of another facet.

22.1.2.1 Class `locale::facet` [lib.locale.facet]

```
// locale::facet -- base class for locale feature sets.
//   Any class deriving from facet must declare a *static* member:
//      static std::locale::id id;
```

```
class locale::facet {
  facet(const facet&);          // not defined
  void operator=(const facet&); // not defined
  void operator&() {}           // not usable

  size_t refcount_;

  //  MT environments must lock during add_ref() and rem_ref().
  void add_ref() { if (this) ++refcount_; }
  void rem_ref() { if (this && refcount_-- == 0) delete this; }

 protected:
  facet(size_t refs = 0) : refcount_(refs-1) { }
  virtual ~facet() {}

  friend class locale;
  friend class imp;
};
```

22.1.2.2 Class `locale::id` [lib.locale.ctype.id]

```
// class id: identification of a locale facet interface,
//  used as an index for lookup.

class locale::id {
  void operator=(const id&); // not defined
  id(const id&);             // not defined
  void operator&() {}        // not usable

  mutable size_t id_;
  static size_t master_id_;  // init'd to 0 by loader
  void init()  //  MT environments must lock during this function.
    { if (!id_) id_ = ++master_id_; }
  operator size_t() const { return id_; }

 public:
  id() {}      // does *not* initialize member id_.
  friend class locale;
};
```

22.2 Standard `locale` facets [lib.std.facets]

22.2.1 The `ctype` facet [lib.facet.ctype]

22.2.1.1 Template class `locale::ctype_base` [lib.locale.ctype.base]

```
struct locale::ctype_base : locale::facet {
  enum ctype {
    SPACE=1<<0, PRINT=1<<1, CNTRL=1<<2, UPPER=1<<3, LOWER=1<<4,
    ALPHA=1<<5, DIGIT=1<<6, PUNCT=1<<7, XDIGIT=1<<8,
    ALNUM=(1<<5)|(1<<6), GRAPH=(1<<7)|(1<<6)|(1<<5)
  };
 protected:
   ctype_base(size_t refs = 0) : facet(refs) {}
  ~ctype_base() {}
};
```

22.2.1.2 Template class `locale::ctype` [lib.locale.ctype]

```
template <class charT>
class locale::ctype : public locale::facet {
public:
  typedef charT char_type;

protected:
  virtual bool do_is(ctype mask, charT c) const;
  virtual const charT* do_is(
            const charT* low, const charT* high, ctype* vec) const;
  virtual const char* do_scan_is(
            ctype mask, const charT* low, const charT* high) const;
  virtual const char* do_scan_not(
            ctype mask, const charT* low, const charT* high) const;
  virtual charT       do_toupper(charT)                         const;
  virtual const charT* do_toupper(charT* low, const charT* high) const;
  virtual charT       do_tolower(charT)                         const;
  virtual const charT* do_tolower(charT* low, const charT* high) const;

  virtual charT        do_widen(char) const ;
  virtual const char*  do_widen(const char* lo,
                                const char* hi, charT* dest) const;
  virtual char         do_narrow(charT, char dfault) const;
  virtual const charT* do_narrow(const charT* lo, const charT* hi,
                                 char dfault, char* dest) const;

public:
  bool         is(ctype mask, charT c) const
                 { return do_is(mask, c); }
  const charT* is(const charT* low, const charT* high, ctype* vec) const
                 { return do_is(low, high, vec); }
  const charT* scan_is(ctype mask, const charT* low, const charT* high) const
                 { return do_scan_is(mask, low, high); }
  const charT* scan_not(ctype mask, const charT* low, const charT* high) const
                 { return do_scan_not(mask, low, high); }

  charT        toupper(charT)              const;
                 { return do_toupper(c); }
  const charT* toupper(charT* low, const charT* high) const
                 { return do_toupper(low, high); }
  charT        tolower(charT c)             const;
                 { return do_tolower(c); }
  const charT* tolower(charT* low, const charT* high) const
                 { return do_tolower(low, high); }

  charT  widen(char c) const { return do_widen(c); }
  const char* widen(const char* lo, const char* hi, charT* to) const
          { return do_widen(lo, hi, to); }
  char   narrow(charT c, char dfault) const { return do_narrow(c, dfault); }
  const charT* narrow(const charT* lo, const charT*
                      char dfault, char* to) const
          { return do_narrow(lo, hi, dfault, to); }

  static locale::id id;

  ctype(size_t refs = 0) : locale::ctype_base(refs) {}
protected:
  ~ctype() {}
};
```

```
template <class charT> locale::id locale::ctype<charT>::id;
```

1 Class `locale::ctype` encapsulates the C library `<cctype>` features. The members of `locale::ctype<charT>` are much as in the previous proposal, with the addition of the `scan_is()` and `scan_not()` functions which find the first character in a buffer that does or does not satisfy a bit-mask criterion. `istream` members are required to use `locale::ctype<>` for character classing.

2 A specialization `locale::ctype<char>` is provided, so that the member functions on type `char` may be implemented inline. Definitions of these functions have been provided for exposition. Only the `char`, and not the `unsigned char` and `signed char` forms, have been provided. The specialization is specified in the standard (and not left as an implementation detail) because it affects the derivation interface for `locale::ctype<char>`.

22.2.2 The `ctype<char>` facet [lib.facet.ctype.char]

22.2.2.1 Class `ctype<char>` [lib.locale.ctype.char]

```
// a specialization, so that char operations may be inline.
// (We must specify this in the standard because it affects the
//  derivation interface)

class locale::ctype<char> : public locale::ctype_base {
public:
  typedef char char_type;

 protected:
  const ctype* const table_;
  static const ctype classic_table_[UCHAR_MAX];

  virtual char        do_toupper(char)                        const;
  virtual const char* do_toupper(char* low, const char* high) const;
  virtual char        do_tolower(char)                        const;
  virtual const char* do_tolower(char* low, const char* high) const;

 public:
  bool is(ctype mask, char c) const
    { return table_[(unsigned char)c] & mask; }
  const char* is(const char* lo, const char* hi, ctype* vec) const
    { while (lo != hi) *vec++ = table_[(unsigned char)*lo++]; return lo; }
  const char* scan_is(ctype mask, const char* low, const char* high) const
  {
    while (low != high && !(table_[(unsigned char) *low++] & mask);
    return low;
  }
  const char* scan_not(ctype mask, const char* low, const char* high) const
  {
    while (low != high && (table_[(unsigned char)*low++] & mask);
    return low;
  }

  char        toupper(char c)               const;
               { return do_toupper(c); }
  const char* toupper(char* low, const char* high) const
               { return do_toupper(low, high); }
  char        tolower(char c)               const;
               { return do_tolower(c); }
  const char* tolower(char* low, const char* high) const
               { return do_tolower(low, high); }
  char  widen(char c) const { return c; }
  const char* widen(const char* lo, const char* hi, char* to) const
           { memcpy(to, lo, hi-lo); return hi; }
  char   narrow(char c, char /*dfault*/) const { return c; }
  const char* narrow(const char* lo, const char* hi,
                     char /*dfault*/, char* to) const
           { memcpy(to, lo, hi-lo); return hi; }

  static locale::id id;

  ctype(const ctype* tab = 0, bool del = false, size_t refs = 0)
    : ctype_base(id, refs), table_(tab ? tab : classic_table_),
      delete_it_(tab ? del : false) {}
protected:
  ~ctype() { if (delete_it) delete table_; }
};

locale::id locale::ctype<char>::id;
```

22.2.2.2 Template class `ctype_byname` [lib.locale.ctype.byname]

```
template <class charT>
class locale::ctype_byname : public locale::ctype<charT> {
  // this class is specialized by vendors for char and wchar_t.
 protected:
  // ... declarations for all virtuals.
 public:
  ctype_byname(const char*, size_t refs = 0);
 protected:
  ~ctype_byname();
};
```

22.2.2.3 Convenience interfaces [lib.locale.ctype.convenience]

```
template <class charT> inline bool locale::isspace(charT c) const
  { return use<locale::ctype<charT> >().is(locale::ctype<charT>::SPACE, c); }
template <class charT> inline bool locale::isprint(charT c) const
  { return use<locale::ctype<charT> >().is(locale::ctype<charT>::PRINT, c); }
template <class charT> inline bool locale::iscntrl(charT c) const
  { return use<locale::ctype<charT> >().is(locale::ctype<charT>::CNTRL, c); }
template <class charT> inline bool locale::isupper(charT c) const
  { return use<locale::ctype<charT> >().is(locale::ctype<charT>::UPPER, c); }
template <class charT> inline bool locale::islower(charT c) const
  { return use<locale::ctype<charT> >().is(locale::ctype<charT>::LOWER, c); }
template <class charT> inline bool locale::isalpha(charT c) const
  { return use<locale::ctype<charT> >().is(locale::ctype<charT>::ALPHA, c); }
template <class charT> inline bool locale::isdigit(charT c) const
  { return use<locale::ctype<charT> >().is(locale::ctype<charT>::DIGIT, c); }
template <class charT> inline bool locale::ispunct(charT c) const
  { return use<locale::ctype<charT> >().is(locale::ctype<charT>::PUNCT, c); }
template <class charT> inline bool locale::isxdigit(charT c) const
  { return use<locale::ctype<charT> >().is(locale::ctype<charT>::XDIGIT, c); }
template <class charT> inline bool locale::isalnum(charT c) const
  { return use<locale::ctype<charT> >().is(locale::ctype<charT>::ALNUM, c); }
template <class charT> inline bool locale::isgraph(charT c) const
  { return use<locale::ctype<charT> >().is(locale::ctype<charT>::GRAPH, c); }

template <class charT> inline charT locale::toupper(charT c) const
  { return use<locale::ctype<charT> >().toupper(c); }
template <class charT> inline charT locale::tolower(charT c) const
  { return use<locale::ctype<charT> >().tolower(c); }
```

22.2.3 The numeric facet [lib.facet.numeric]

22.2.3.1 Template class `num_get` [lib.locale.num.get]

```
template <class charT, class InputIterator = istreambuf_iterator<charT> >
class locale::num_get : public locale::facet {
public:
  typedef charT         char_type;
  typedef InputIterator iter_type;
  typedef basic_ios<charT> ios;

  // members of locale::num_get take a locale argument because they
  //   may need to refer to the locale’s numpunct facet.

protected:
  virtual iter_type do_get(iter_type, ios&, const locale&,
                           bool& v)          const;
  virtual iter_type do_get(iter_type, ios&, const locale&,
                           long& v)          const;
  virtual iter_type do_get(iter_type, ios&, const locale&,
                           unsigned long& v) const;
//  virtual iter_type do_get(iter_type, ios&, const locale&,
//                           long long& v)   const;
  virtual iter_type do_get(iter_type, ios&, const locale&,
                           double& v)        const;
  virtual iter_type do_get(iter_type, ios&, const locale&,
                           long double& v)   const;
public:
  iter_type get(iter_type s, ios& f, const locale&, bool& v)          const;
    { return this->do_get(s, f, l, v); }
  iter_type get(iter_type s, ios& f, const locale&, long& v)          const;
    { return this->do_get(s, f, l, v); }
  iter_type get(iter_type s, ios& f, const locale&, unsigned long& v) const;
    { return this->do_get(s, f, l, v); }
//  iter_type get(iter_type s, ios& f, const locale&, long long& v)   const;
//    { return this->do_get(s, f, l, v); }
  iter_type get(iter_type s, ios& f, const locale&, double& v)        const;
    { return this->do_get(s, f, l, v); }
  iter_type get(iter_type s, ios& f, const locale&, long double& v)   const;
    { return this->do_get(s, f, l, v); }

  static locale::id id;

  num_get(size_t refs = 0) : locale::facet(refs) {}
protected:
  ~num_get() {}
};

template <class charT, class InputIterator = istreambuf_iterator<charT> >
locale::id locale::num_get<charT, InputIterator>::id;
```

1 The classes `locale::num_get<>` and `num_put<>` handle numeric formatting and parsing. Virtual functions are provided for several numeric types; implementations are allowed to delegate conversion of smaller types to extractors for larger ones, but are not required to. The functions take a `locale` argument because their base class implementation refers to `locale::numpunct` features, which identify preferred numeric punctuation. Extractors and inserters for the standard iostreams are required to call `locale::num_get` and `num_put` member functions. The `ios&` argument is used both for format control and to report errors.

22.2.3.2 Template class `num_put` [lib.locale.num.put]

```
template <class charT, class OutputIterator = ostreambuf_iterator<charT> >
class locale::num_put : public locale::facet {
  typedef charT       char_type;
  typedef OutputIterator iter_type;
  typedef basic_ios<charT> ios;

  // Members of locale::num_put take a locale argument because they
  //   may need to refer to the locale's numpunct facet.  The ios&
  //   argument is used for formatting reference only and error reporting.

protected:
  virtual iter_type do_put(iter_type, ios&, const locale&, bool v)      const;
  virtual iter_type do_put(iter_type, ios&, const locale&, long v)      const;
  virtual iter_type do_put(iter_type, ios&, const locale&, unsigned long)
    const;
//virtual iter_type do_put(iter_type, ios&, const locale&, long long v) const;
  virtual iter_type do_put(iter_type, ios&, const locale&, double v)    const;
  virtual iter_type do_put(iter_type, ios&, const locale&, long double v)
    const;
public:
  iter_type put(iter_type s, ios& f, const locale& l, bool v)          const
    { return this->do_put(s, f, l, v); }
  iter_type put(iter_type s, ios& f, const locale& l, long v)          const
    { return this->do_put(s, f, l, v); }
  iter_type put(iter_type s, ios& f, const locale& l, unsigned long v) const
    { return this->do_put(s, f, l, v); }
//  iter_type put(iter_type s, ios& f, const locale& l, long long v)   const
//    { return this->do_put(s, f, l, v); }
  iter_type put(iter_type s, ios& f, const locale& l, double v)        const
    { return this->do_put(s, f, l, v); }
  iter_type put(iter_type s, ios& f, const locale& l, long double v)   const
    { return this->do_put(s, f, l, v); }

  static locale::id id;

  num_put(size_t refs = 0) : locale::facet(refs) {}
protected:
  ~num_put() {}
};

template <class charT, class OutputIterator = ostreambuf_iterator<charT> >
locale::id locale::num_put<charT, OutputIterator>::id;
```

22.2.4 The numeric punctuation facet [lib.facet.numpunct]

22.2.4.1 Template class numpunct [lib.locale.numpunct]

```
// locale::numpunct is used by locale::num_get and num_put facets.

template <class charT>
class locale::numpunct : public locale::facet {
public:
  typedef charT char_type;
  typedef basic_string<charT> string;

protected:
  virtual string do_decimal_point() const;
  virtual string do_thousands_sep() const;
  virtual vector<char>  do_grouping() const;
  virtual string  do_truename() const;  // for bool
  virtual string  do_falsename() const; // for bool

public:
  string decimal_point()   const { return do_decimal_point(); }
  string thousands_sep()   const { return do_thousands_sep(); }
  vector<char>  grouping() const { return do_grouping(); }
  string truename()        const { return do_truename(); }
  string falsename()       const { return do_falsename(); }

  static locale::id id;

  numpunct(size_t refs = 0) : locale::facet(refs) {}
protected:
  ~numpunct() {}
};

template <class charT> locale::id locale::numpunct<charT>::id;
```

1 `numpunct<>` specifies numeric punctuation. The base class provides classic "C" numeric formats, while the `_byname` version supports general POSIX numeric formatting semantics.

22.2.4.2 Template class `numpunct_byname` [lib.locale.numpunct.byname]

```
template <class charT>
class locale::numpunct_byname : public locale::numpunct<charT> {
  // this class is specialized by vendors for char and wchar_t.
protected:
  // ... declarations for all virtuals.
public:
  numpunct_byname(const char*, size_t refs = 0);
protected:
  ~numpunct_byname();
};
```

22.2.5 The collation facet [lib.facet.collate]

22.2.5.1 Template class `collate` [lib.locale.collate]

```
// locale.use<locale::collate>() is used for string comparisons.

template <class charT>
class locale::collate : public locale::facet {
public:
  typedef charT              char_type;
  typedef basic_string<charT> string;
protected:
  virtual int    do_collate(const char* low1, const char* high1,
                            const char* low2, const char* high2);;
  virtual string do_transform(const char* low, const char* high);
  virtual long   do_hash(     const char* low, const char* high);

public:
  int collate(const char* low1, const char* high1,
              const char* low2, const char* high2);;
  string transform(const char* low, const char* high);
  long hash(const char* low, const char* high);

  static locale::id id;

  collate(size_t refs = 0) : locale::facet(refs) {}
protected:
  ~collate() {}
};

template <class charT> locale::id locale::collate<charT>::id;
```

1 The class `locale::collate<charT>` provides features for use in the collation of strings. A locale member function template, `operator()`, uses the collate facet to allow a locale to act directly as the predicate argument for algorithms. The base class uses lexicographic ordering.

22.2.5.2 Template class `collate_byname` [lib.locale.collate.byname]

```
template <class charT>
class locale::collate_byname : public locale::collate<charT> {
  // this class is specialized by vendors for char and wchar_t.
protected:
  // ... declarations for all virtuals.
public:
  collate_byname(const char*);
protected:
  ~collate_byname();
};

//  this template function satisfies requirements for a
//  comparator predicate template argument:

template <class charT>
inline bool locale::operator()(const basic_string<charT>& s1,
                               const basic_string<charT>& s2);
{
   return use< collate<charT> >().collate(s1.begin(), s1.end(),
                                          s2.begin(), s2.end()) < 0;
}
```

22.2.6 The codeset conversion facet [lib.facet.codecvt]

22.2.6.1 Template class codecvt [lib.locale.codecvt]

```
template <class fromT, class toT, class stateT>
class locale::codecvt : public locale::facet {
  // use two of these to convert both ways
public:
  typedef fromT  from_type;
  typedef toT    to_type;
  typedef stateT state_type;

  enum result { ok = 0, partial, error }
protected:
  virtual result do_convert(stateT& state,
    const fromT* from, const fromT* from_end, const fromT*& from_next,
          toT*   to,   toT*         to_limit,       toT*&   to_next);

public:
   result convert(stateT& state,
     const fromT* from, const fromT* from_end, const fromT*& from_next,
            toT*   to,         toT* to_limit,         toT*& to_next);

  static locale::id id;

  codecvt(size_t refs = 0) : locale::facet(refs) {}
protected:
  ~codecvt() {}
};

template <class fromT, class toT, class stateT>
locale::id locale::codecvt<fromT, toT, stateT>::id;
```

1 The class `locale::codecvt<fromT, toT, stateT>` is for use when converting from one codeset to another, such as from wide characters to multibyte characters, or between wide character sets such as Unicode and EUC. Instances of this facet are typically used in pairs.

2 Its only member function, `convert()`, returns an enumeration value which indicates whether it completed the conversion (ok), ran out of space in the destination (partial), or encountered a "from" character it could not convert (error).

3 In all cases it leaves the `from_next` and `to_next` pointers pointing one beyond the last character successfully converted.

4 The `stateT` argument selects the pair of codesets being mapped between. Base class members are pure virtual.

5 Implementations are obliged to provide specializations for `<char, wchar_t, mbstate_t>` and `<wchar_t, char, mbstate_t>`.

22.2.6.2 Template class `codecvt_byname` [lib.locale.codecvt.byname]

```
template <class fromT, class toT, class stateT>
class locale::codecvt_byname : public locale::codecvt<fromT, toT, stateT> {
 protected:
  // ... declarations for all virtuals.
 public:
  codecvt_byname(const char*, size_t refs = 0);
 protected:
  ~codecvt_byname();
};
```

22.2.7 The date and time facet [lib.facet.date.time]

1 The classes `locale::time_get<charT>` and `time_put<charT>` provide date and time formatting and parsing. The time formatting function `put()` takes an extra format argument to accommodate the POSIX `strftime()` extensions. The `ios&` argument is used for format control and to report errors.

22.2.7.1 Template class `time_get` [lib.locale.time.get]

```
template <class charT, class InputIterator = istreambuf_iterator<charT> >
class locale::time_get : public locale::facet {
public:
  typedef charT       char_type;
  typedef InputIterator iter_type;
  typedef basic_ios<charT> ios;

  enum dateorder { NO_ORDER, DMY, MDY, YMD, YDM };

protected:
  virtual dateorder do_date_order()  const;

  virtual iter_type do_get_time(iter_type s, ios&, const locale&, tm* t) const;
  virtual iter_type do_get_date(iter_type s, ios&, const locale&, tm* t) const;
  virtual iter_type do_get_weekday(iter_type s, ios&, const locale&, tm* t)
                    const;
  virtual iter_type do_get_monthname(iter_type s, ios&, const locale&, tm* t)
                    const;
  virtual iter_type do_get_year(iter_type s, ios&, const locale&, tm* t) const;
public:
  dateorder date_order()  const { return do_date_order(); }
  iter_type get_time(iter_type s, ios& f, const locale& l, tm* t)  const
    { return do_get_time(s,f,l,t); }
  iter_type get_date(iter_type s, ios& f, const locale& l, tm* t)  const
    { return do_get_date(s,f,l,t); }
  iter_type get_weekday(iter_type s, ios& f, const locale& l, tm* t) const
    { return do_get_weekday(s,f,l,t); }
  iter_type get_monthname(iter_type s, ios& f, const locale& l, tm* t) const
    { return do_get_monthname(s,f,l,t); }
  iter_type get_year(iter_type s, ios& f, const locale& l, tm* t) const
    { return do_get_year(s,f,l,t); }

  static locale::id id;

  time_get(size_t refs = 0) : locale::facet(refs) {}
protected:
  ~time_get() {}
};

template <class charT, class InputIterator = istreambuf_iterator<charT> >
locale::id locale::time<charT, InputIterator>::id;
```

22.2.7.2 Template class `time_get_byname` [lib.locale.time.get.byname]

```
template <class charT, class InputIterator = istreambuf_iterator<charT> >
class locale::time_get_byname : public locale::time_get<charT, InputIterator> {
protected:
  // ... declarations for all virtuals.
public:
  time_get_byname(const char*, size_t refs = 0);
protected:
  ~time_get_byname();
};
```

22.2.7.3 Template class `time_put` [lib.locale.time.put]

```
template <class charT, class OutputIterator = ostreambuf_iterator<charT> >
class locale::time_put : public locale::facet {
public:
  typedef charT          char_type;
  typedef OutputIterator iter_type;
  typedef basic_ios<charT> ios;

protected:
  virtual iter_type do_put(iter_type s, ios&, const locale&, const tm* t,
                      char format, char modifier) const;
public:
  // the following is implemented in terms of other member functions.
  iter_type put(iter_type s, ios& f, struct tm const* tmb,
           const charT* pattern, const charT* pat_end) const;

  iter_type put(iter_type s, ios& f, const locale& l, struct tm const* t,
           char format, char modifier = ' ') const
    { return do_put(s,f,l,tmb,format,modifier); }

  static locale::id id;

  time_put(size_t refs = 0) : locale::facet(refs) {}
protected:
  ~time_put() {}
};

template <class charT, class OutputIterator = ostreambuf_iterator<charT> >
locale::id locale::time_put<charT, OutputIterator>::id;
```

22.2.7.4 Template class `time_put_byname` [lib.locale.time.put.byname]

```
template <class charT, class OutputIterator = ostreambuf_iterator<charT> >
class locale::time_put_byname : public locale::time_put<charT, OutputIterator>
{
protected:
  // ... declarations for all virtuals.
public:
  time_put_byname(const char*, size_t refs = 0);
protected:
  ~time_put_byname();
};
```

22.2.8 The money facet [lib.facet.money]

1 These handle money formats. A template parameter indicates whether local or international monetary formats are to be used. `money_get<>` and `money_put<>` use `locale::moneypunct<>` features if appropriate. `locale::moneypunct<>` provides basic format information for money processing. The `ios&` argument is used for format control and to report errors.

22.2.8.1 Template class `money_get` [lib.locale.money.get]

```
template <class charT, bool Intl = false,
          class InputIterator = istreambuf_iterator<charT> >
class locale::money_get : public locale::facet {
  typedef charT       char_type;
  const bool          intl = Intl;
  typedef InputIterator iter_type;
  typedef basic_string<charT> string;
  typedef basic_ios<charT> ios;

protected:
  virtual iter_type do_get(iter_type, ios&, const locale&, double& units) const;
  virtual iter_type do_get(iter_type, ios&, const locale&, string& digits) const;
public:
  iter_type get(iter_type s, ios& f, const locale& l, double& units) const
     { return do_get(s, f, l, units); }
  iter_type get(iter_type s, ios& f, const locale& l, string& digits) const
     { return do_get(s, f, l, digits); }

  static locale::id id;

  money_get(size_t refs = 0) : locale::facet(refs) {}
 protected:
  ~money_get() {}
};

template <class charT, bool Intl = false,
          class InputIterator = istreambuf_iterator<charT> >
locale::id locale::money_get<charT, Intl, InputIterator>::id;
```

22.2.8.2 Template class money_put [lib.locale.money.put]

```
template <class charT, bool Intl = false,
          class OutputIterator = ostreambuf_iterator<charT> >
class locale::money_put : public locale::facet {
public:
  typedef charT       char_type;
  const bool          intl = Intl;
  typedef OutputIterator iter_type;
  typedef basic_string<charT> string;
  typedef basic_ios<charT> ios;

protected
  virtual iter_type do_put(iter_type, ios&, const locale&, double units) const;
  virtual iter_type do_put(iter_type, ios&, const locale&, const string& digits) const;

public:
  iter_type put(iter_type s, ios& f, const locale& l, double units) const
     { return do_put(s, f, l, units); }
  iter_type put(iter_type s, ios& f, const locale& l, const string& dgts) const
     { return do_put(s, f, l, dgts); }

  static locale::id id;

  money_put(size_t refs = 0) : locale::facet(refs) {}
protected:
  ~money_put() {}
};

template <class charT, bool Intl = false,
          class OutputIterator = ostreambuf_iterator<charT> >
locale::id locale::money_put<charT, Intl, OutputIterator>::id;
```

22.2.9 The money punctuation facet [lib.facet.moneypunct]

22.2.9.1 Template class moneypunct [lib.locale.moneypunct]

```
template <class charT, bool International = false>
class locale::moneypunct : public locale::facet {
public:
  typedef charT char_type;
  const bool intl = International;
  typedef basic_string<charT> string;

  enum part { SYMBOL='$', SIGN='-', SPACE=' ', VALUE='v', NONE=' ' };
  struct pattern { char field[4]; };

protected:
  virtual charT        do_decimal_point()  const;
  virtual charT        do_thousands_sep()  const;
  virtual vector<char> do_grouping()       const;
  virtual string       do_curr_symbol()    const;
  virtual string       do_positive_sign()  const;
  virtual string       do_negative_sign()  const;
  virtual int          do_frac_digits()    const;
  virtual pattern      do_format()         const;

public:
  charT        decimal_point() const  { return do_decimal_point(); }
  charT        thousands_sep() const  { return do_thousands_sep(); }
  vector<char> grouping()      const  { return do_grouping(); }
  string       curr_symbol()   const  { return do_curr_symbol(); };
  string       positive_sign() const  { return do_positive_sign; }
  string       negative_sign() const  { return do_negative_sign; }
  int          frac_digits()   const  { return do_frac_digits(); }
  pattern      format()        const  { return do_format(); }

  static locale::id id;

  moneypunct(size_t refs = 0) : locale::facet(refs) {}
protected:
  ~moneypunct() {}
};

template <class charT, bool Intl = false>
locale::id locale::moneypunct<charT, Intl>::id;
```

1 This provides money punctuation, similar to numpunct above.

22.2.9.2 Template class `moneypunct_byname` [lib.locale.moneypunct.byname]

```
template <class charT, bool Intl = false>
class locale::moneypunct_byname : public locale::moneypunct<charT, Intl> {
 protected:
  // ... declarations for all virtuals.
 public:
  moneypunct_byname(const char*, size_t refs = 0);
 protected:
  ~moneypunct_byname();
};
```

22.2.10 The message retrieval facet [lib.facet.messages]

1 Class locale::msg<charT> implements POSIX message retrieval. It should be flexible enough to retrieve messages from X, MS Windows, or Macintosh resource files as well.

22.2.10.1 Template class msg [lib.locale.msg]

```
template <class charT>
class locale::msg {
public:
  typedef charT char_type;
  typedef int   catalog;
  typedef basic_string<charT> string;

protected:
  virtual catalog do_open(const string&, const locale&) const;
  virtual string  do_get(catalog, int set, int msgid,
                         const string& default)         const;
  virtual void    do_close(catalog) const;

public:
  catalog open(const string& fn, const locale& l) const
    { return do_open(fn, l); }
  string get(catalog c, int set, int msgid, const string& default) const
    { return do_get(c, set, msgid, default); }
  void close(catalog c) const { do_close(c); }

  static locale::id id;

  msg(size_t refs = 0) : locale::facet(refs) {}
protected:
  ~msg() {}
};

template <class charT> locale::id locale::msg<charT>::id;
```

22.2.10.2 Template class msg_byname [lib.locale.msg.byname]

```
template <class charT>
class locale::msg_byname : public locale::msg<charT> {
protected:
  // ... declarations for all virtuals.
public:
  msg_byname(const char*, size_t refs = 0);
protected:
  ~msg_byname();
};
```

22.2.11 User-defined facets [lib.facets.examples]

1 A user-defined facet may be added to a locale and used identically as the built-in facets. To create a new facet interface users simply derive, from `locale::facet`, a class containing a static member: `static locale::id id;`. (The locale member function templates verify its type and storage class.)

2 For those curious about the mechanics: this initialization/identification system depends only on the initialization to 0 of static objects, before static constructors are called. When an instance of a facet is installed in a locale, the locale checks whether an id has been assigned, and if not, assigns one. Before this occurs, any attempted use of its interface causes the `bad_cast` exception to be thrown.

3 Here is a program that just calls C functions:

```
#include <locale>
extern "C" void c_function();
int main()
{
  using namespace std;
  locale::global(locale(""));  // same as setlocale(LC_ALL, "");

  c_function();
  return 0;
}
```

4 In other words, C library localization is unaffected.

5 Traditional global localization is still easy:

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

int main(int argc, char** argv)
{
  using namespace std;
  locale::global(locale(""));  // set the global locale
   cin.imbue(locale());        // imbue it on the std streams
  cout.imbue(locale());
  cerr.imbue(locale());
  return MyObject(argc, argv).doit();
}
```

6 Greater flexibility is possible:

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

int main()
{
  using namespace std;
  cin.imbue(locale(""));  // the user's preferred locale
  cout.imbue(locale::classic());

  double f;
  while (cin >> f) cout << f << endl;
  return (cin.fail() != 0);
}
```

7 In a European locale, with input `3.456,78`, output is `3456.78`. This can be important even for simple programs, which may need to write a data file in a fixed format, regardless of a user's preference.

8 Here is an example:

```
// file: Date.h
#include <locale>
   ...
class Date {
  ...
 public:
  Date(unsigned day, unsigned month, unsigned year);
  std::string asString(const std::locale& = std::locale());
};

istream& operator>>(istream& s, Date& d);
ostream& operator<<(ostream& s, Date d);
...
```

9 This example illustrates two architectural uses of class `locale`. The first is as a default argument in `Date::asString()`, where the default is the global (presumably user-preferred) locale. The second is in the operators `<<` and `>>`, where a locale ‘‘hitchhikes’’ on another object, in this case a stream, to the point where it is needed.

```
// file: Date.C
#include <Date>
#include <stringstream>

std::string Date::asString(const std::locale& l)
{
  using namespace std;
  stringstream s; s.imbue(l);
  s << *this; return s.data();
}

std::istream& operator>>(std::istream& s, Date& d)
{
  using namespace std;
  if (!s.ipfx(0)) return s;
  locale loc = s.rdloc();
  struct tm t;
  loc.use<locale::time_get<char> >().get_date(s, s, loc, &t);
  if (s) d = Date(t.tm_day, t.tm_mon + 1, t.tm_year + 1900);
  s.isfx();
  return s;
}
```

10 A locale object may be extended with a new facet simply by constructing it with an instance of a class derived from `locale::facet`. The only member you must define is the static member `id`, which identifies your class interface as a new facet. For example, imagine we want to classify Japanese characters:

```
// file: jctype.h
#include <locale>
namespace My {
  using namespace std;

  class JCtype : public locale::facet {
   public:
    static locale::id id;  // required for use as a new locale facet

    bool is_kanji(wchar_t c);

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

```
// file: filt.C
#include <iostream>
#include <locale>
#include <jctype> // above

std::locale::id JCtype::id;  // the static JCtype member declared above.

int main()
{
  using namespace std;
  typedef locale::ctype<wchar_t> ctype;

  locale loc(locale(""),       // the user’s preferred locale ...
             new My::JCType);  // and a new feature...

  wchar_t c = loc.use<ctype>().widen(’!’);
  if (loc.use<My::JCType>().is_kanji(c))
    cout << "no it isn’t!" << endl;
  return 0;
}
```

11 The new facet is used exactly like the built-in facets.

12 Replacing an existing facet is even easier. Here we do not define a member `id` because we are reusing the `locale::numpunct<charT>` facet interface:

```
// my_bool.C
#include <iostream>
#include <locale>
#include <string>

namespace My {
  using namespace std;
  typedef locale::numpunct_byname<char> numpunct;

  class BoolNames : public numpunct {
    typedef basic_string<char> string;
   protected:
    string do_truename()  { return "Oui Oui!"; }
    string do_falsename() { return "Mais Non!"; }
    ~BoolNames() {}
   public:
    BoolNames(const char* name) : numpunct(name) {}
  };
}

int main(int argc, char** argv)
{
  using namespace std;
  // make the user’s preferred locale, except for...
  locale loc(locale(""), new My::BoolNames(""));
  cout.imbue(loc);
  cout << "Any arguments today? " << (argc > 1) << endl;
  return 0;
}
```

22.3 C Library Locales [lib.c.locales]

1 Headers:

— `<clocale>`

2 Table 79:

Table 79—Header `<clocale>` synopsis

Type	Name(s)		
Macros:			
	`LC_MONETARY`	`LC_NUMERIC`	`LC_TIME`
Struct:	`lconv`		
Functions:	`localeconv`	`setlocale`	

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 components for sequences (23.1) and associative containers (23.2).

23.1 Sequences [lib.sequences]

1 Headers:

— `<bits>`

— `<bitstring>`

— `<dynarray>`

— `<ptrdynarray>`

— `<stl containers (TBD)>`

2 Table 80:

Table 80—Header `<bits>` synopsis

Type	Name(s)	
Template class:	`bits`	
Operator functions:		
`operator& (bits)`	`operator>> (bits)`	`operator| (bits)`
`operator<< (bits)`	`operator^ (bits)`	

3 Table 81:

Table 81—Header `<bitstring>` synopsis

Type	Name(s)
Class:	`bitstring`
Operator functions:	
`operator& (bit_string)`	`operator>> (bit_string)`
`operator+ (bit_string)`	`operator^ (bit_string)`
`operator<< (bit_string)`	`operator| (bit_string)`

4 Table 82:

Table 82—Header `<dynarray>` synopsis

Type	Name(s)
Template class:	`dyn_array`
Operator function:	`operator+  (dyn_array) [3]`

5 Table 83:

Table 83—Header `<ptrdynarray>` synopsis

Type	Name(s)
Class:	`ptr_dyn_array`
Operator function:	`operator+  (ptr_dyn_array) [3]`

6 Table 84:

Table 84—Header `<stl containers (TBD)>` synopsis

Type	Name(s)		
Template classes:			
`deque`	`multimap`	`queue`	`vector`
`list`	`multiset`	`set`	
`map`	`priority_queue`	`stack`	
Template operators:			
`operator<  (deque)`	`operator<  (vector)`	`operator== (queue)`	
`operator<  (list)`	`operator== (deque)`	`operator== (set)`	
`operator<  (map)`	`operator== (list)`	`operator== (stack)`	
`operator<  (multimap)`	`operator== (map)`	`operator== (vector)`	
`operator<  (multiset)`	`operator== (multimap)`		
`operator<  (set)`	`operator== (multiset)`		
Class:	`vector<bool,allocator>`		
Operator functions:	`operator<  (vector<bool,allocator>)`		
	`operator== (vector<bool,allocator>)`		

23.1.1 Template class `bits` [lib.template.bits]

1 The header `<bits>` defines a template class and several related functions for representing and manipulating fixed-size sequences of bits.

```
template<size_t N> class bits {
public:
    bits();
    bits(unsigned long val);
    bits(const string& str, size_t pos = 0, size_t n = NPOS);
    bits<N>& operator&=(const bits<N>& rhs);
    bits<N>& operator|=(const bits<N>& rhs);
    bits<N>& operator^=(const bits<N>& rhs);
    bits<N>& operator<<=(size_t pos);
    bits<N>& operator>>=(size_t pos);
    bits<N>& set();
    bits<N>& set(size_t pos, int val = 1);
    bits<N>& reset();
    bits<N>& reset(size_t pos);
    bits<N>  operator~() const;
    bits<N>& toggle();
    bits<N>& toggle(size_t pos);

    unsigned short to_ushort() const;
    unsigned long  to_ulong() const;
    string to_string() const;
    size_t count() const;
    size_t length() const;
    bool operator==(const bits<N>& rhs) const;
    bool operator!=(const bits<N>& rhs) const;
    bool test(size_t pos) const;
    bool any() const;
    bool none() const;
    bits<N> operator<<(size_t pos) const;
    bits<N> operator>>(size_t pos) const;
private:
//  char array[N];       exposition only
};
```

2 The template class `bits<N>` describes an object that can store a sequence consisting of a fixed number of bits, *N*.

3 Each bit represents either the value zero (reset) or one (set). To *toggle* a bit is to change the value zero to one, or the value one to zero. Each bit has a non-negative position *pos*. When converting between an object of class `bits<N>` and a value of some integral type, bit position *pos* corresponds to the *bit value* `1 << pos`. The integral value corresponding to two or more bits is the sum of their bit values.

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

— `char array[N]`, the sequence of bits, stored one bit per element.[107)]

5 The functions described in this subclause can report three kinds of errors, each associated with a distinct exception:

— an *invalid-argument* error is associated with exceptions of type `invalid_argument`;

— an *out-of-range* error is associated with exceptions of type `out_of_range`;

— an *overflow* error is associated with exceptions of type `overflow_error`.

23.1.1.1 `bits` constructors [lib.cons.bits]

```
    bits();
```

1 Constructs an object of class `bits<N>`, initializing all bits to zero.

```
bits(unsigned long val);
```

107) An implementation is free to store the bit sequence more efficiently.

2 Constructs an object of class `bits<N>`, initializing the first *`M`* bit positions to the corresponding bit values in *`val`*. *`M`* is the smaller of *`N`* and the value `CHAR_BIT * sizeof (unsigned long)`.

3 The macro `CHAR_BIT` is defined in `<climits>`

4 If *`M < N`*, remaining bit positions are initialized to zero.

```
bits(const string& str, size_t pos = 0, size_t n = NPOS);
```

5 Throws `out_of_range` if *`pos > str.len`*. Otherwise, the function determines the effective length *`rlen`* of the initializing string as the smaller of *`n`* and *`str.len - pos`*.

6 The function then throws `invalid_argument` if any of the *`rlen`* characters in *`str`* beginning at position *`pos`* is other than `0` or `1`.

7 Otherwise, the function constructs an object of class `bits<N>`, initializing the first *`M`* bit positions to values determined from the corresponding characters in the string *`str`*. *`M`* is the smaller of *`N`* and *`rlen`*.

8 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.

9 If *`M < N`*, remaining bit positions are initialized to zero.

23.1.1.2 `bits::operator&=` [lib.bits::op&=.bt]

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

1 Clears each bit in `*this` for which the corresponding bit in *`rhs`* is clear, and leaves all other bits unchanged.

2 Returns `*this`.

23.1.1.3 `bits::operator|=` [lib.bits::op|=.bt]

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

1 Sets each bit in `*this` for which the corresponding bit in *`rhs`* is set, and leaves all other bits unchanged.

2 Returns `*this`.

23.1.1.4 `bits::operator^=` [lib.bits::opˆ=.bt]

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

1 Toggles each bit in `*this` for which the corresponding bit in *`rhs`* is set, and leaves all other bits unchanged.

2 Returns `*this`.

23.1.1.5 `bits::operator<<=` [lib.bits::op.lsh=]

```
bits<N>& operator<<=(size_t pos);
```

1 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`*.

2 Returns `*this`.

23.1.1.6 `bits::operator>>=` [lib.bits::op.rsh=]

```
bits<N>& operator>>=(size_t pos);
```

1 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`.

2 Returns `*this`.

23.1.1.7 `bits::set` [lib.bits::set]

```
bits<N>& set();
```

1 Sets all bits in `*this`.

2 Returns `*this`.

```
bits<N>& set(size_t pos, int val = 1);
```

3 Throws `out_of_range` if *pos* does not correspond to a valid bit position. Otherwise, the function 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.

4 Returns `*this`.

23.1.1.8 `bits::reset` [lib.bits::reset]

```
bits<N>& reset();
```

1 Resets all bits in `*this`.

2 Returns `*this`.

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

3 Throws `out_of_range` if *pos* does not correspond to a valid bit position. Otherwise, the function resets the bit at position *pos* in `*this`.

4 Returns `*this`.

23.1.1.9 `bits::operator~` [lib.bits::op˜]

```
bits<N> operator~() const;
```

1 Constructs an object *x* of class `bits<N>` and initializes it with `*this`.

2 Returns *x*`.toggle()`.

23.1.1.10 `bits::toggle` [lib.bits::toggle]

```
bits<N>& toggle();
```

1 Toggles all bits in `*this`.

2 Returns `*this`.

```
bits<N>& toggle(size_t pos);
```

3 Throws `out_of_range` if *pos* does not correspond to a valid bit position. Otherwise, the function toggles the bit at position *pos* in `*this`.

4 Returns `*this`.

23.1.1.11 `bits::to_ushort` [lib.bits::to.ushort]

```
unsigned short to_ushort() const;
```

1 If the integral value *x* corresponding to the bits in `*this` cannot be represented as type `unsigned short`, throws `overflow_error`.

2 Otherwise, returns *x*.

23.1.1.12 `bits::to_ulong` [lib.bits::to.ulong]

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

1 If the integral value *x* corresponding to the bits in `*this` cannot be represented as type `unsigned long`, throws `overflow_error`.

2 Otherwise, returns *x*.

23.1.1.13 `bits::to_string` [lib.bits::to.string]

```
string to_string() const;
```

1 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`.

2 Returns the created object.

23.1.1.14 `bits::count` [lib.bits::count]

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

1 Returns a count of the number of bits set in `*this`.

23.1.1.15 `bits::length` [lib.bits::length]

```
size_t length() const;
```

1 Returns *N*.

23.1.1.16 `bits::operator==` [lib.bits::op==.bt]

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

1 Returns a nonzero value if the value of each bit in `*this` equals the value of the corresponding bit in *rhs*.

23.1.1.17 `bits::operator!=` [lib.bits::op!=.bt]

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

1 Returns a nonzero value if `!(*this ==` *rhs*`)`.

23.1.1.18 `bits::test` [lib.bits::test]

```
bool test(size_t pos) const;
```

1 Throws `out_of_range` if *pos* does not correspond to a valid bit position.

2 Otherwise, returns a nonzero value if the bit at position *pos* in `*this` has the value one.

23.1.1.19 `bits::any` [lib.bits::any]

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

1 Returns a nonzero value if any bit in `*this` is one.

23.1.1.20 `bits::none` [lib.bits::none]

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

1 Returns a nonzero value if no bit in `*this` is one.

23.1.1.21 `bits::operator<<` [lib.bits::op.lsh]

```
bits<N> operator<<(size_t pos) const;
```

1 Returns `bits<N>(*this) <<=` *pos*.

23.1.1.22 `bits::operator>>` [lib.bits::op.rsh]

```
bits<N> operator>>(size_t pos) const;
```

1 Returns `bits<N>(*this) >>=` *pos*.

23.1.1.23 `operator&` [lib.op&.bt.bt]

```
bits<N> operator&(const bits<N>& lhs, const bits<N>& rhs);
```

1 Returns `bits<N>(`*lhs*`) &=` *pos*.

23.1.1.24 `operator|` [lib.op|.bt.bt]

```
bits<N> operator|(const bits<N>& lhs, const bits<N>& rhs);
```

1 Returns `bits<N>(`*lhs*`) |=` *pos*.

23.1.1.25 `operator^` [lib.opˆ.bt.bt]

```
bits<N> operator^(const bits<N>& lhs, const bits<N>& rhs);
```

1 Returns `bits<N>(`*lhs*`) ^=` *pos*.

23.1.1.26 `operator>>` [lib.ext.bt]

```
istream& operator>>(istream& is, bits<N>& x);
```

1 A formatted input function, extracts up to *N* (single-byte) characters from *is*. The function stores these characters in a temporary object *str* of type `string`, then evaluates the expression *x* `= bits<N>(`*str*`)`. Characters are extracted and stored until any of the following occurs:

— *N* characters have been extracted and stored;

— end-of-file occurs on the input sequence;

— the next input character is neither `0` or `1` (in which case the input character is not extracted).

2 If no characters are stored in *str*, the function calls *is*`.setstate(ios::failbit)`.

3 Returns *is*.

23.1.1.27 `operator<<` [lib.ins.bt]

```
    ostream& operator<<(ostream& os, const bits<N>& x);
```

1 Returns *os* `<<` *x*`.to_string()`.

23.1.2 Class `bit_string` [lib.bit.string]

1 The header `<bitstring>` defines a class and several function signatures for representing and manipulating varying-length sequences of bits.

```
class bit_string {
public:
    bit_string();
    bit_string(unsigned long val, size_t n);
    bit_string(const bit_string& str, size_t pos = 0, size_t n = NPOS);
    bit_string(const string& str, size_t pos = 0, size_t n = NPOS);

    bit_string& operator+=(const bit_string& rhs);
    bit_string& operator&=(const bit_string& rhs);
    bit_string& operator|=(const bit_string& rhs);
    bit_string& operator^=(const bit_string& rhs);
    bit_string& operator<<=(size_t pos);
    bit_string& operator>>=(size_t pos);

    bit_string& append(const bit_string& str, pos = 0, n = NPOS);
    bit_string& assign(const bit_string& str, pos = 0, n = NPOS);
    bit_string& insert(size_t pos1, const bit_string& str,
                       size_t pos2 = 0, size_t n = NPOS);
    bit_string& remove(size_t pos = 0, size_t n = NPOS);
    bit_string& replace(size_t pos1, size_t n1, const bit_string& str,
                        size_t pos2 = 0, size_t n2 = NPOS);

    bit_string& set();
    bit_string& set(size_t pos, bool val = 1);
    bit_string& reset();
    bit_string& reset(size_t pos);
    bit_string& toggle();
    bit_string& toggle(size_t pos);

    string to_string() const;
    size_t count() const;
    size_t length() const;
    size_t resize(size_t n, bool val = 0);
    size_t trim();

    size_t find(bool val, size_t pos = 0, size_t n = NPOS) const;
    size_t rfind(bool val, size_t pos = 0, size_t n = NPOS) const;
    bit_string substr(size_t pos, size_t n = NPOS) const;
    bool operator==(const bit_string& rhs) const;
    bool operator!=(const bit_string& rhs) const;
    bool test(size_t pos) const;
    bool any() const;
    bool none() const;
    bit_string operator<<(size_t pos) const;
    bit_string operator>>(size_t pos) const;
    bit_string operator~() const;
private:
//  char* ptr;   exposition only
//  size_t len;  exposition only
};
```

2 The class `bit_string` describes an object that can store a sequence consisting of a varying number of bits. Such a sequence is also called a *bit string* (or simply a *string* if the type of the elements is clear from context). Storage for the string is allocated and freed as necessary by the member functions of class `bit_string`.

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 `bit_string` of length *len* and a value of some integral type, bit position *pos* corresponds to the *bit value* `1 << (len - pos - 1)`.[108] The integral value corresponding to two or more bits is the sum of their bit values.

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

— `char* ptr`, points to the sequence of bits, stored one bit per element;[109]

— `size_t len`, the length of the bit sequence.

5 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`;

— a *length* error is associated with exceptions of type `length_error`;

— an *out-of-range* error is associated with exceptions of type `out_of_range`.

23.1.2.1 `bit_string` constructors [lib.cons.bit.string]

```
bit_string();
```

1 Constructs an object of class `bit_string`, initializing:

— *ptr* to an unspecified value;

— *len* to zero.

```
bit_string(unsigned long val, size_t n);
```

2 Throws `length_error` if *n* equals `NPOS`. Otherwise, the function constructs an object of class `bit_string` and determines its initial string value from *val*. If *val* is zero, the corresponding string is the empty string. Otherwise, the corresponding string is the shortest sequence of bits with the same bit value as `val`. If the corresponding string is shorter than *n*, the string is extended with elements whose values are all zero. Thus, the function initializes:

— *ptr* to point at the first element of the string;

— *len* to the length of the string.

[108] Note that bit position zero is the *most-significant* bit for an object of class `bit_string`, while it is the *least-significant* bit for an object of class `bits<N>`.
[109] An implementation is, of course, free to store the bit sequence more efficiently.

```
bit_string(const bit_string& str, size_t pos = 0, size_t n = NPOS);
```

3 Throws `out_of_range` if *`pos > str.len`*. Otherwise, the function constructs an object of class `bit_string` and determines the effective length *`rlen`* of the initial string value as the smaller of *`n`* and *`str.len - pos`*. Thus, the uunction initializes:

— *`ptr`* to point at the first element of an allocated copy of *`rlen`* elements of the string controlled by *`str`* beginning at position *`pos`*;

— *`len`* to *`rlen`*.

```
bit_string(const string& str, size_t pos = 0, size_t n = NPOS);
```

4 Throws `out_of_range` if *`pos > str.len`*. Otherwise, the function determines the effective length *`rlen`* of the initializing string as the smaller of *`n`* and *`str.len - pos`*.

5 The function then throws `invalid_argument` if any of the *`rlen`* characters in *`str`* beginning at position *`pos`* is other than `0` or `1`.

6 Otherwise, the function constructs an object of class `bit_string` and determines its initial string value from *`str`*. The length of the constructed string is *`rlen`*. 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.

7 Thus, the function initializes:

— *`ptr`* to point at the first element of the string;

— *`len`* to *`rlen`*.

23.1.2.2 `bit_string::operator+=` [lib.bit.string::op+=.bs]

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

1 Throws `length_error` if *`len`* `>= NPOS -` *`rhs.len`*.

2 Otherwise, the function replaces the string controlled by `*this` with a string of length *`len + rhs.len`* whose first *`len`* elements are a copy of the original string controlled by `*this` and whose remaining elements are a copy of the elements of the string controlled by *`rhs`*.

3 Returns `*this`.

23.1.2.3 `bit_string::operator&=` [lib.bit.string::op&=.bs]

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

1 Determines a length *`rlen`* which is the larger of *`len`* and *`rhs.len`*, then behaves as if the shorter of the two strings controlled by `*this` and *`rhs`* were temporarily extended to length *`rlen`* by adding elements all with value zero. The function then replaces the string controlled by `*this` with a string of length *`rlen`* whose elements have the value one only if both of the corresponding elements of `*this` and *`rhs`* are one.

2 Returns `*this`.

23.1.2.4 `bit_string::operator|=` [lib.bit.string::op|=.bs]

```
bit_string& operator|=(const bit_string& rhs);
```

1 Determines a length *`rlen`* which is the larger of *`len`* and *`rhs.len`*, then behaves as if the shorter of the two strings controlled by `*this` and *`rhs`* were temporarily extended to length *`rlen`* by adding elements all with value zero. The function then replaces the string controlled by `*this` with a string of length *`rlen`* whose elements have the value one only if either of the corresponding elements of `*this` and *`rhs`*

are one.

2 Returns `*this`.

23.1.2.5 `bit_string::operator^=` [lib.bit.string::opˆ=.bs]

```
bit_string& operator^=(const bit_string& rhs);
```

1 Determines a length *`rlen`* which is the larger of *`len`* and *`rhs.len`*, then behaves as if the shorter of the two strings controlled by `*this` and *`rhs`* were temporarily extended to length *`rlen`* by adding elements all with value zero. The function then replaces the string controlled by `*this` with a string of length *`rlen`* whose elements have the value one only if the corresponding elements of `*this` and *`rhs`* have different values.

2 Returns `*this`.

23.1.2.6 `bit_string::operator<<=` [lib.bit.string::op.lsh=]

```
bit_string& operator<<=(size_t pos);
```

1 Replaces each element at position *`I`* in the string controlled by `*this` with a value determined as follows:

— If *`pos >= len - I`*, the new value is zero;

— If *`pos < len - I`*, the new value is the previous value of the element at position *`I + pos`*.

2 Returns `*this`.

23.1.2.7 `bit_string::operator>>=` [lib.bit.string::op.rsh=]

```
bit_string& operator>>=(size_t pos);
```

1 Replaces each element at position *`I`* in the string controlled by `*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 element at position *`I - pos`*.

23.1.2.8 `bit_string::append` [lib.bit.string::append]

```
bit_string& append(const bit_string& str, size_t pos = 0,
                   size_t n = NPOS);
```

1 Throws `out_of_range` if *`pos > str.len`*. Otherwise, the function determines the effective length *`rlen`* of the string to append as the smaller of *`n`* and *`str.len - pos`*.

2 The function then throws `length_error` if *`len`* `>= NPOS -` *`rlen`*.

3 Otherwise, the function replaces the string controlled by `*this` with a string of length *`len + rlen`* whose first *`len`* elements are a copy of the original string controlled by `*this` and whose remaining elements are a copy of the initial elements of the string controlled by *`str`* beginning at position *`pos`*.

4 Returns `*this`.

23.1.2.9 `bit_string::assign` [lib.bit.string::assign]

```
bit_string& assign(const bit_string& str, size_t pos = 0,
                   size_t n = NPOS);
```

1 Throws `out_of_range` if *`pos > str.len`*. Otherwise, the function determines the effective length *`rlen`* of the string to assign as the smaller of *`n`* and *`str.len - pos`*.

2 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*.

3 Returns `*this`.

23.1.2.10 `bit_string::insert` [lib.bit.string::insert]

```
bit_string& insert(size_t pos1, const bit_string& str, size_t pos2 = 0,
                   size_t n = NPOS);
```

1 Throws `out_of_range` if *pos1* `>` *len* or *pos2* `>` *str*`.`*len*. Otherwise, the function determines the effective length *rlen* of the string to insert as the smaller of *n* and *str*`.`*len* `-` *pos2*.

2 The function then throws `length_error` if *len* `>= NPOS -` *rlen*.

3 Otherwise, the function replaces the string controlled by `*this` with a string of length *len* `+` *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`.

4 Returns `*this`.

23.1.2.11 `bit_string::remove` [lib.bit.string::remove]

```
bit_string& remove(size_t pos = 0, size_t n = NPOS);
```

1 Throws `out_of_range` if *pos* `>` *len*. Otherwise, the function determines the effective length *xlen* of the string to be removed as the smaller of *n* and *len* `-` *pos*.

2 The function then replaces the string controlled by `*this` with a string of length *len* `-` *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*.

3 Returns `*this`.

23.1.2.12 `bit_string::replace` [lib.bit.string::replace]

```
bit_string& replace(size_t pos1, size_t n1, const bit_string& str,
                    size_t pos2 = 0, size_t n2 = NPOS);
```

1 Throws `out_of_range` if *pos1* `>` *len* or *pos2* `>` *str*`.`*len*. Otherwise, the function determines the effective length *xlen* of the string to be removed as the smaller of *n1* and *len* `-` *pos1*. It also determines the effective length *rlen* of the string to be inserted as the smaller of *n2* and *str*`.`*len* `-` *pos2*.

2 The function then throws `length_error` if *len* `-` *xlen* `>= NPOS -` *rlen*.

3 Otherwise, the function replaces the string controlled by `*this` with a string of length *len* `-` *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*.

4 Returns `*this`.

23.1.2.13 `bit_string::set` [lib.bit.string::set]

```
bit_string& set();
```

1 Sets all elements of the string controlled by `*this`.

2 Returns `*this`.

```
bit_string& set(size_t pos, bool val = 1);
```

3 Throws `out_of_range` if *`pos > len`*. Otherwise, if *`pos == len`*, the function replaces the string controlled by `*this` with a string of length *`len`* `+ 1` whose first *`len`* elements are a copy of the original string and whose remaining element is set according to *`val`*.

4 Otherwise, the function sets the element at position *`pos`* in the string controlled by `*this`. If *`val`* is nonzero, the stored value is one, otherwise it is zero.

5 Returns `*this`.

23.1.2.14 `bit_string::reset` [lib.bit.string::reset]

```
bit_string& reset();
```

1 Resets all elements of the string controlled by `*this`.

2 Returns `*this`.

```
bit_string& reset(size_t pos);
```

3 Throws `out_of_range` if *`pos > len`*. Otherwise, if *`pos == len`*, the function replaces the string controlled by `*this` with a string of length *`len`* `+ 1` whose first *`len`* elements are a copy of the original string and whose remaining element is zero. Otherwise, the function resets the element at position *`pos`* in the string controlled by `*this`.

23.1.2.15 `bit_string::toggle` [lib.bit.string::toggle]

```
bit_string& toggle();
```

1 Toggles all elements of the string controlled by `*this`.

2 Returns `*this`.

```
bit_string& toggle(size_t pos);
```

3 Throws `out_of_range` if *`pos >= len`*. Otherwise, the function toggles the element at position *`pos`* in `*this`.

23.1.2.16 `bit_string::to_string` [lib.bit.string::to.string]

```
string to_string() const;
```

1 Creates an object of type `string` and initializes it to a string of length *`len`* characters. Each character is determined by the value of its corresponding element in the string controlled by `*this`. Bit value zero becomes the character `0`, bit value one becomes the character `1`.

2 Returns the created object.

23.1.2.17 `bit_string::count` [lib.bit.string::count]

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

1 Returns a count of the number of elements set in the string controlled by `*this`.

23.1.2.18 `bit_string::length` [lib.bit.string::length]

```
size_t length() const;
```

1 Returns *len*.

23.1.2.19 `bit_string::resize` [lib.bit.string::resize]

```
size_t resize(size_t n, bool val = 0);
```

1 Throws `length_error` if *n* equals `NPOS`. Otherwise, the function alters the length of the string controlled by `*this` as follows:

— If `n <= len`, the function replaces the string controlled by `*this` with a string of length *n* whose elements are a copy of the initial elements of the original string controlled by `*this`.

— If `n > len`, the function replaces the string controlled by `*this` with a string of length *n* whose first *len* elements are a copy of the original string controlled by `*this`, and whose remaining elements all have the value one if *val* is nonzero, or zero otherwise.

2 Returns the previous value of *len*.

23.1.2.20 `bit_string::trim` [lib.bit.string::trim]

```
size_t trim();
```

1 Determines the highest position *pos* of an element with value one in the string controlled by *`*this`*, if possible. If no such position exists, the function replaces the string with an empty string (*len* is zero). Otherwise, the function replaces the string with a string of length `pos + 1` whose elements are a copy of the initial elements of the original string controlled by `*this`.

2 Returns the new value of *len*.

23.1.2.21 `bit_string::find` [lib.bit.string::find]

```
size_t find(bool val, size_t pos = 0, size_t n = NPOS) const;
```

1 Returns `NPOS` if `pos >= len`. Otherwise, the function determines the effective length *rlen* of the string to be scanned as the smaller of *n* and `len - pos`. The function then determines the lowest position *xpos*, if possible, such that both of the following conditions obtain:

— `pos <= xpos`;

— The element at position *xpos* in the string controlled by *`*this`* is one if *val* is nonzero, or zero otherwise.

2 Returns *xpos* if the function can determine such a value for *xpos*. Otherwise, returns `NPOS`.

23.1.2.22 `bit_string::rfind` [lib.bit.string::rfind]

```
size_t rfind(bool val, size_t pos = 0, size_t n = NPOS) const;
```

1 Returns `NPOS` if `pos >= len`. Otherwise, the function determines the effective length *rlen* of the string to be scanned as the smaller of *n* and `len - pos`. The function then determines the highest position *xpos*, if possible, such that both of the following conditions obtain:

— *pos* <= *xpos*;

— The element at position *xpos* in the string controlled by **this* is one if *val* is nonzero, or zero otherwise.

2 Returns *xpos* if the function can determine such a value for *xpos*. Otherwise, returns `NPOS`.

23.1.2.23 `bit_string::substr` [lib.bit.string::substr]

```
bit_string substr(size_t pos, size_t n = NPOS) const;
```

1 Returns `bit_string(*this, pos, n)`.

23.1.2.24 `bit_string::operator==` [lib.bit.string::op==.bs]

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

1 Returns zero if *len* != *rhs.len* or if the value of any element of the string controlled by **this* differs from the value of the corresponding element of the string controlled by *rhs*.

23.1.2.25 `bit_string::operator!=` [lib.bit.string::op!=.bs]

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

1 Returns a nonzero value if `!(*this == rhs)`.

23.1.2.26 `bit_string::test` [lib.bit.string::test]

```
bool test(size_t pos) const;
```

1 Throws `out_of_range` if *pos* >= *len*. Otherwise, the function returns a nonzero value if the element at position *pos* in the string controlled by `*this` is one.

23.1.2.27 `bit_string::any` [lib.bit.string::any]

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

1 Returns a nonzero value if any bit is set in the string controlled by `*this`.

23.1.2.28 `bit_string::none` [lib.bit.string::none]

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

1 Returns a nonzero value if no bit is set in the string controlled by `*this`.

23.1.2.29 `bit_string::operator<<` [lib.bit.string::op.lsh]

```
bit_string operator<<(size_t pos) const;
```

1 Constructs an object *x* of class `bit_string` and initializes it with `*this`.

2 Returns *x* <<= *pos*.

23.1.2.30 `bit_string::operator>>` [lib.bit.string::op.rsh]

```
bit_string operator>>(size_t pos) const;
```

1 Constructs an object *x* of class `bit_string` and initializes it with `*this`.

2 Returns *x* >>= *pos*.

23.1.2.31 `bit_string::operator~` [lib.bit.string::op~]

```
bit_string operator~() const;
```

1 Constructs an object *x* of class `bit_string` and initializes it with `*this`.

2 Returns `x.toggle()`.

23.1.2.32 `operator+` [lib.op+.bs.bs]

```
bit_string operator+(const bit_string& lhs, const bit_string& rhs);
```

1 Constructs an object *x* of class `bit_string` and initializes it with *lhs*.

2 Returns `x += rhs`.

23.1.2.33 `operator&` [lib.op&.bs.bs]

```
bit_string operator&(const bit_string& lhs, const bit_string& rhs);
```

1 Constructs an object *x* of class `bit_string` and initializes it with *lhs*.

2 Returns `x &= rhs`.

23.1.2.34 `operator|` [lib.op|.bs.bs]

```
bit_string operator|(const bit_string& lhs, const bit_string& rhs);
```

1 Constructs an object *x* of class `bit_string` and initializes it with *lhs*.

2 Returns `x |= rhs`.

23.1.2.35 `operator^` [lib.op^.bs.bs]

```
bit_string operator^(const bit_string& lhs, const bit_string& rhs);
```

1 Constructs an object *x* of class `bit_string` and initializes it with *lhs*.

2 Returns `x ^= rhs`.

23.1.2.36 `operator>>` [lib.ext.bs]

```
istream& operator>>(istream& is, bit_string& x);
```

1 A formatted input function, extracts up to `NPOS - 1` (single-byte) characters from *is*. The function behaves as if it stores these characters in a temporary object *str* of type `string`, then evaluates the expression `x = bit_string(str)`. Characters are extracted and stored until any of the following occurs:

— `NPOS - 1` characters have been extracted and stored;

— end-of-file occurs on the input sequence;

— the next character to read is neither `0` or `1` (in which case the input character is not extracted).

2 If no characters are stored in *str*, the function calls `is.setstate(ios::failbit)`.

3 Returns *is*.

23.1.2.37 operator<< [lib.ins.bs]

```
    ostream& operator<<(ostream& os, const bit_string& x);
```

1 Returns `os << x.to_string()`.

23.1.3 Template class dyn_array [lib.template.dyn.array]

1 The header `<dynarray>` defines a template class and several related functions for representing and manipulating varying-size sequences of some object type *T*.

```
template<class T> class dyn_array {
public:
    dyn_array();
    dyn_array(size_t size, capacity cap);
    dyn_array(const dyn_array<T>& arr);
    dyn_array(const T& obj, size_t rep = 1);
    dyn_array(const T* parr, size_t n);
    dyn_array<T>& operator+=(const dyn_array<T>& rhs);
    dyn_array<T>& operator+=(const T& obj);

    dyn_array<T>& append(const T& obj, size_t rep = 1);
    dyn_array<T>& append(const T* parr, size_t n = 1);
    dyn_array<T>& assign(const T& obj, size_t rep = 1);
    dyn_array<T>& assign(const T* parr, size_t n = 1);
    dyn_array<T>& insert(size_t pos, const dyn_array<T>& arr);
    dyn_array<T>& insert(size_t pos, const T& obj, size_t rep = 1);
    dyn_array<T>& insert(size_t pos, const T* parr, size_t n = 1);
    dyn_array<T>& remove(size_t pos = 0, size_t n = NPOS);

    dyn_array<T>& sub_array(dyn_array<T>& arr, size_t pos,
                            size_t n = NPOS);
    void swap(dyn_array<T>& arr);
    const T& get_at(size_t pos) const;
    void     put_at(size_t pos, const T& obj);
    T&       operator[](size_t pos);
    const T& operator[](size_t pos) const;
    T*       data();
    const T* data() const;

    size_t length() const;
    void resize(size_t n);
    void resize(size_t n, const T& obj);
    size_t reserve() const;
    void reserve(size_t res_arg);
private:
//  T* ptr;                 exposition only
//  size_t len, res;        exposition only
};
```

2 The template class `dyn_array<T>` describes an object that can store a sequence consisting of a varying number of objects of type *T*. The first element of the sequence is at position zero. Such a sequence is also called a *dynamic array*. An object of type *T* shall have:

— a default constructor *T*`()`;

— a copy constructor *T*`(const T&)`;

— an assignment operator *T*`& operator=(const T&)`;

— a destructor `~T()`.

3 For the function signatures described in this subclause:

— it is unspecified whether an operation described in this subclause as initializing an object of type *T* with a copy calls its copy constructor, calls its default constructor followed by its assignment operator, or does nothing to an object that is already properly initialized;

— it is unspecified how many times objects of type *T* are copied, or constructed and destroyed.[110)]

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

— `T *ptr`, points to the sequence of objects;

— `size_t len`, counts the number of objects currently in the sequence;

— `size_t res`, for an unallocated sequence, holds the recommended allocation size of the sequence, while for an allocated sequence, becomes the currently allocated size.

5 In all cases, `len <= res`.

6 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`;

— a *length* error is associated with exceptions of type `length_error`.

— an *out-of-range* error is associated with exceptions of type `out_of_range`;

23.1.3.1 `dyn_array` constructors [lib.cons.dyn.array]

```
dyn_array();
```

1 Constructs an object of class `dyn_array<T>`, initializing:

— *ptr* to an unspecified value;

— *len* to zero;

— *res* to an unspecified value.

```
dyn_array(size_t size, capacity cap);
```

2 Throws `length_error` if *size* equals `NPOS` and *cap* is `default_size`. Otherwise, the function constructs an object of class `dyn_array<T>`. If *cap* is *default_size*, the function initializes:

— *ptr* to point at the first element of an allocated array of *size* elements of type *T*, each initialized with the default constructor for type *T*;

— *len* to *size*;

— *res* to a value at least as large as *len*.

3 Otherwise, *cap* shall be *reserve* and the function initializes:

— *ptr* to an unspecified value;

— *len* to zero;

— *res* to *size*.

```
dyn_array(const dyn_array<T>& arr);
```

110) Objects that cannot tolerate this uncertainty, or that fail to meet the stated requirements, can sometimes be organized into dynamic arrays through the intermediary of an object of class `ptrdyn_array<T>`.

4 Constructs an object of class `dyn_array<T>` and determines its initial dynamic array value by copying the elements from the dynamic array designated by *arr*. Thus, the function initializes:

— *ptr* to point at the first element of an allocated array of *arr.len* elements of type *T*, each initialized with a copy of the corresponding element from the dynamic array designated by *arr*;

— *len* to *arr.len*;

— *res* to a value at least as large as *len*.

```
dyn_array(const T& obj, size_t rep = 1);
```

5 Throws `length_error` if *rep* equals `NPOS`. Otherwise, the function constructs an object of class `dyn_array<T>` and determines its initial dynamic array value by copying *obj* into all *rep* values. Thus, the function initializes:

— *ptr* to point at the first element of an allocated array of *rep* elements of type *T*, each initialized by copying *obj*;

— *len* to *rep*;

— *res* to a value at least as large as *len*.

```
dyn_array(const T* parr, size_t n);
```

6 Throws `length_error` if *n* equals `NPOS`. Otherwise, throws `invalid_argument` if *parr* is a null pointer. Otherwise, *parr* shall designate the first element of an array of at least *n* elements of type *T*.

7 The function then constructs an object of class `dyn_array<T>` and determines its initial dynamic array value by copying the elements from the array designated by *parr*. Thus, the function initializes:

— *ptr* to point at the first element of an allocated array of *n* elements of type *T*, each initialized with a copy of the corresponding element from the array designated by *parr*;

— *len* to *n*;

— *res* to a value at least as large as *len*.

23.1.3.2 `dyn_array::operator+=` [lib.dyn.array::op+=]

```
dyn_array<T>& operator+=(const dyn_array<T>& rhs);
```

1 Throws `length_error` if `len >= NPOS - rhs.len`. Otherwise, the function replaces the dynamic array designated by `*this` with a dynamic array of length `len + rhs.len` whose first *len* elements are a copy of the original dynamic array designated by `*this` and whose remaining elements are a copy of the elements of the dynamic array designated by *rhs*.

2 Returns `*this`.

```
dyn_array<T>& operator+=(const T& obj);
```

3 Returns `append(obj)`.

23.1.3.3 `dyn_array::append` [lib.dyn.array::append]

```
dyn_array<T>& append(const T& obj, size_t rep = 1);
```

1 Throws `length_error` if `len >= NPOS - rep`. Otherwise, the function replaces the dynamic array designated by `*this` with a dynamic array of length `len + rep` whose first *len* elements are a copy of the original dynamic array designated by `*this` and whose remaining elements are each a copy of *obj*.

2 Returns `*this`.

```
dyn_array<T>& append(const T* parr, size_t n = 1);
```

3 Throws `length_error` if *`len`* `>= NPOS -` *`n`*. Otherwise, the function throws `invalid_argument` if *`n`* `> 0` and *`parr`* is a null pointer. Otherwise, *`parr`* shall designate the first element of an array of at least *`n`* elements of type *`T`*.

4 The function then replaces the dynamic array designated by `*this` with a dynamic array of length *`len`* `+` *`n`* whose first *`len`* elements are a copy of the original dynamic array designated by `*this` and whose remaining elements are a copy of the initial elements of the array designated by *`parr`*.

5 Returns `*this`.

23.1.3.4 `dyn_array::assign` [lib.dyn.array::assign]

```
dyn_array<T>& assign(const T& obj, size_t rep = 1);
```

1 Throws `length_error` if *`rep`* `== NPOS`. Otherwise, the function replaces the dynamic array designated by `*this` with a dynamic array of length *`rep`* each of whose elements is a copy of *`obj`*.

2 Returns `*this`.

```
dyn_array<T>& assign(const T* parr, size_t n = 1);
```

3 Throws `length_error` if *`n`* `== NPOS`. Otherwise, the function throws `invalid_argument` if *`n`* `> 0` and *`parr`* is a null pointer.

4 Otherwise, *`parr`* shall designate the first element of an array of at least *`n`* elements of type *`T`*.

5 The function then replaces the dynamic array designated by `*this` with a dynamic array of length *`n`* whose elements are a copy of the initial elements of the array designated by *`parr`*.

6 Returns `*this`.

23.1.3.5 `dyn_array::insert` [lib.dyn.array::insert]

```
dyn_array<T>& insert(size_t pos, const dyn_array<T>& arr);
```

1 Throws `out_of_range` if *`pos`* `>` *`len`*. Otherwise, the function throws `length_error` if *`len`* `>= NPOS -` *`arr.len`*.

2 Otherwise, the function replaces the dynamic array designated by `*this` with a dynamic array of length *`len`* `+` *`arr.len`* whose first *`pos`* elements are a copy of the initial elements of the original dynamic array designated by `*this`, whose next *`arr.len`* elements are a copy of the initial elements of the dynamic array designated by *`arr`*, and whose remaining elements are a copy of the remaining elements of the original dynamic array designated by `*this`.

3 Returns `*this`.

```
dyn_array<T>& insert(size_t pos, const T& obj, size_t rep = 1);
```

4 Throws `out_of_range` if *`pos`* `>` *`len`*. Otherwise, the function throws `length_error` if *`len`* `>= NPOS -` *`rep`*.

5 Otherwise, the function replaces the dynamic array designated by `*this` with a dynamic array of length *`len`* `+` *`rep`* whose first *`pos`* elements are a copy of the initial elements of the original dynamic array designated by `*this`, whose next *`rep`* elements are each a copy of *`obj`*, and whose remaining elements are a copy of the remaining elements of the original dynamic array designated by `*this`.

6 Returns `*this`.

```
dyn_array<T>& insert(size_t pos, const T* parr, size_t n = 1);
```

7 Throws `out_of_range` if *pos* `>` *len*. Otherwise, the function throws `length_error` if *len* `>= NPOS -` *n*. Otherwise, the function throws `invalid_argument` if *n* `> 0` and *parr* is a null pointer. Otherwise, *parr* shall designate the first element of an array of at least *n* elements of type *T*.

8 The function then replaces the dynamic array designated by `*this` with a dynamic array of length *len* `+` *n* whose first *pos* elements are a copy of the initial elements of the original dynamic array designated by `*this`, whose next *n* elements are a copy of the initial elements of the array designated by *parr*, and whose remaining elements are a copy of the remaining elements of the original dynamic array designated by `*this`.

9 Returns `*this`.

23.1.3.6 `dyn_array::remove` [lib.dyn.array::remove]

```
dyn_array<T>& remove(size_t pos = 0, size_t n = NPOS);
```

1 Throws `out_of_range` if *pos* `>` *len*. Otherwise, the function determines the effective length *xlen* of the sequence to be removed as the smaller of *n* and *len* `-` *pos*.

2 The function then replaces the dynamic array designated by `*this` with a dynamic array of length *len* `-` *xlen* whose first *pos* elements are a copy of the initial elements of the original dynamic array designated by `*this`, and whose remaining elements are a copy of the elements of the original dynamic array designated by `*this` beginning at position *pos* `+` *xlen*. The original *xlen* elements beginning at position *pos* are destroyed.

3 Returns `*this`.

23.1.3.7 `dyn_array::sub_array` [lib.dyn.array::sub.array]

```
dyn_array<T>& sub_array(dyn_array<T>& arr, size_t pos, size_t n = NPOS);
```

1 Throws `out_of_range` if *pos* `>` *len*. Otherwise, the function determines the effective length *rlen* of the dynamic array designated by `*this` as the smaller of *n* and *arr*`.`*len* `-` *pos*.

2 The function then replaces the dynamic array designated by *arr* with a dynamic array of length *rlen* whose elements are a copy of the elements of the dynamic array designated by `*this` beginning at position *pos*.

3 Returns *arr*.

23.1.3.8 `dyn_array::swap` [lib.dyn.array::swap]

```
void swap(dyn_array<T>& arr);
```

1 Replaces the dynamic array designated by `*this` with the dynamic array designated by *arr*, and replaces the dynamic array designated by *arr* with the dynamic array originally designated by `*this`.[111)]

23.1.3.9 `dyn_array::get_at` [lib.dyn.array::get.at]

```
const T& get_at(size_t pos) const;
```

1 Throws `out_of_range` if *pos* `>=` *len*. Otherwise, returns *ptr*`[`*pos*`]`.

2 The reference returned is invalid after a subsequent call to any member function for the object.

111) Presumably, this operation occurs with no actual copying of array elements.

23.1.3.10 dyn_array::put_at [lib.dyn.array::put.at]

```
void put_at(size_t pos, const T& obj);
```

1 Throws `out_of_range` if *pos* `>=` *len*. Otherwise, the function assigns *obj* to the element at position *pos* in the dynamic array designated by `*this`.

23.1.3.11 dyn_array::operator[] [lib.dyn.array::op.array]

```
T&       operator[](size_t pos);
const T& operator[](size_t pos) const;
```

1 If *pos* `<` *len*, returns the element at position *pos* in the dynamic array designated by `*this`. Otherwise, the behavior is undefined.

2 The reference returned is invalid after a subsequent call to any member function for the object.

23.1.3.12 dyn_array::data [lib.dyn.array::data]

```
T*       data();
const T* data() const;
```

1 Returns `ptr` if *len* is nonzero, otherwise a null pointer. The program shall not alter any of the values stored in the dynamic 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 `dyn_array<T>` that designates the same object as `this`.

23.1.3.13 dyn_array::length [lib.dyn.array::length]

```
size_t length() const;
```

1 Returns *len*.

23.1.3.14 dyn_array::resize [lib.dyn.array::resize]

```
void resize(size_t n);
```

1 Throws `length_error` if *n* equals `NPOS`. Otherwise, if *n* `!=` *len* the function alters the length of the dynamic array designated by `*this` as follows:

— If *n* `<` *len*, the function replaces the dynamic array designated by `*this` with a dynamic array of length *n* whose elements are a copy of the initial elements of the original dynamic array designated by `*this`. Any remaining elements are destroyed.

— If *n* `>` *len*, the function replaces the dynamic array designated by `*this` with a dynamic array of length *n* whose first *len* elements are a copy of the original dynamic array designated by `*this`, and whose remaining elements are all initialized with the default constructor for class *T*.

```
void resize(size_t n, const T& obj);
```

2 Throws `length_error` if *n* equals `NPOS`. Otherwise, if *n* `!=` *len* the function alters the length of the dynamic array designated by `*this` as follows:

— If *n* `<` *len*, the function replaces the dynamic array designated by `*this` with a dynamic array of length *n* whose elements are a copy of the initial elements of the original dynamic array designated by `*this`. Any remaining elements are destroyed.

— If *n* `>` *len*, the function replaces the dynamic array designated by `*this` with a dynamic array of length *n* whose first *len* elements are a copy of the original dynamic array designated by `*this`, and whose remaining elements are all initialized by copying *obj*.

23.1.3.15 `dyn_array::reserve` [lib.dyn.array::reserve]

```
    size_t reserve() const;
```

1 Returns *res*.

```
    void reserve(size_t res_arg);
```

2 If no dynamic array is allocated, assigns *res_arg* to *res*. Otherwise, whether or how the function alters *res* is unspecified.

23.1.3.16 `operator+` [lib.op+.da.da]

```
    dyn_array<T> operator+(const dyn_array<T>& lhs,
                                               const dyn_array<T>& rhs);
```

1 Returns `dyn_array<T>(lhs) += rhs`.

```
    dyn_array<T> operator+(const dyn_array<T>& lhs, const T& obj);
```

2 Returns `dyn_array<T>(lhs) += rhs`.

```
    dyn_array<T> operator+(const T& obj, const dyn_array<T>& rhs);
```

3 Returns `dyn_array<T>(lhs) += rhs`.

23.1.4 Template class `ptr_dyn_array` [lib.template.ptr.dyn.array]

1 The header `<ptrdynarray>` defines a template and several related functions for representing and manipulating varying-size sequences of pointers to some object type *T*.

```
template<class T> class ptr_dyn_array : public dyn_array<void*> {
public:
    ptr_dyn_array();
    ptr_dyn_array(size_t size, capacity cap);
    ptr_dyn_array(const ptr_dyn_array<T>& arr);
    ptr_dyn_array(T* obj, size_t rep = 1);
    ptr_dyn_array(T** parr, size_t n = 1);

    ptr_dyn_array<T>& operator+=(const ptr_dyn_array<T>& rhs);
    ptr_dyn_array<T>& operator+=(T* obj);
    ptr_dyn_array<T>& append(T* obj, size_t rep = 1);
    ptr_dyn_array<T>& append(T** parr, size_t n = 1);
    ptr_dyn_array<T>& assign(T* obj, size_t rep = 1);
    ptr_dyn_array<T>& assign(T** parr, size_t n = 1);
    ptr_dyn_array<T>& insert(size_t pos, const ptr_dyn_array<T>& arr);
    ptr_dyn_array<T>& insert(size_t pos, T* obj, size_t rep = 1);
    ptr_dyn_array<T>& insert(size_t pos, T** parr, size_t n = 1);
    ptr_dyn_array<T>& remove(size_t pos = 0, size_t n = NPOS);

    ptr_dyn_array<T>& sub_array(ptr_dyn_array<T>& arr, size_t pos,
                                                  size_t n = NPOS);
    void swap(ptr_dyn_array<T>& arr);
    T*   get_at(size_t pos) const;
    void put_at(size_t pos, T* obj);
    T*&       operator[](size_t pos);
    T* const& operator[](size_t pos) const;
    T**       data();
    const T** data() const;
    size_t length() const;
    void resize(size_t n);
    void resize(size_t n, T* obj);
    size_t reserve() const;
    void   reserve(size_t res_arg);
};
```

2 The template class `ptr_dyn_array<T>` describes an object that can store a sequence consisting of a varying number of objects of type pointer to *T*. Such a sequence is also called a *dynamic pointer array*. Objects of type *T* are never created, destroyed, copied, assigned, or otherwise accessed by the function signatures described in this subclause.

23.1.4.1 `ptr_dyn_array` constructors [lib.cons.ptr.dyn.array]

```
    ptr_dyn_array();
```

1 Constructs an object of class `ptr_dyn_array<T>`, initializing the base class with `dyn_array<void*>()`.

```
    ptr_dyn_array(size_t size, capacity cap);
```

2 Constructs an object of class `ptr_dyn_array<T>`, initializing the base class with `dyn_array<void*>(`*size*`, `*cap*`)`.

```
    ptr_dyn_array(const ptr_dyn_array<T>& arr);
```

3 Constructs an object of class `ptr_dyn_array<T>`, initializing the base class with `dyn_array<void*>(`*arr*`)`.

```
    ptr_dyn_array(T* obj, size_t rep = 1);
```

4 Constructs an object of class `ptr_dyn_array<T>`, initializing the base class with `dyn_array<void*>((void*)`*obj*`, `*rep*`)`.

```
    ptr_dyn_array(const T** parr, size_t n);
```

5 Constructs an object of class `ptr_dyn_array<T>`, initializing the base class with `dyn_array<void*>((void**)`*parr*`, `*n*`)`.

23.1.4.2 `ptr_dyn_array::operator+=` [lib.ptr.dyn.array::op+=]

```
    ptr_dyn_array<T>& operator+=(const ptr_dyn_array<T>& rhs);
```

1 Returns `(ptr_dyn_array<T>&)dyn_array<void*>::operator+=((const dyn_array<void*>&)`*rhs*`)`.

```
    ptr_dyn_array<T>& operator+=(T* obj);
```

2 Returns `(ptr_dyn_array<T>&)dyn_array<void*>:: operator+=((void*)`*obj*`)`.

23.1.4.3 `ptr_dyn_array::append` [lib.ptr.dyn.array::append]

```
    ptr_dyn_array<T>& append(T* obj, size_t rep = 1);
```

1 Returns `(ptr_dyn_array<T>&)dyn_array<void*>::append((void*)`*obj*`, `*rep*`)`.

```
    ptr_dyn_array<T>& append(T** parr, size_t n = 1);
```

2 Returns `(ptr_dyn_array<T>&)dyn_array<void*>::append((void**)`*parr*`, `*n*`)`.

23.1.4.4 `ptr_dyn_array::assign` [lib.ptr.dyn.array::assign]

```
ptr_dyn_array<T>& assign(T* obj, size_t rep = 1);
```

1 Returns `(ptr_dyn_array<T>&)dyn_array<void*>::assign((void*)`*obj*`, `*rep*`)`.

```
ptr_dyn_array<T>& assign(T** parr, size_t n = 1);
```

2 Returns `(ptr_dyn_array<T>&)dyn_array<void*>::assign((void**)`*parr*`, `*n*`)`.

23.1.4.5 `ptr_dyn_array::insert` [lib.ptr.dyn.array::insert]

```
ptr_dyn_array<T>& insert(size_t pos, const ptr_dyn_array<T>& arr);
```

1 Returns `(ptr_dyn_array<T>&)dyn_array<void*>::insert(`*pos*`, (const dyn_array<void*>&)`*arr*`)`.

```
ptr_dyn_array<T>& insert(size_t pos, T*obj, size_t rep = 1);
```

2 Returns `(ptr_dyn_array<T>&)dyn_array<void*>::insert(`*pos*`, (void*)`*obj*`, `*rep*`)`.

```
ptr_dyn_array<T>& insert(size_t pos, T**parr, size_t n = 1);
```

3 Returns `(ptr_dyn_array<T>&)dyn_array<void*>::insert(`*pos*`, (void**)`*parr*`, `*n*`)`.

23.1.4.6 `ptr_dyn_array::remove` [lib.ptr.dyn.array::remove]

```
ptr_dyn_array<T>& remove(size_t pos = 0, size_t n = NPOS);
```

1 Returns `(ptr_dyn_array<T>&)dyn_array<void*>::remove(`*pos*`, `*n*`)`.

23.1.4.7 `ptr_dyn_array::sub_array` [lib.ptr.dyn.array::sub.array]

```
ptr_dyn_array<T>& sub_array(ptr_dyn_array<T>& arr, size_t pos,
                                              size_t n = NPOS);
```

1 Returns `(ptr_dyn_array<T>&)dyn_array<void*>::sub_array(`*arr*`, `*pos*`, `*n*`)`.

23.1.4.8 `ptr_dyn_array::swap` [lib.ptr.dyn.array::swap]

```
void swap(ptr_dyn_array<T>& arr);
```

1 Calls `dyn_array<void*>::swap(`*arr*`)`.

23.1.4.9 `ptr_dyn_array::get_at` [lib.ptr.dyn.array::get.at]

```
T* get_at(size_t pos) const;
```

1 Returns `(T*)dyn_array<void*>::get_at(`*pos*`)`.

23.1.4.10 `ptr_dyn_array::put_at` [lib.ptr.dyn.array::put.at]

```
void put_at(size_t pos, T* obj);
```

1 Calls `dyn_array<void*>::put_at(`*pos*`, (void*)`*obj*`)`.

23.1.4.11 `ptr_dyn_array::operator[]` [lib.ptr.dyn.array::op.array]

```
T*&       operator[](size_t pos);
T* const& operator[](size_t pos) const;
```

1 Returns `(T*&)dyn_array<void*>::operator[](`*pos*`)`.

23.1.4.12 `ptr_dyn_array::data` [lib.ptr.dyn.array::data]

```
T** data();
const T** data() const;
```

1 Returns `(T*)dyn_array<void*>::data()`.

23.1.4.13 `ptr_dyn_array::length` [lib.ptr.dyn.array::length]

```
size_t length() const;
```

1 Returns `dyn_array<void*>::length()`.

23.1.4.14 `ptr_dyn_array::resize` [lib.ptr.dyn.array::resize]

```
void resize(size_t n);
```

1 Calls `dyn_array<void*>::resize(`*n*`)`.

```
void resize(size_t n, T* obj);
```

2 Calls `dyn_array<void*>::resize(`*n*`, (void*)`*obj*`)`.

23.1.4.15 `ptr_dyn_array::reserve` [lib.ptr.dyn.array::reserve]

```
size_t reserve() const;
```

1 Returns `dyn_array<void*>::reserve()`.

```
void reserve(size_t res_arg);
```

2 Returns `dyn_array<void*>::reserve(`*res_arg*`)`.

23.1.4.16 `operator+` [lib.op+.pda.pda]

```
ptr_dyn_array<T> operator+(const ptr_dyn_array<T>& lhs,
const ptr_dyn_array<T>& rhs);
```

1 Returns `ptr_dyn_array<T><T>(`*lhs*`) += `*rhs*`)`.

```
ptr_dyn_array<T> operator+(const ptr_dyn_array<T>& lhs, T* obj);
```

2 Returns `ptr_dyn_array<T><T>(`*lhs*`) += `*rhs*`)`.

```
ptr_dyn_array<T> operator+(T* obj, const ptr_dyn_array<T>& rhs);
```

3 Returns `ptr_dyn_array<T><T>(`*lhs*`) += `*rhs*`)`.

23.1.5 Template class `vector` [lib.vector]

1 `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.

```
template <class T, template <class U> class Allocator = allocator>
class vector {
public:
  // typedefs:
    typedef ? iterator;
    typedef ? const_iterator;
    typedef ? size_type;
    typedef ? difference_type;
    typedef T value_type;

  // allocation/deallocation:
    vector();
    vector(size_type n, const T& value = T());
    vector(const vector<T, Allocator>& x);
    template <class InputIterator>
      vector(InputIterator first, InputIterator last);
   ~vector();
    vector<T, Allocator>& operator=(const vector<T, Allocator>& x);
    void reserve(size_type n);

  // accessors:
    iterator       begin();
    const_iterator begin() const;
    iterator       end();
    const_iterator end() const;
    size_type size() const;
    size_type max_size() const;
    size_type capacity() const;
    bool empty() const;
    T&       operator[](size_type n);
    const T& operator[](size_type n) const;
    T&       front();
    const T& front() const;
    T&       back();
    const T& back() const;

  // insert/erase:
    void push_back(const T& x);
    void pop_back();
    iterator insert(iterator position, const T& x = T());
    void     insert(iterator position, size_type n, const T& x = T());
    template <class InputIterator>
        void insert(iterator position, InputIterator first, InputIterator last);
    void erase(iterator position);
    void erase(iterator first, iterator last);
};

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

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

23.1.5.1 Typedefs [lib.vector.typedefs]

1 `iterator` is a random access iterator referring to `T`. The exact type is implementation dependent and determined by `Allocator`.

2 `const_iterator` is a constant random access iterator referring to `const T`. The exact type is implementation dependent and determined by `Allocator`. It is guaranteed that there is a constructor for

`const_iterator` out of `iterator`.

3 `size_type` is an unsigned integral type. The exact type is implementation dependent and determined by `Allocator`.

4 `difference_type` is a signed integral type. The exact type is implementation dependent and determined by `Allocator`.

23.1.5.2 Constructors [lib.vector.cons]

1 The constructor `template <class InputIterator> vector(InputIterator first, InputIterator last)` makes only `N` calls to the copy constructor of `T` (where `N` is the distance between `first` and `last`) and no reallocations if iterators first and last are of forward, bidirectional, or random access categories. It does at most `2N` calls to the copy constructor of `T` and `logN` reallocations if they are just input iterators, since it is impossible to determine the distance between `first` and `last` and then do copying.

23.1.5.3 Member functions [lib.vector.members]

1 The member function `capacity` returns the size of the allocated storage in the vector. The member function `reserve` is a directive that informs `vector` of a planned change in size, so that it can manage the storage allocation accordingly. It does not change the size of the sequence and takes at most linear time in the size of the sequence. Reallocation happens at this point if and only if the current capacity is less than the argument of `reserve`. After `reserve`, `capacity` is greater or equal to the argument of `reserve` if reallocation happens; and equal to the previous value of `capacity` otherwise. Reallocation invalidates all the references, pointers, and iterators referring to the elements in the sequence. It is guaranteed that no reallocation takes place during the insertions that happen after `reserve` takes place till the time when the size of the vector reaches the size specified by `reserve`.

2 `insert` 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. 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. In other words, it is much faster to insert many elements into the middle of a vector at once than to do the insertion one at a time. The insert template member function preallocates enough storage for the insertion if the iterators `first` and `last` are of forward, bidirectional or random access category. Otherwise, it does insert elements one by one and should not be used for inserting into the middle of vectors.

3 `erase` 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.

23.1.6 Class `vector<bool>` [lib.vector.bool]

1 To optimize space allocation, a specialization for `bool` is provided:

```
class vector<bool, allocator> {
public:
  // bit reference:
    class reference {
    public:
       ~reference();
        operator bool() const;
        reference& operator=(const bool x);
        void flip();      // flips the bit
    };
  // typedefs:
    typedef ? iterator;
    typedef ? const_iterator;
    typedef size_t    size_type;
    typedef ptrdiff_t difference_type ;
    typedef bool      value_type;

  // allocation/deallocation:
    vector();
    vector(size_type n, const bool& value = bool());
    vector(const vector<bool, allocator>& x);
    template <class InputIterator>
      vector(InputIterator first, InputIterator last);
   ~vector();
    vector<bool, allocator>& operator=(const vector<bool, allocator>& x);
    void reserve(size_type n);

  // accessors:
    iterator       begin();
    const_iterator begin() const;
    iterator       end();
    const_iterator end() const;
    size_type size() const;
    size_type max_size() const;
    size_type capacity() const;
    bool empty() const;
    reference       operator[](size_type n);
    const reference operator[](size_type n) const;
    reference       front();
    const reference front() const;
    reference       back();
    const reference back() const;

  // insert/erase:
    void push_back(const bool& x);
    void pop_back();
    iterator insert(iterator position, const bool& x = bool());
    void     insert (iterator position, size_type n, const bool& x = bool());
    template <class InputIterator>
        void insert (iterator position, InputIterator first, InputIterator last);
    void erase(iterator position);
    void erase(iterator first, iterator last);
};

bool operator==(const vector<bool, allocator>& x, const vector<bool, allocator>& y);

bool operator< (const vector<bool, allocator>& x, const vector<bool, allocator>& y);
```

2 `reference` is a class that simulates the behavior of references of a single bit in `vector<bool>`.

3 Every implementation is expected to provide specializations of `vector<bool>` for all supported memory models.

Box 98

At present, it is not possible to templatize a specialization. That is, we cannot write:

```
  template <template <class U> class Allocator = allocator>
  class vector<bool, Allocator> { /* ... */ };
```

Therefore, only `vector<bool, allocator>` is provided.

23.1.7 Template class `list` [lib.list]

1 `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 and deques, fast random access to list elements is not supported, but many algorithms only need sequential access anyway.

```
template <class T, template <class U> class Allocator = allocator>
class list {
public:
  // typedefs:
    typedef ? iterator;
    typedef ? const_iterator;
    typedef ? size_type;
    typedef ? difference_type;
    typedef T value_type;

  // allocation/deallocation:
    list();
    list(size_type n, const T& value = T());
    template <class InputIterator>
      list(InputIterator first, InputIterator last);
    list(const list<T, Allocator>& x);
   ~list();
    list<T, Allocator>& operator=(const list<T, Allocator>& x);

  // accessors:
    iterator       begin();
    const_iterator begin() const;
    iterator       end();
    const_iterator end() const;
    bool empty() const;
    size_type size() const;
    size_type max_size() const;
    T&       front();
    const T& front() const;
    T&       back();
    const T& back() const;

  // insert/erase:
    void push_front(const T& x);
    void pop_front();
    void push_back(const T& x);
    void pop_back();
    iterator insert(iterator position, const T& x = T());
    void     insert(iterator position, size_type n, const T& x = T());
    template <class InputIterator>
        void insert(iterator position, InputIterator first, InputIterator last);
    void erase(iterator position);
    void erase(iterator first, iterator last);
  // special mutative operations on list:
    void splice(iterator position, list<T, Allocator>& x);
    void splice(iterator position, list<T, Allocator>& x, iterator i);
    void splice(iterator position, list<T, Allocator>& x, iterator first,
                iterator last);
    void   remove(const T& value);
    template <class Predicate>
      void remove_if(Predicate pred);
    void   unique();
    template <class BinaryPredicate>
      void unique(BinaryPredicate binary_pred);
    void   merge(list<T, Allocator>& x);
    template <class Compare>
      void merge(list<T, Allocator>& x, Compare comp);
    void   sort();
    template <class Compare>
      void sort(Compare comp);
    void reverse();
};

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

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

23.1.7.1 Typedefs [lib.list.typedefs]

1 `iterator` is a bidirectional iterator referring to `T`. The exact type is implementation dependent and determined by `Allocator`.

2 `const_iterator` is a constant bidirectional iterator referring to `const T`. The exact type is implementation dependent and determined by `Allocator`. It is guaranteed that there is a constructor for `const_iterator` out of `iterator`.

3 `size_type` is an unsigned integral type. The exact type is implementation dependent and determined by `Allocator`.

4 `difference_type` is a signed integral type. The exact type is implementation dependent and determined by `Allocator`.

23.1.7.2 Member functions [lib.list.members]

1 `insert` does not affect the validity of iterators and references. 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.

2 `erase` invalidates only the iterators and references to the erased elements. 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.

3 Since lists allow fast insertion and erasing from the middle of a list, certain operations are provided specifically for them:

4 `list` provides three splice operations that destructively move elements from one list to another:

5 `void splice(iterator position, list<T, Allocator>& x)` inserts the contents of `x` before `position` and `x` becomes empty. It takes constant time.

6 `void splice(iterator position, list<T, Allocator>& x, iterator i)` inserts an element pointed to by `i` from list `x` before position and removes the element from `x`. It takes constant time. `i` is a valid iterator of `x`.

7 `void splice(iterator position, list<T, Allocator>& x, iterator first, iterator last)` inserts elements in the range `[first, last)` before `position` and removes the elements from `x`. It takes linear time. `[first, last)` is a valid range in `x`.

8 `remove` erases all the elements in the list referred by the list iterator `i` for which the following conditions hold: `*i == value, pred(*i) == true`. `remove` is stable, that is, the relative order of the elements that are not removed is the same as their relative order in the original list. Exactly `size()` applications of the corresponding predicate are done.

9 `unique` erases all but the first element from every consecutive group of equal elements in the list. Exactly `size() - 1` applications of the corresponding binary predicate are done.

10 `merge` merges the argument list into the list (both are assumed to be sorted). The merge is stable, that is, 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. At most `size() + x.size() - 1` comparisons are done.

11 `reverse` reverses the order of the elements in the list. It is linear time.

12 `sort` sorts the list according to the `operator<` or a compare function object. It is stable, that is, the relative order of the equal elements is preserved. Approximately `NlogN` comparisons are done where `N` is equal to `size()`.

23.1.8 Template class `deque` [lib.deque]

1 `deque` is a kind of sequence that, like a `vector`, 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. As with vectors, storage management is handled automatically.

```
template <class T, template <class U> class Allocator = allocator>
class deque {
public:
  // typedefs:
    typedef ? iterator;
    typedef ? const_iterator;
    typedef ? size_type;
    typedef ? difference_type;
    typedef T value_type;

  // allocation/deallocation:
    deque();
    deque(size_type n, const T& value = T());
    deque(const deque<T, Allocator>& x);
    template <class InputIterator>
      deque(InputIterator first, InputIterator last);
   ~deque();
    deque<T, Allocator>& operator=(const deque<T, Allocator>& x);
  // accessors:
    iterator       begin();
    const_iterator begin() const;
    iterator       end();
    const_iterator end() const;
    size_type size() const;
    size_type max_size() const;
    bool empty() const;
    T&       operator[](size_type n);
    const T& operator[](size_type n) const;
    T&       front();
    const T& front() const;
    T&       back();
    const T& back() const;

// insert/erase:
  void push_front(const T& x);
  void push_back(const T& x);
  iterator insert(iterator position, const T& x = T());
  void     insert (iterator position, size_type n, const T& x = T());
  template <class InputIterator>
     void insert (iterator position, InputIterator first, InputIterator last);
  void pop_front();
  void pop_back();
  void erase(iterator position);
  void erase(iterator first, iterator last);
};

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

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

23.1.8.1 Typedefs [lib.deque.typedefs]

1 `iterator` is a random access iterator referring to `T`. The exact type is implementation dependent and determined by `Allocator`.

2 `const_iterator` is a constant random access iterator referring to `const T`. The exact type is implementation dependent and determined by `Allocator`. It is guaranteed that there is a constructor for `const_iterator` out of `iterator`.

3 `size_type` is an unsigned integral type. The exact type is implementation dependent and determined by `Allocator`.

4 `difference_type` is a signed integral type. The exact type is implementation dependent and determined by `Allocator`.

23.1.8.2 Member functions [lib.deque.members]

1 `insert` invalidates all the iterators and references to the deque if the insertion pointer is not at either end. Insertion at either end does not affect iterators and references. 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`. That is, a deque is especially optimized for pushing and popping elements at the beginning and end.

2 `erase` invalidates all the iterators and references to the deque if the erasing point is not at either end. Erasing at either end does not affect iterators and references. The number of calls to the destructor is the same as the number of elements erased, but the number of the calls to the assignment operator is equal to the minimum of the number of elements before the erased elements and the number of element after the erased elements.

23.1.9 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`, `list` and `deque` can be used.

```
template <class Container>
class stack {
friend bool operator==(const stack<Container>& x, const stack<Container>& y);
public:
  typedef Container::value_type value_type;
  typedef Container::size_type  size_type;
protected:
  Container c;

public:
  bool empty() const                    { return c.empty(); }
  size_type size() const                { return c.size(); }
  value_type&       top()               { return c.back(); }
  const value_type& top() const         { return c.back(); }
  void push(const value_type& x)        { c.push_back(x); }
  void pop()                            { c.pop_back(); }
};

template <class Container>
bool operator==(const stack<Container>& x, const stack<Container>& y) {
  return x.c == y.c;
}
```

2 For example, `stack<vector<int> >` is an integer stack made out of `vector`, and `stack<deque<char> >` is a character stack made out of `deque`.

23.1.10 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` and `deque` can be used.

```
template <class Container>
class queue {
friend bool operator==(const queue<Container>& x, const queue<Container>& y);
public:
  typedef Container::value_type value_type;
  typedef Container::size_type  size_type;
protected:
  Container c;
public:
  bool empty() const                    { return c.empty(); }
  size_type size() const                { return c.size(); }
  value_type&       front()             { return c.front(); }
  const value_type& front() const       { return c.front(); }
  value_type&       back()              { return c.back(); }
  const value_type& back() const        { return c.back(); }
  void push(const value_type& x)        { c.push_back(x); }
  void pop()                            { c.pop_front(); }
};

template <class Container>
bool operator==(const queue<Container>& x, const queue<Container>& y) {
  return x.c == y.c;
}
```

23.1.11 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` and `deque` can be used.

```
template <class Container, class Compare = less<Container::value_type> >
class priority_queue {
public:
  typedef Container::value_type value_type;
  typedef Container::size_type  size_type;
protected:
  Container c;
  Compare comp;

public:
  priority_queue(const Compare& x = Compare()) : c(), comp(x) {}
  template <class InputIterator>
    priority_queue(InputIterator first, InputIterator last,
      const Compare& x = Compare())
    : c(first, last), comp(x) {
      make_heap(c.begin(), c.end(), comp);
  }

  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) {
    c.push_back(x);
    push_heap(c.begin(), c.end(), comp);
  }

  void pop() {
    pop_heap(c.begin(), c.end(), comp);
    c.pop_back();
  }
};

// no equality is provided
```

23.2 Associative containers [lib.associative]

23.2.1 Template class `set` [lib.set]

1 `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.

```
template <class Key, class Compare = less<Key>,
  template <class U> class Allocator = allocator>
class set {
public:
// typedefs:
  typedef Key     key_type;
  typedef Key     value_type;
  typedef Compare key_compare;
  typedef Compare value_compare;
  typedef ? iterator;
  typedef iterator const_iterator;
  typedef ? size_type;
  typedef ? difference_type;

// allocation/deallocation:
  set(const Compare& comp = Compare());
  template <class InputIterator>
    set(InputIterator first, InputIterator last,
        const Compare& comp = Compare());
  set(const set<Key, Compare, Allocator>& x);
 ~set();
  set<Key, Compare, Allocator>& operator=(const set<Key, Compare, Allocator>& x);

// accessors:
  key_compare   key_comp() const;
  value_compare value_comp() const;
  iterator      begin() const;
  iterator      end() const;
  bool          empty() const;
  size_type     size() const;
  size_type     max_size() const;

// insert/erase:
  pair<iterator, bool> insert(const value_type& x);
  iterator             insert(iterator position, const value_type& x);
  template <class InputIterator>
     void insert(InputIterator first, InputIterator last);
  void      erase(iterator position);
  size_type erase(const key_type& x);
  void      erase(iterator first, iterator last);

// set operations:
  iterator  find(const key_type& x) const;
  size_type count(const key_type& x) const;
  iterator  lower_bound(const key_type& x) const;
  iterator  upper_bound(const key_type& x) const;
  pair<iterator, iterator> equal_range(const key_type& x) const;
};
```

```
template <class Key, class Compare, class Allocator>
bool operator==(const set<Key, Compare, Allocator>& x,
  const set<Key, Compare, Allocator>& y);

template <class Key, class Compare, class Allocator>
bool operator< (const set<Key, Compare, Allocator>& x,
  const set<Key, Compare, Allocator>& y);
```

23.2.1.1 Typedefs [lib.set.typedefs]

1 `iterator` is a constant bidirectional iterator referring to `const value_type`. The exact type is implementation dependent and determined by `Allocator`.

2 `const_iterator` is the same type as `iterator`.

3 `size_type` is an unsigned integral type. The exact type is implementation dependent and determined by `Allocator`.

4 `difference_type` is a signed integral type. The exact type is implementation dependent and determined by `Allocator`.

23.2.2 Template class multiset [lib.multiset]

1 `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.

```
template <class Key, class Compare = less<Key>,
  template <class U> class Allocator = allocator>
class multiset {
public:
// typedefs:
  typedef Key     key_type;
  typedef Key     value_type;
  typedef Compare key_compare;
  typedef Compare value_compare;
  typedef ? iterator;
  typedef iterator const_iterator;
  typedef ? size_type;
  typedef ? difference_type;

// allocation/deallocation:
  multiset(const Compare& comp = Compare());
  template <class InputIterator>
    multiset(InputIterator first, InputIterator last,
             const Compare& comp = Compare());
  multiset(const multiset<Key, Compare, Allocator>& x);
 ~multiset();
  multiset<Key, Compare, Allocator>&
      operator=(const multiset<Key, Compare, Allocator>& x);

// accessors:
  key_compare   key_comp() const;
  value_compare value_comp() const;
  iterator      begin() const;
  iterator      end() const;
  bool          empty() const;
  size_type     size() const;
  size_type     max_size() const;
// insert/erase:
  iterator insert(const value_type& x);
  iterator insert(iterator position, const value_type& x);
  template <class InputIterator>
      void insert(InputIterator first, InputIterator last);
  void      erase(iterator position);
  size_type erase(const key_type& x);
  void      erase(iterator first, iterator last);

// multiset operations:
  iterator  find(const key_type& x) const;
  size_type count(const key_type& x) const;
  iterator  lower_bound(const key_type& x) const;
  iterator  upper_bound(const key_type& x) const;
  pair<iterator, iterator> equal_range(const key_type& x) const;
};

template <class Key, class Compare, class Allocator>
bool operator==(const multiset<Key, Compare, Allocator>& x,
  const multiset<Key, Compare, Allocator>& y);

template <class Key, class Compare, class Allocator>
bool operator< (const multiset<Key, Compare, Allocator>& x,
  const multiset<Key, Compare, Allocator>& y);
```

23.2.2.1 Typedefs [lib.multiset.typedefs]

1 `iterator` is a constant bidirectional iterator referring to `const value_type`. The exact type is implementation dependent and determined by `Allocator`.

2 `const_iterator` is the same type as `iterator`.

3 `size_type` is an unsigned integral type. The exact type is implementation dependent and determined by `Allocator`.

4 `difference_type` is a signed integral type. The exact type is implementation dependent and determined by `Allocator`.

23.2.3 Template class `map` [lib.map]

1 `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.

```
template <class Key, class T, class Compare = less<Key>,
  template <class U> class Allocator = allocator>
class map {
public:
// typedefs:
  typedef Key key_type;
  typedef pair<const Key, T> value_type;
  typedef Compare key_compare;
  class value_compare
    : public binary_function<  value_type, value_type, bool> {
  friend class map;
  protected:
    Compare comp;
    value_compare(Compare c) : comp(c) {}
  public:
    bool operator()(const value_type& x, const value_type& y) {
      return comp(x.first, y.first);
    }
  };

  typedef ? iterator;
  typedef ? const_iterator;
  typedef ? size_type;
  typedef ? difference_type;

// allocation/deallocation:
  map(const Compare& comp = Compare());
  template <class InputIterator>
    map(InputIterator first, InputIterator last,
        const Compare& comp = Compare());
  map(const map<Key, T, Compare, Allocator>& x);
 ~map();
  map<Key, T, Compare, Allocator>&
      operator=(const map<Key, T, Compare, Allocator>& x);

// accessors:
  key_compare   key_comp() const;
  value_compare value_comp() const;
  iterator       begin();
  const_iterator begin() const;
  iterator       end();
  const_iterator end() const;
  bool empty() const;
  size_type size() const;
  size_type max_size() const;
  T& operator[](const key_type& x);

// insert/erase:
  pair<iterator, bool> insert(const value_type& x);
  iterator             insert(iterator position, const value_type& x);
  template <class InputIterator>
      void insert(InputIterator first, InputIterator last);
  void      erase(iterator position);
  size_type erase(const key_type& x);
  void      erase(iterator first, iterator last);

// map operations:
  iterator       find(const key_type& x);
  const_iterator find(const key_type& x) const;
  size_type      count(const key_type& x) const;
  iterator       lower_bound(const key_type& x);
  const_iterator lower_bound(const key_type& x) const;
  iterator       upper_bound(const key_type& x);
  const_iterator upper_bound(const key_type& x) const;
  pair<iterator, iterator>             equal_range(const key_type& x);
  pair<const_iterator, const_iterator> equal_range(const key_type& x) const;
};
```

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

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

23.2.3.1 Typedefs [lib.map.typedefs]

1 `iterator` is a bidirectional iterator referring to `value_type`. The exact type is implementation dependent and determined by `Allocator`.

2 `const_iterator` is the a constant bidirectional iterator referring to `const value_type`. The exact type is implementation dependent and determined by `Allocator`. It is guaranteed that there is a constructor for `const_iterator` out of `iterator`.

3 `size_type` is an unsigned integral type. The exact type is implementation dependent and determined by `Allocator`.

4 `difference_type` is a signed integral type. The exact type is implementation dependent and determined by `Allocator`.

23.2.3.2 Member functions [lib.map.members]

1 In addition to the standard set of member functions of associative containers, `map` provides `T& operator[](const key_type&)`. For a map `m` and key `k`, `m[k]` is semantically equivalent to `(*((m.insert(make_pair(k, T()))).first)).second`.

23.2.4 Template class `multimap` [lib.multimap]

1 `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.

```
template <class Key, class T, class Compare = less<Key>,
           template <class U> class Allocator = allocator>
class multimap {
public:
// typedefs:
  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 ? iterator;
  typedef ? const_iterator;
  typedef ? size_type;
  typedef ? difference_type;
// allocation/deallocation:
  multimap(const Compare& comp = Compare());
  template <class InputIterator>
    multimap(InputIterator first, InputIterator last,
             const Compare& comp = Compare());
  multimap(const multimap<Key, T, Compare, Allocator>& x);
 ~multimap();
  multimap<Key, T, Compare, Allocator>&
      operator=(const multimap<Key, T, Compare, Allocator>& x);

// accessors:
  key_compare    key_comp() const;
  value_compare  value_comp() const;
  iterator       begin();
  const_iterator begin() const;
  iterator       end();
  const_iterator end() const;
  bool           empty() const;
  size_type      size() const;
  size_type      max_size() const;

// insert/erase:
  iterator insert(const value_type& x);
  iterator insert(iterator position, const value_type& x);
  template <class InputIterator>
      void insert(InputIterator first, InputIterator last);
  void      erase(iterator position);
  size_type erase(const key_type& x);
  void      erase(iterator first, iterator last);

// multimap operations:
  iterator       find(const key_type& x);
  const_iterator find(const key_type& x) const;
  size_type      count(const key_type& x) const;
  iterator       lower_bound(const key_type& x);
  const_iterator lower_bound(const key_type& x) const;
  iterator       upper_bound(const key_type& x);
  const_iterator upper_bound(const key_type& x) const;
  pair<iterator, iterator>             equal_range(const key_type& x);
  pair<const_iterator, const_iterator> equal_range(const key_type& x) const;
};

template <class Key, class T, class Compare, class Allocator>
bool operator==(const multimap<Key, T, Compare, Allocator>& x,
  const multimap<Key, T, Compare, Allocator>& y);

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

23.2.4.1 Typedefs [lib.multimap.typedefs]

1 `iterator` is a bidirectional iterator referring to `value_type`. The exact type is implementation dependent and determined by `Allocator`.

2 `const_iterator` is the a constant bidirectional iterator referring to `const value_type`. The exact type is implementation dependent and determined by `Allocator`. It is guaranteed that there is a constructor for `const_iterator` out of `iterator`.

3 `size_type` is an unsigned integral type. The exact type is implementation dependent and determined by `Allocator`.

4 `difference_type` is a signed integral type. The exact type is implementation dependent and determined by `Allocator`.

24 Iterators library [lib.iterators]

1 This clause describes components that C++ programs may use to perform iterations over containers (23), streams (27.2), and stream buffers (27.2.1).

2 The following subclauses describe components for iterator tags (24.1), predefined iterators (24.2), stream iterators (24.3), and streambuf iterators (24.4).

3 Headers:

— `<stl iterators (TBD)>`

4 Table 85:

Table 85—Header `<stl iterators (TBD)>` synopsis

Type	Name(s)	
Template classes:		
`back_insert_iterator`	`ostream_iterator`	
`front_insert_iterator`	`reverse_bidirectional_iterator`	
`insert_iterator`	`reverse_iterator`	
`istream_iterator`		
Template structs:		
`bidirectional_iterator`	`input_iterator`	
`forward_iterator`	`random_access_iterator`	
Template operators:		
`operator+ (reverse_iterator)`	`operator== (reverse_bidir_iter)`	
`operator- (reverse_iterator)`	`operator== (reverse_iterator)`	
`operator< (reverse_iterator)`	`operator== (istream_iterator)`	
Template functions:		
`advance`	`distance_type [5]`	`iterator_category [5]`
`back_inserter`	`front_inserter`	`value_type [5]`
`distance`	`inserter`	
Structs:		
`bidirectional_iterator_tag`	`output_iterator_tag`	
`forward_iterator_tag`	`random_access_iterator_tag`	
`input_iterator_tag`		
`output_iterator`		
Function:	`iterator_category(output_iterator)`	

24.1 Iterator tags [lib.iterator.tags]

1 To implement algorithms only in terms of iterators, it is often necessary to infer both of the value type and the distance type from the iterator. To enable this task it is required that for an iterator `i` of any category other than output iterator, the expression `value_type(i)` returns `(T*)(0)` and the expression `distance_type(i)` returns `(Distance*)(0)`. For output iterators, these expressions are not required.

24.1.1 Examples of using iterator tags [lib.examples]

1 For all the regular pointer types we can define `value_type` and `distance_type` with the help of:

```
  template <class T>
  inline T* value_type(const T*) { return (T*)(0); }

  template <class T>
  inline ptrdiff_t* distance_type(const T*) { return (ptrdiff_t*)(0); }
```

2 Then, if we want to implement a generic `reverse` function, we do the following:

```
template <class BidirectionalIterator>
inline void reverse(BidirectionalIterator first, BidirectionalIterator last) {
  __reverse(first, last, value_type(first), distance_type(first));
}
```

3 where `__reverse` is defined as:

```
  template <class BidirectionalIterator, class T, class Distance>
  void __reverse(BidirectionalIterator first, BidirectionalIterator last, T*,
                 Distance*)
  {
    Distance n;
    distance(first, last, n); // see Iterator operations section
    --n;
    while (n > 0) {
      T tmp = *first;
      *first++ = *--last;
      *last = tmp;
      n -= 2;
    }
  }
```

4 If there is an additional pointer type `far` such that the difference of two far pointers is of the type `long`, we define:

```
  template <class T>
  inline T* value_type(const T far *) { return (T*)(0); }

  template <class T>
  inline long* distance_type(const T far *) { return (long*)(0); }
```

5 It is often desirable for a template function to find out what is the most specific category of its iterator argument, so that the function can select the most efficient algorithm at compile time. To facilitate this, the library introduces *category tag* classes which are used as compile time tags for algorithm selection. They are: `input_iterator_tag`, `output_iterator_tag`, `forward_iterator_tag`, `bidirectional_iterator_tag` and `random_access_iterator_tag`. Every iterator `i` must have an expression `iterator_category(i)` defined on it that returns the most specific category tag that describes its behavior. For example, we define that all the pointer types are in the random access iterator category by:

```
  template <class T>
  inline random_access_iterator_tag iterator_category(T*) {
    return random_access_iterator_tag();
  }
```

6 For a user-defined iterator `BinaryTreeIterator`, it can be included into the bidirectional iterator category by saying:

```
  template <class T>
  inline bidirectional_iterator_tag iterator_category(
    const BinaryTreeIterator<T>&) {
    return bidirectional_iterator_tag();
  }
```

7 If a template function `evolve` is well defined for bidirectional iterators, but can be implemented more efficiently for random access iterators, then the implementation is like:

```
  template <class BidirectionalIterator>
  inline void evolve(BidirectionalIterator first, BidirectionalIterator last) {
    evolve(first, last, iterator_category(first));
  }

  template <class BidirectionalIterator>
  void evolve(BidirectionalIterator first, BidirectionalIterator last,
              bidirectional_iterator_tag) {
  // ... more generic, but less efficient algorithm
  }

  template <class RandomAccessIterator>
  void evolve(RandomAccessIterator first, RandomAccessIterator last,
    random_access_iterator_tag) {
  // ... more efficient, but less generic algorithm
  }
```

8 If a user wants to define a bidirectional iterator for some data structure containing `double` and such that it works on a large memory model of his computer, he can do it with:

```
  class MyIterator : public bidirectional_iterator<double, long> {

  // code implementing ++, etc.

};
```

9 Then there is no need to define `iterator_category`, `value_type`, and `distance_type` on `MyIterator`.

24.1.2 Library defined primitives [lib.library.primitives]

1 To simplify the task of defining the `iterator_category`, `value_type` and `distance_type` for user definable iterators, the library provides the following predefined classes and functions:

24.1.2.1 Standard iterator tags [lib.std.iterator.tags]

```
struct input_iterator_tag : empty {};
struct output_iterator_tag : empty {};
struct forward_iterator_tag : empty {};
struct bidirectional_iterator_tag : empty {};
struct random_access_iterator_tag : empty {};
```

24.1.2.2 Basic iterators [lib.basic.iterators]

```
template <class T, class Distance = ptrdiff_t> struct input_iterator : empty{};
struct output_iterator : empty{};

// output_iterator is not a template because output iterators
// do not have either value type or distance type defined.

template <class T, class Distance = ptrdiff_t> struct forward_iterator : empty{};
template <class T, class Distance = ptrdiff_t> struct bidirectional_iterator
  : empty {};
template <class T, class Distance = ptrdiff_t> struct random_access_iterator
  : empty {};
```

24.1.2.3 `iterator_category` [lib.iterator.category]

```
// iterator_category

template <class T, class Distance>
inline input_iterator_tag
iterator_category(const input_iterator<T, Distance>&) {
  return input_iterator_tag();
}

inline output_iterator_tag iterator_category(const output_iterator&) {
  return output_iterator_tag();
}

template <class T, class Distance>
inline forward_iterator_tag
iterator_category(const forward_iterator<T, Distance>&) {
  return forward_iterator_tag();
}

template <class T, class Distance>
inline bidirectional_iterator_tag
iterator_category(const bidirectional_iterator<T, Distance>&) {
  return bidirectional_iterator_tag();
}

template <class T, class Distance>
inline random_access_iterator_tag
iterator_category(const random_access_iterator<T, Distance>&) {
  return random_access_iterator_tag();
}

template <class T>
inline random_access_iterator_tag iterator_category(const T*) {
  return random_access_iterator_tag();
}
```

24.1.2.4 `value_type` [lib.value.type]

```
template <class T, class Distance>
inline T* value_type(const input_iterator<T, Distance>&) {
  return (T*)(0);
}
```

```
template <class T, class Distance>
inline T* value_type(const forward_iterator<T, Distance>&) {
  return (T*)(0);
}

template <class T, class Distance>
inline T* value_type(const bidirectional_iterator<T, Distance>&) {
  return (T*)(0);
}

template <class T, class Distance>
inline T* value_type(const random_access_iterator<T, Distance>&) {
  return (T*)(0);
}

template <class T>
inline T* value_type(const T*) { return (T*)(0); }
```

24.1.2.5 `distance_type` [lib.distance.type]

```
// distance_type of iterator

template <class T, class Distance>
inline Distance* distance_type(const input_iterator<T, Distance>&) {
  return (Distance*)(0);
}

template <class T, class Distance>
inline Distance* distance_type(const forward_iterator<T, Distance>&) {
  return (Distance*)(0);
}

template <class T, class Distance>
inline Distance*
distance_type(const bidirectional_iterator<T, Distance>&) {
  return (Distance*)(0);
}

template <class T, class Distance>
inline Distance*
distance_type(const random_access_iterator<T, Distance>&) {
  return (Distance*)(0);
}

template <class T>
inline ptrdiff_t* distance_type(const T*) { return (ptrdiff_t*)(0); }
```

24.1.3 Iterator operations [lib.iterator.operations]

1 Since only random access iterators provide `+` and `-` operators, the library provides two template functions `advance` and `distance`. These functions use `+` and `-` for random access iterators (and are, therefore, constant time for them); for input, forward and bidirectional iterators they use `++` to provide linear time implementations. `advance` takes negative argument `n` for random access and bidirectional iterators only. `advance` increments (or decrements for negative `n`) iterator reference `i` by `n`. `distance` increments `n` by the number of times it takes to get from `first` to `last`.

```
template <class InputIterator, class Distance>
inline void advance(InputIterator& i, Distance n);

template <class InputIterator, class Distance>
inline void distance(InputIterator first, InputIterator last, Distance& n);
```

2 `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*`.

24.2 Predefined iterators [lib.predef.iterators]

24.2.1 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.

24.2.1.1 Template class `reverse_bidirectional_iterator` [lib.reverse.bidir.iter]

```
template <class BidirectionalIterator, class T, class Distance = ptrdiff_t>
class reverse_bidirectional_iterator
  : public bidirectional_iterator<T, Distance> {
protected:
  BidirectionalIterator current;
public:
  reverse_bidirectional_iterator() {}
  reverse_bidirectional_iterator(BidirectionalIterator x) : current(x) {}
  operator BidirectionalIterator() { return current; }

  T operator*() {
    BidirectionalIterator tmp = current;
    return *--tmp;
  }

  reverse_bidirectional_iterator<BidirectionalIterator, T, Distance>&
  operator++() {
    --current;
    return *this;
  }
  reverse_bidirectional_iterator<BidirectionalIterator, T, Distance>
  operator++(int) {
    reverse_bidirectional_iterator<BidirectionalIterator, T, Distance>
      tmp = *this;
    --current;
    return tmp;
  }

  reverse_bidirectional_iterator<BidirectionalIterator, T, Distance>&
  operator--() {
    ++current;
    return *this;
  }

  reverse_bidirectional_iterator<BidirectionalIterator, T, Distance>
  operator--(int) {
    reverse_bidirectional_iterator<BidirectionalIterator, T, Distance>
      tmp = *this;
    ++current;
    return tmp;
  }
};

template <class BidirectionalIterator, class T, class Distance>
inline bool
  operator==(
    const reverse_bidirectional_iterator<BidirectionalIterator,T,Distance>& x,
    const reverse_bidirectional_iterator<BidirectionalIterator,T,Distance>& y)
  {
    return x.current == y.current;
  }
```

24.2.1.2 Template class `reverse_iterator` [lib.reverse.iterator]

```
template <class RandomAccessIterator, class T, class Distance = ptrdiff_t>
class reverse_iterator : public random_access_iterator<T, Distance> {
protected:
  RandomAccessIterator current;
public:
  reverse_iterator() {}
  reverse_iterator(RandomAccessIterator x) : current(x) {}

  operator RandomAccessIterator() { return current; }
  T operator*() {
    RandomAccessIterator tmp = current;
    return *--tmp;
  }

  reverse_iterator<RandomAccessIterator, T, Distance>& operator++() {
    --current;
    return *this;
  }

  reverse_iterator<RandomAccessIterator, T, Distance> operator++(int) {
    reverse_iterator<RandomAccessIterator, T, Distance> tmp = *this;
    --current;
    return tmp;
  }
  reverse_iterator<RandomAccessIterator, T, Distance>& operator--() {
    ++current;
    return *this;
  }

  reverse_iterator<RandomAccessIterator, T, Distance> operator--(int) {
    reverse_iterator<RandomAccessIterator, T, Distance> tmp = *this;
    ++current;
    return tmp;
  }

  reverse_iterator<RandomAccessIterator, T, Distance>
  operator+(Distance n)   const {
    return reverse_iterator<RandomAccessIterator, T, Distance>(current - n);
  }

  reverse_iterator<RandomAccessIterator, T, Distance>&
  operator+=(Distance n) const {
    current -= n;
    return *this;
  }

  reverse_iterator<RandomAccessIterator, T, Distance>
  operator-(Distance n) const {
    return reverse_iterator<RandomAccessIterator, T, Distance>(current + n);
  }

  reverse_iterator<RandomAccessIterator, T, Distance>
  operator-(Distance n) const {
    current += n;
    return *this;
  }
  T operator[](Distance n) { return *(*this + n); }
};

template <class RandomAccessIterator, class T, class Distance>
inline bool operator==(
  const reverse_iterator<RandomAccessIterator, T, Distance>& x,
  const reverse_iterator<RandomAccessIterator, T, Distance>& y)
{
  return x.current == y.current;
}

template <class RandomAccessIterator, class T, class Distance>
inline bool operator<(
  const reverse_iterator<RandomAccessIterator, T, Distance>& x,
  const reverse_iterator<RandomAccessIterator, T, Distance>& y)
{
  return y.current < x.current;
}

template <class RandomAccessIterator, class T, class Distance>
inline Distance operator-(
  const reverse_iterator<RandomAccessIterator, T, Distance>& x,
  const reverse_iterator<RandomAccessIterator, T, Distance>& y)
{
  return y.current - x.current;
}
```

```
template <class RandomAccessIterator, class T, class Distance>
inline reverse_iterator<RandomAccessIterator, T, Distance> operator+(
  Distance n,
  const reverse_iterator<RandomAccessIterator, T, Distance>& x)
{
  return reverse_iterator<RandomAccessIterator, T, Distance>(current - n);
}
```

1 There is no way a default for `T` can be expressed in terms of `BidirectionalIterator` because the value type cannot be deduced from built-in iterators such as `int*`. Otherwise, we would have written

```
template <class BidirectionalIterator,
  class T = BidirectionalIterator::reference_type,
  class Distance = BidirectionalIterator::difference_type>
class reverse_bidirectional_iterator: bidirectional_iterator<T, Distance> {

/* ... */

};
```

24.2.2 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.

24.2.2.1 Template class `back_insert_iterator` [lib.back.insert.iterator]

```
template <class Container>
class back_insert_iterator : public output_iterator {
protected:
  Container& container;

public:
  back_insert_iterator(Container& x) : container(x) {}
  back_insert_iterator<Container>&
  operator=(const Container::value_type& value) {
    container.push_back(value);
    return *this;
  }
  back_insert_iterator<Container>& operator*() { return *this; }
  back_insert_iterator<Container>& operator++() { return *this; }
  back_insert_iterator<Container> operator++(int) { return *this; }
};

template <class Container>
back_insert_iterator<Container> back_inserter(Container& x) {
  return back_insert_iterator<Container>(x);
}
```

24.2.2.2 Template class `front_insert_iterator` [lib.front.insert.iterator]

```
template <class Container>
class front_insert_iterator : public output_iterator {
protected:
  Container& container;

public:
  front_insert_iterator(Container& x) : container(x) {}
  front_insert_iterator<Container>&
  operator=(const Container::value_type& value) {
    container.push_front(value);
    return *this;
  }

  front_insert_iterator<Container>& operator*() { return *this; }
  front_insert_iterator<Container>& operator++() { return *this; }
  front_insert_iterator<Container> operator++(int) { return *this; }
};

template <class Container>
front_insert_iterator<Container> front_inserter(Container& x) {
  return front_insert_iterator<Container>(x);
}
```

24.2.2.3 Template class `insert_iterator` [lib.insert.iterator]

```
template <class Container>
class insert_iterator : public output_iterator {
protected:
  Container& container;
  Container::iterator iter;

public:
  insert_iterator(Container& x, Container::iterator i)
    : container(x), iter(i) {}

  insert_iterator<Container>& operator=(const Container::value_type& value) {
    iter = container.insert(iter, value);
    ++iter;
    return *this;
  }

  insert_iterator<Container>& operator*() { return *this; }
  insert_iterator<Container>& operator++() { return *this; }
  insert_iterator<Container> operator++(int) { return *this; }
};
```

```
template <class Container, class Iterator>
insert_iterator<Container> inserter(Container& x, Iterator i) {
  return insert_iterator<Container>(x, Container::iterator(i));
}
```

24.3 Stream iterators [lib.stream.iterators]

1 To make it possible for algorithmic templates to work directly with input/output streams, appropriate iterator-like template classes are provided. For example,

```
partial_sum_copy(istream_iterator<double>(cin),  istream_iterator<double>(),
  ostream_iterator<double>(cout, "n"));
```

2 reads a file containing floating point numbers from cin, and prints the partial sums onto cout.

24.3.1 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.

```
template <class T, class Distance = ptrdiff_t>
class istream_iterator : input_iterator<T, Distance> {
public:
  istream_iterator();
  istream_iterator(istream& s);
  istream_iterator(const istream_iterator<T, Distance>& x);
 ~istream_iterator();

  const T& operator*() const;
  istream_iterator<T, Distance>& operator++();
  istream_iterator<T, Distance> operator++(int);
};

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

24.3.2 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:

```
template <class T>
class ostream_iterator : public output_iterator {
public:
  ostream_iterator(ostream& s);
  ostream_iterator(const char* delimiter);
  ostream_iterator(ostream& s, const char* delimiter);
  ostream_iterator(const ostream_iterator<T>& x);
 ~ostream_iterator();
  ostream_iterator<T>& operator=(const T& value);

  ostream_iterator<T>& operator*();
  ostream_iterator<T>& operator++();
  ostream_iterator<T> operator++(int);
};
```

24.4 Streambuf iterators [lib.streambuf.iterators]

Box 99

TO BE SPECIFIED.

`istreambuf_iterator` is currently part of `<istream>`, subclause _lib.input.streams_.

`ostreambuf_iterator` is currently part of `<ostream>`, subclause _lib.output.streams_.

Both should be moved here.

25 Algorithms library [lib.algorithms]

1 This clause describes components that C++ programs may use to perform algorithmic operations on containers (23) and other sequences.

2 The following subclauses describe components for non-mutating sequence operation (25.1), mutating sequence operations (25.2), and sorting and related operations (25.3).

3 Headers:

— `<stl algorithms (TBD)>`

— `<cstdlib>` (partial)

4 Table 86:

Table 86—Header `<stl algorithms (TBD)>` synopsis

Type	Name(s)
Template functions:	
`adjacent_find [2]`	`prev_permutation [2]`
`binary_search [2]`	`push_heap [2]`
`copy`	`random_shuffle [2]`
`copy_backward`	`remove`
`count`	`remove_copy`
`count_if`	`remove_copy_if`
`equal [2]`	`remove_if`
`equal_range [2]`	`replace`
`fill`	`replace_copy`
`fill_n`	`replace_copy_if`
`find`	`replace_if`
`find_if`	`reverse`
`for_each`	`reverse_copy`
`generate`	`rotate`
`generate_n`	`rotate_copy`
`includes [2]`	`search [2]`
`inplace_merge [2]`	`set_difference [2]`
`lexicographical_compare [2]`	`set_intersection [2]`
`lower_bound [2]`	`set_symmetric_difference [2]`
`make_heap [2]`	`set_union [2]`
`max [2]`	`sort [2]`
`max_element [2]`	`sort_heap [2]`
`merge [2]`	`stable_partition`
`min [2]`	`stable_sort [2]`
`min_element [2]`	`swap`
`mismatch [2]`	`swap_ranges`
`next_permutation [2]`	`transform [2]`
`nth_element [2]`	`unique [2]`
`partial_sort [2]`	`unique_copy [2]`
`partial_sort_copy [2]`	`upper_bound [2]`
`partition`	
`pop_heap [2]`	

5 Table 87:

Table 87—Header `<cstdlib>` synopsis

Type	Name(s)	
Functions:	`bsearch`	`qsort`

SEE ALSO: ISO C subclause 7.10.5.

6 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 user defined data structures, as long as these data structures have iterator types satisfying the assumptions on the algorithms.

7 Both in-place and copying versions are provided for certain algorithms. 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`. 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`).

8 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.

9 The `BinaryPredicate` class is used whenever an algorithm expects a function object that when applied to the result of dereferencing two corresponding iterators or to dereferencing an iterator and type `T` when `T` is part of the signature returns a value testable as `true`. In other words, if an algorithm takes `BinaryPredicate binary_pred` as its argument and `first1` and `first2` as its iterator arguments, it should work correctly in the construct `if (`*`pred`*`(*first, *first2)){...}`. `BinaryPredicate` always takes the first iterator type as its first argument, that is, in those cases when `T value` is part of the signature, it should work correctly in the context of `if (`*`pred`*`(*first, `*`value`*`)){...}`. It is expected that `binary_pred` will not apply any non-constant function through the dereferenced iterators.

10 In the description of the algorithms operators `+` and `-` are used for some of the iterator categories for which they do not have to be defined. In these cases the semantics of `a+n` is the same is that of

```
{ X tmp = a;
  advance(tmp, n);
  return tmp;
}
```

and that of `a-b` is the same as of

```
{ Distance n;
  distance(a, b, n);
  return n;
}
```

Box 100
For the following algorithms: `reverse`, `rotate`, `random_shuffle`, `partition`, `stable_partition`, `sort`, `stable_sort` and `inplace_merge` the iterator requirement can be relaxed to `ForwardIterator`. These algorithms could then be dispatched upon the iterator category tags to use the most efficient implementation for each iterator category. We have not included the relaxation at this stage since it is not yet fully implemented.

25.1 Non-mutating sequence operations [lib.alg.nonmutating]

25.1.1 For each [lib.alg.foreach]

```
template <class InputIterator, class Function>
void for_each(InputIterator first, InputIterator last, Function f);
```

1 Applies `f` to the result of dereferencing every iterator in the range `[`*`first`*`, `*`last`*`)`. `f` shall not apply any non-constant function through the dereferenced iterator.

2 Complexity: `f` is applied exactly `last - first` times.

3 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);
```

1 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.

2 Complexity: At most `last - first` applications of the corresponding predicate are done.

25.1.3 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);
```

1 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.

2 Complexity: At most `max((last - first) - 1, 0)` applications of the corresponding predicate are done.

25.1.4 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);
```

1 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`.

2 Complexity: Exactly `last - first` applications of the corresponding predicate are done.

3 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*`.

25.1.5 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);
```

1 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.

2 Complexity: At most `last1 - first1` applications of the corresponding predicate are done.

25.1.6 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);
```

1 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`.

2 Complexity: At most `last1 - first1` applications of the corresponding predicate are done.

25.1.7 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);
```

1 Finds a subsequence of equal values in a sequence.

2 Returns the first iterator `i` in the range `[first1, last1 - (last2 - first2))` such that for any non-negative integer `n` less than `last2 - first2` the following corresponding conditions hold: `*(i + n) == *(first2 + n)`, `pred(*(i + n), *(first2 + n)) == true`. Returns `last1` if no such iterator is found.[112)]

25.2 Mutating sequence operations [lib.alg.mutating.operations]

25.2.1 Copy [lib.alg.copy]

25.2.1.1 `copy` [lib.copy]

```
template <class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last,
                    OutputIterator result);
```

1 Copies elements. For each non-negative integer `n < (last - first)`, `*(result + n) = *(first + n)` is performed. `copy` returns `result + (last - first)`.

2 `result` shall not be in the range `[first, last)`.

112) The Knuth-Morris-Pratt algorithm is not used here. While the KMP algorithm guarantees linear time, it tends to be slower in most practical cases than the naive algorithm with worst-case quadratic behavior. The worst case is extremely unlikely. We expect that most implementations will provide a specialization:

```
char* search(char* first1, char* last1,  char* first2,  char* last2);
```

3 that will use a variation of the Boyer-Moore algorithm for fast string searching.

3 Complexity: Exactly `last - first` assignments are done.

25.2.1.2 `copy_backward` [lib.copy.backward]

```
template <class BidirectionalIterator1, class BidirectionalIterator2>
BidirectionalIterator2
    copy_backward(BidirectionalIterator1 first,
                  BidirectionalIterator1 last, BidirectionalIterator2 result);
```

1 Copies elements in the range `[first, last)` into the range `[result - (last - first), result)` starting from `last - 1` and proceeding to `first`.[113] For each positive integer `n <= (last - first)`, `*(result - n) = *(last - n)` is performed.

2 `result` shall not be in the range `[first, last)`.

3 Returns `result - (last - first)`.

4 Complexity: Exactly `last - first` assignments are done.

25.2.2 Swap [lib.alg.swap]

25.2.2.1 `swap` [lib.swap]

```
template <class T>
void swap(T& a, T& b);
```

1 Exchanges values stored in two locations.

25.2.2.2 `swap_ranges` [lib.swap.ranges]

```
template <class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 swap_ranges(ForwardIterator1 first1, ForwardIterator1 last1,
                             ForwardIterator2 first2);
```

1 For each non-negative integer `n < (last1 - first1)` the swap is performed: `swap(*(first1 + n), *(first2 + n))`.

2 The two ranges `[first1, last1)` and `[first2, first2 + (last1 - first1))` shall not overlap.

3 Returns `first2 + (last1 - first1)`.

4 Complexity: Exactly `last1 - first1` swaps are done.

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);
```

[113] `copy_backward` (_lib.copy.backward_) should be used instead of copy when `last` is in the range `[result - (last - first), result)`.

1 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*`)))`.

2 `op` and `binary_op` shall not have any side effects.

3 Returns `result + (last1 - first1)`.

4 Complexity: Exactly `last1 - first1` applications of `op` or `binary_op` are performed.

5 Notes: `result` may be equal to `first` in case of unary transform, or to `first1` or `first2` in case of binary transform.

25.2.4 Replace [lib.alg.replace]

25.2.4.1 `replace` [lib.replace]

```
template <class ForwardIterator, class T>
void replace(ForwardIterator first, ForwardIterator last, const T& old_value,
             const T& new_value);

template <class ForwardIterator, class Predicate, class T>
void replace_if(ForwardIterator first, ForwardIterator last, Predicate pred,
                const T& new_value);
```

1 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`.

2 Complexity: Exactly `last - first` applications of the corresponding predicate are done.

25.2.4.2 `replace_copy` [lib.replace.copy]

```
template <class InputIterator, class OutputIterator, class T>
OutputIterator replace_copy(InputIterator first, InputIterator last,
                            OutputIterator result,
                            const T& old_value, const T& new_value);

template <class Iterator, class OutputIterator, class Predicate, class T>
OutputIterator replace_copy_if(Iterator first, Iterator last,
                               OutputIterator result,
                               Predicate pred, const T& new_value);
```

1 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`.

2 Returns `result + (last - first)`.

3 Complexity: Exactly `last - first` applications of the corresponding predicate are done.

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);
```

1 Assigns value through all the iterators in the range `[`*first*`,` *last*`)` or `[`*first*`, first + n)`.

2 Complexity: Exactly `last - first` (or `n`) assignments are done.

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);
```

1 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)`. `gen` takes no arguments.

2 Complexity: Exactly `last - first` (or `n`) invocations of `gen` and assignments are done.

25.2.7 Remove [lib.alg.remove]

25.2.7.1 `remove` [lib.remove]

```
template <class ForwardIterator, class T>
ForwardIterator remove(ForwardIterator first, ForwardIterator last,
                       const T& value);

template <class ForwardIterator, class Predicate>
ForwardIterator remove_if(ForwardIterator first, ForwardIterator last,
                          Predicate pred);
```

1 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`.

2 Returns the end of the resulting range. `remove` is stable, that is, the relative order of the elements that are not removed is the same as their relative order in the original range.

3 Complexity: Exactly `last - first` applications of the corresponding predicate are done.

25.2.7.2 `remove_copy` [lib.remove.copy]

```
template <class InputIterator, class OutputIterator, class T>
OutputIterator remove_copy(InputIterator first, InputIterator last,
                           OutputIterator result, const T& value);

template <class InputIterator, class OutputIterator, class Predicate>
OutputIterator remove_copy_if(InputIterator first, InputIterator last,
                              OutputIterator result, Predicate pred);
```

1 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`.

2 Returns the end of the resulting range.

3 Complexity: Exactly `last - first` applications of the corresponding predicate are done.

4 Notes: `remove_copy` is stable, that is, the relative order of the elements in the resulting range is the same as their relative order in the original range.

25.2.8 Unique [lib.alg.unique]

25.2.8.1 `unique` [lib.unique]

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

template <class ForwardIterator, class BinaryPredicate>
ForwardIterator unique(ForwardIterator first, ForwardIterator last,
                       BinaryPredicate pred);
```

1 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`.

2 Returns the end of the resulting range.

3 Complexity: Exactly `(last - first) - 1` applications of the corresponding predicate are done.

25.2.8.2 `unique_copy` [lib.unique.copy]

```
template <class InputIterator, class OutputIterator>
OutputIterator unique_copy(InputIterator first, InputIterator last,
                           OutputIterator result);

template <class InputIterator, class OutputIterator, class BinaryPredicate>
OutputIterator unique_copy(InputIterator first, InputIterator last,
                           OutputIterator result, BinaryPredicate pred);
```

1 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`.

2 Returns the end of the resulting range.

3 Complexity: Exactly `last - first` applications of the corresponding predicate are done.

25.2.9 Reverse [lib.alg.reverse]

25.2.9.1 `reverse` [lib.reverse]

```
template <class BidirectionalIterator>
void reverse(BidirectionalIterator first, BidirectionalIterator last);
```

1 For each non-negative integer `i <= (last - first)/2`, `reverse` applies `swap` to all pairs of iterators `first + i, (last - i) - 1`.

2 Complexity: Exactly `(last - first)/2` swaps are performed.

25.2.9.2 `reverse_copy` [lib.reverse.copy]

```
template <class BidirectionalIterator, class OutputIterator>
OutputIterator reverse_copy(BidirectionalIterator first,
                            BidirectionalIterator last, OutputIterator result);
```

1 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).`

2 The ranges `[`*first*`,` *last*`)` and `[`*result*`, result + (last - first))` shall not overlap.

3 Returns `result + (last - first)`.

4 Complexity: Exactly `last - first` assignments are done.

25.2.10 Rotate [lib.alg.rotate]

25.2.10.1 rotate [lib.rotate]

```
template <class BidirectionalIterator>
void rotate(BidirectionalIterator first, BidirectionalIterator middle,
            BidirectionalIterator last);
```

1 For each non-negative integer `i < (last - first)`, `rotate` places the element from the position `first + i` into position `first + (i + (middle - first)) % (last - first)`.

2 Complexity: At most `last - first` swaps are done.

25.2.10.2 rotate_copy [lib.rotate.copy]

```
template <class ForwardIterator, class OutputIterator>
OutputIterator rotate_copy(ForwardIterator first, ForwardIterator middle,
                           ForwardIterator last, OutputIterator result);
```

1 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))`. `rotate_copy` returns `result + (last - first)`.

2 The ranges `[first, last)` and `[result, result + (last - first))` shall not overlap.

3 Complexity: Exactly `last - first` assignments are done.

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);
```

1 Shuffles the elements in the range `[first, last)` with uniform distribution.

2 Complexity: Exactly `(last - first) - 1` swaps are done.

3 Notes: `random_shuffle` can take a particular random number generating function object `rand` such that `rand` returns a randomly chosen `double` in the interval `[0, 1)`.

25.2.12 Partitions [lib.alg.partitions]

25.2.12.1 partition [lib.partition]

```
template <class BidirectionalIterator, class Predicate>
BidirectionalIterator partition(BidirectionalIterator first,
                                BidirectionalIterator last, Predicate pred);
```

1 Places all the elements in the range `[first, last)` that satisfy `pred` before all the elements that do not satisfy it.

2 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`.

3 Complexity: At most `(last - first)/2` swaps. Exactly `last - first` applications of the predicate is done.

25.2.12.2 `stable_partition` [lib.stable.partition]

```
template <class BidirectionalIterator, class Predicate>
ForwardIterator stable_partition(BidirectionalIterator first,
                                 BidirectionalIterator last, Predicate pred);
```

1 Places all the elements in the range `[`*`first, last`*`)` that satisfy pred before all the elements that do not satisfy it.

2 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.

3 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 are done.

25.3 Sorting and related operations [lib.alg.sorting]

1 All the operations in this section have two versions: one that takes a function object of type `Compare` and one that uses an `operator<`.

2 `Compare` is used as a function object which returns `true` if the first argument is less than the second, and `false` otherwise. `Compare comp` is used throughout for algorithms assuming an ordering relation. It is assumed that `comp` will not apply any non-constant function through the dereferenced iterator. For all algorithms that take `Compare`, there is a version that uses `operator<` instead. That is, `comp(*i, *j) == true` defaults to `*i < *j == true`. For the algorithms to work correctly, `comp` has to induce a total ordering on the values.

3 A sequence is sorted with respect to a comparator `comp` if for any iterator `i` pointing to the sequence and any non-negative integer `n` such that `i + n` is a valid iterator pointing to an element of the sequence, `comp(*(i + n), *i) == false`.

4 In the descriptions of the functions that deal with ordering relationships we frequently use a notion of equality to describe concepts such as stability. The equality to which we refer is not necessarily an `operator==`, but an equality relation induced by the total ordering. That is, two element `a` and `b` are considered equal if and only if `!(a < b) && !(b < a)`.

25.3.1 Sorting [lib.alg.sort]

25.3.1.1 `sort` [lib.sort]

```
template <class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
void sort(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

1 Sorts the elements in the range `[`*`first, last`*`)`.

2 Complexity: Approximately `NlogN` (where `N` equals to `last - first`) comparisons on the average.[114)]

114) If the worst case behavior is important `stable_sort` (_lib.stable.sort_) or `partial_sort` (_lib.partial.sort_) should be used.

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);
```

1 Sorts the elements in the range [*first*, *last*).

2 Complexity: It does at most Nlog2N (where N equals to last - first) comparisons; if enough extra memory is available, it is NlogN.

3 Notes: Stable, the relative order of the equal elements is preserved.

25.3.1.3 partial_sort [lib.partial.sort]

```
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);
```

1 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.

2 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);
```

1 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)).

2 Returns either result_last or result_first + (last - first) whichever is smaller.

3 Complexity: Approximately (last - first) * log(min(last - *first*, result_last - result_first)) comparisons.

25.3.2 Nth element [lib.alg.nth.element]

```
template <class RandomAccessIterator>
void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                 RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                 RandomAccessIterator last, Compare comp);
```

1 After `nth_element` the element in the position pointed to by `nth` is the element that would be in that position if the whole range were sorted. Also for any iterator `i` in the range `[`*`first`*`, nth)` and any iterator `j` in the range `[nth, `*`last`*`)` it holds that `!(*i > *j)` or `comp(*i, *j) == false`.

2 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);
```

1 Finds the first position into which value can be inserted without violating the ordering. `lower_bound` 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`.

2 Complexity: At most `log(last - first) + 1` comparisons are done.

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);
```

1 Finds the furthermost position into which value can be inserted without violating the ordering. `upper_bound` 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`.

2 Complexity: At most `log(last - first) + 1` comparisons are done.

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);
```

1 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`.

2 Complexity: At most `2 * log(last - first)` comparisons are done.

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);
```

1 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`.

2 Complexity: At most `log(last - first) + 1` comparisons are done.

25.3.4 Merge [lib.alg.merge]

25.3.4.1 `merge` [lib.merge]

```
template <class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result);

template <class InputIterator1, class InputIterator2, class OutputIterator,
          class Compare>
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result, Compare comp);
```

1 Merges two sorted ranges `[first1, last1)` and `[first2, last2)` into the range `[result, result + (last1 - first1) + (last2 - first2))`.

2 The resulting range shall not overlap with either of the original ranges.

3 Returns `result + (last1 - first1) + (last2 - first2)`.

4 Complexity: At most `(last1 - first1) + (last2 - first2) - 1` comparisons are performed.

5 Notes: The merge is stable, that is, for equal elements in the two ranges, the elements from the first range always precede the elements from the second.

25.3.4.2 `inplace_merge` [lib.inplace.merge]

```
template <class BidirectionalIterator>
void inplace_merge(BidirectionalIterator first, BidirectionalIterator middle,
                   BidirectionalIterator last);

template <class BidirectionalIterator, class Compare>
void inplace_merge(BidirectionalIterator first, BidirectionalIterator middle,
                   BidirectionalIterator last, Compare comp);
```

1 Merges two sorted consecutive ranges `[`*`first`*`, middle)` and `[middle, `*`last`*`)` putting the result of the merge into the range `[`*`first`*`, `*`last`*`)`.

2 Complexity: At most `last - first` comparisons are performed. If no additional memory is available, the number of assignments can be equal to `NlogN` where `N` is equal to `last - first`.

3 Notes: The merge is stable, that is, for equal elements in the two ranges, the elements from the first range always precede the elements from the second.

25.3.5 Set operations on sorted structures [lib.alg.set.operations]

1 This section defines all the basic set operations on sorted structures. They even work with `multiset`s containing multiple copies of equal elements. The semantics of the set operations is generalized to multisets in a standard way by defining union to contain the maximum number of occurrences of every element, intersection to contain the minimum, and so on.

25.3.5.1 `includes` [lib.includes]

```
template <class InputIterator1, class InputIterator2>
bool includes(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2);

template <class InputIterator1, class InputIterator2, class Compare>
bool includes(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2, Compare comp);
```

1 Returns `true` if every element in the range `[`*`first2`*`, `*`last2`*`)` is contained in the range `[`*`first1`*`, last1)`. Returns `false` otherwise.

2 Complexity: At most `((last1 - first1) + (last2 - first2)) * 2 - 1` comparisons are performed.

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);
```

1 Constructs a sorted union of the elements from the two ranges.

2 The resulting range shall not overlap with either of the original ranges.

3 Returns the end of the constructed range.

4 Complexity: At most ((last1 - first1) + (last2 - first2)) * 2 - 1 comparisons are performed.

5 Notes: set_union is stable, that is, 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);
```

1 Constructs a sorted intersection of the elements from the two ranges.

2 The resulting range shall not overlap with either of the original ranges.

3 Returns the end of the constructed range.

4 Complexity: At most ((last1 - first1) + (last2 - first2)) * 2 - 1 comparisons are performed.

5 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);
```

1 Constructs a sorted difference of the elements from the two ranges.

2 The resulting range shall not overlap with either of the original ranges.

3 Returns the end of the constructed range.

4 Complexity: At most ((last1 - first1) + (last2 - first2)) * 2 - 1 comparisons are performed.

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);
```

1 Constructs a sorted symmetric difference of the elements from the two ranges.

2 The resulting range shall not overlap with either of the original ranges.

3 Returns the end of the constructed range.

4 Complexity: At most `((last1 - first1) + (last2 - first2)) * 2 - 1` comparisons are performed.

25.3.6 Heap operations [lib.alg.heap.operations]

1 A heap is a particular organization of elements in a range between two random access iterators `[a, b)`. Its two key properties are: (1) `*a` is the largest element in the range and (2) `*a` may be removed by `pop_heap`, or a new element added by `push_heap`, in `O(logN)` time. These properties make heaps useful as priority queues. `make_heap` converts a range into a heap and `sort_heap` turns a heap into a sorted sequence.

25.3.6.1 push_heap [lib.push.heap]

```
template <class RandomAccessIterator>
void push_heap(RandomAccessIterator first, RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
void push_heap(RandomAccessIterator first, RandomAccessIterator last,
               Compare comp);
```

1 The range `[`*first*`, last - 1)` shall be a valid heap.

2 Places the value in the location `last - 1` into the resulting heap `[`*first*`, `*last*`)`.

3 Complexity: At most `log(last - first)` comparisons are performed.

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);
```

1 The range `[`*first*`, `*last*`)` shall be a valid heap.

2 Swaps the value in the location `first` with the value in the location `last - 1` and makes `[`*first*`, last - 1)` into a heap.

3 Complexity: At most `2 * log(last - first)` comparisons are performed.

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);
```

1 Constructs a heap out of the range `[first, last)`.

2 Complexity: At most `3*(last - first)` comparisons are performed.

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);
```

1 Sorts elements in the heap `[first, last)`.

2 Complexity: At most `NlogN` comparisons are performed where `N` is equal to `last - first`.

3 Notes: Not stable.

25.3.7 Minimum and maximum [lib.alg.min.max]

25.3.7.1 min [lib.min]

```
template <class T>
T min(const T& a, const T& b);

template <class T, class Compare>
T min(const T& a, const T& b, Compare comp);
```

25.3.7.2 max [lib.max]

1 Returns the smaller value. Returns the first argument when their arguments are equal.

```
template <class T>
T max(const T& a, const T& b);

template <class T, class Compare>
T max(const T& a, const T& b, Compare comp);
```

2 Returns the larger value. Returns the first argument when their arguments are equal.

25.3.7.3 max_element [lib.max.element]

```
template <class InputIterator>
InputIterator max_element(InputIterator first, InputIterator last);

template <class InputIterator, class Compare>
InputIterator max_element(InputIterator first, InputIterator last, Compare comp);
```

1 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`.

2 Complexity: Exactly `max((last - first) - 1, 0)` applications of the corresponding comparisons are done.

25.3.7.4 `min_element` [lib.min.element]

```
template <class InputIterator>
InputIterator min_element(InputIterator first, InputIterator last);

template <class InputIterator, class Compare>
InputIterator min_element(InputIterator first, InputIterator last, Compare comp);
```

1 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`.

2 Complexity: Exactly `max((last - first) - 1, 0)` applications of the corresponding comparisons are done.

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);
```

1 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.

2 Complexity: At most `min((last1 - first1), (last2 - first2))` applications of the corresponding comparison are done.

25.3.9 Permutation generators [lib.alg.permutation.generators]

25.3.9.1 `next_permutation` [lib.next.permutation]

```
template <class BidirectionalIterator>
bool next_permutation(BidirectionalIterator first,
                      BidirectionalIterator last);

template <class BidirectionalIterator, class Compare>
bool next_permutation(BidirectionalIterator first,
                      BidirectionalIterator last,
                      Compare comp);
```

1 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`.

2 Complexity: At most `(last - first)/2` swaps are performed.

25.3.9.2 `prev_permutation` [lib.prev.permutation]

```
template <class BidirectionalIterator>
bool prev_permutation(BidirectionalIterator first,
                      BidirectionalIterator last);

template <class BidirectionalIterator, class Compare>
bool prev_permutation(BidirectionalIterator first,
                      BidirectionalIterator last,
                      Compare comp);
```

1 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`.

2 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`.

3 Complexity: At most `(last - first)/2` swaps are performed.

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 (26.1), numeric arrays (26.2), generalized numeric algorithms (26.3), and facilities included from the ISO C library (26.4).

26.1 Complex numbers [lib.complex]

1 Headers:

— `<complex>`

2 Table 88:

Table 88—Header `<complex>` synopsis

Type	Name(s)
Macro:	`__STD_COMPLEX`
Classes:	`double_complex`
`float_complex`	`long_double_complex`
Operator functions:	
`operator!= (double_complex) [3]`	`operator-= (float_complex)`
`operator!= (float_complex) [3]`	`operator-= (long_double_complex)`
`operator!= (long_double_complex) [3]`	`operator/  (double_complex) [3]`
`operator*  (double_complex) [3]`	`operator/  (float_complex) [3]`
`operator*  (float_complex) [3]`	`operator/  (long_double_complex) [3]`
`operator*  (long_double_complex) [3]`	`operator/= (double_complex)`
`operator*= (double_complex)`	`operator/= (float_complex)`
`operator*= (float_complex)`	`operator/= (long_double_complex)`
`operator*= (long_double_complex)`	`operator<< (double_complex)`
`operator+  (double_complex) [4]`	`operator<< (float_complex)`
`operator+  (float_complex) [4]`	`operator<< (long_double_complex)`
`operator+  (long_double_complex) [4]`	`operator== (double_complex) [3]`
`operator+= (double_complex)`	`operator== (float_complex) [3]`
`operator+= (float_complex)`	`operator== (long_double_complex) [3]`
`operator+= (long_double_complex)`	`operator>> (double_complex)`
`operator-  (double_complex) [4]`	`operator>> (float_complex)`
`operator-  (float_complex) [4]`	`operator>> (long_double_complex)`
`operator-  (long_double_complex) [4]`	
`operator-= (double_complex)`	

Type	Name(s)
Functions:	
`abs  (double_complex)`	`norm (double_complex)`
`abs  (float_complex)`	`norm (float_complex)`
`abs  (long_double_complex)`	`norm (long_double_complex)`
`arg  (double_complex)`	`polar(double_complex)`
`arg  (float_complex)`	`polar(float_complex)`
`arg  (long_double_complex)`	`polar(long_double_complex)`
`conj  (double_complex)`	`pow  (double_complex)`
`conj  (float_complex)`	`pow  (float_complex)`
`conj  (long_double_complex)`	`pow  (long_double_complex)`
`cos  (double_complex)`	`real (double_complex)`
`cos  (float_complex)`	`real (float_complex)`
`cos  (long_double_complex)`	`real (long_double_complex)`
`cosh (double_complex)`	`sin  (double_complex)`
`cosh (float_complex)`	`sin  (float_complex)`
`cosh (long_double_complex)`	`sin  (long_double_complex)`
`exp  (double_complex)`	`sinh (double_complex)`
`exp  (float_complex)`	`sinh (float_complex)`
`exp  (long_double_complex)`	`sinh (long_double_complex)`
`imag (double_complex)`	`sqrt (double_complex)`
`imag (float_complex)`	`sqrt (float_complex)`
`imag (long_double_complex)`	`sqrt (long_double_complex)`
`log  (double_complex)`	`_double_complex`
`log  (float_complex)`	`_float_complex`
`log  (long_double_complex)`	

3 The header `<complex>` defines a macro, three types, and numerous functions for representing and manipulating complex numbers.

4 The macro is:

```
        __STD_COMPLEX
```

5 whose definition is unspecified.

26.1.1 Complex numbers types [lib.complex.types]

26.1.1.1 Class `float_complex` [lib.float.complex]

```
    class float_complex {
    public:
        float_complex(float re_arg = 0, im_arg = 0);
        float_complex& operator+=(float_complex rhs);
        float_complex& operator-=(float_complex rhs);
        float_complex& operator*=(float_complex rhs);
        float_complex& operator/=(float_complex rhs);
    private:
//      float re, im;     exposition only
    };
```

1 The class `float_complex` describes an object that can store the Cartesian components, of type `float`, of a complex number.

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

— `float` *`re`*, the real component;

— `float` *`im`*, the imaginary component.

26.1.1.1.1 `float_complex` constructor [lib.float.complex.cons]

```
float_complex(float re_arg = 0, im_arg = 0);
```

1 Constructs an object of class `float_complex`, initializing *`re`* to *`re_arg`* and *`im`* to *`im_arg`*.

26.1.1.1.2 `operator+=` [lib.op+=.fc]

```
float_complex& operator+=(float_complex rhs);
```

1 Adds the complex value *`rhs`* to the complex value `*this` and stores the sum in `*this`. The function returns `*this`.

26.1.1.1.3 `operator-=` [lib.op-=.fc]

```
float_complex& operator-=(float_complex rhs);
```

1 Subtracts the complex value *`rhs`* from the complex value `*this` and stores the difference in `*this`. The function returns `*this`.

26.1.1.1.4 `operator*=` [lib.op*=.fc]

```
float_complex& operator*=(float_complex rhs);
```

1 Multiplies the complex value *`rhs`* by the complex value `*this` and stores the product in `*this`. The function returns `*this`.

26.1.1.1.5 `operator/=` [lib.op/=.fc]

```
float_complex& operator/=(float_complex rhs);
```

1 Divides the complex value *`rhs`* into the complex value `*this` and stores the quotient in `*this`. The function returns `*this`.

26.1.1.2 `float_complex` operations [lib.float.complex.ops]

26.1.1.2.1 `_float_complex` [lib.float.complex.dc]

```
float_complex _float_complex(const double_complex& rhs);
float_complex _float_complex(const long_double_complex& rhs);
```

1 Returns `float_complex((float)real(`*`rhs`*`), (float)imag(`*`rhs`*`))`.

26.1.1.2.2 `operator+` [lib.op+.fc.fc]

```
float_complex operator+(float_complex lhs);
```

1 Returns `float_complex(`*`lhs`*`)`.

```
float_complex operator+(float_complex lhs, float_complex rhs);
float_complex operator+(float_complex lhs, float rhs);
float_complex operator+(float lhs, float_complex rhs);
```

2 Returns `float_complex(`*`lhs`*`) +=` *`rhs`*.

26.1.1.2.3 `operator-` [lib.op-.fc.fc]

```
float_complex operator-(float_complex lhs);
```

1 Returns `float_complex(-real(`*lhs*`),-imag(`*lhs*`))`.

```
float_complex operator-(float_complex lhs, float_complex rhs);
float_complex operator-(float_complex lhs, float rhs);
float_complex operator-(float lhs, float_complex rhs);
```

2 Returns `float_complex(`*lhs*`) -=` *rhs*.

26.1.1.2.4 `operator*` [lib.op*.fc.fc]

```
float_complex operator*(float_complex lhs, float_complex rhs);
float_complex operator*(float_complex lhs, float rhs);
float_complex operator*(float lhs, float_complex rhs);
```

1 Returns `float_complex(`*lhs*`) *=` *rhs*.

26.1.1.2.5 `operator/` [lib.op/.fc.fc]

```
float_complex operator/(float_complex lhs, float_complex rhs);
float_complex operator/(float_complex lhs, float rhs);
float_complex operator/(float lhs, float_complex rhs);
```

1 Returns `float_complex(`*lhs*`) /=` *rhs*.

26.1.1.2.6 `operator==` [lib.op==.fc.fc]

```
bool operator==(float_complex lhs, float_complex >rhs);
```

1 Returns `real(`*lhs*`) == real(`*rhs*`) && imag(`*lhs*`) == imag(`*rhs*`)`.

```
bool operator==(float_complex lhs, float rhs);
```

2 Returns `real(`*lhs*`) ==` *rhs* `&& imag(`*lhs*`) == 0`.

```
bool operator==(float lhs, float_complex rhs);
```

3 Returns *lhs* `== real(`*rhs*`) && imag(`*rhs*`) == 0`.

26.1.1.2.7 `operator!=` [lib.op!=.fc.fc]

```
bool operator!=(float_complex lhs, float_complex rhs);
```

1 Returns `real(`*lhs*`) != real(`*rhs*`) || imag(`*lhs*`) != imag(`*rhs*`)`.

```
bool operator!=(float_complex lhs, float rhs);
```

2 Returns `real(`*lhs*`) !=` *rhs* `|| imag(`*lhs*`) != 0`.

```
bool operator!=(float lhs, float_complex rhs);
```

3 Returns *lhs* `!= real(`*rhs*`) || imag(`*rhs*`) != 0`.

26.1.1.2.8 `operator>>` [lib.ext.fc]

```
istream& operator>>(istream& is, float_complex& x);
```

1 Evaluates the expression:

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

2 where *ch* is an object of type `char` and *re* and *im* are objects of type `float`. If the result is nonzero, the function assigns `float_complex(`*re*`,` *im*`)` to *x*.

3 Returns *is*.

26.1.1.2.9 operator<< [lib.ins.fc]

```
    ostream& operator<<(ostream& os, float_complex x);
```

1 Returns *os* `<< '(' << real(`*x*`) << ',' << imag(`*x*`) << ')'`.

26.1.1.3 Class double_complex [lib.double.complex]

```
    class double_complex {
    public:
        double_complex(re_arg = 0, im_arg = 0);
        double_complex(const float_complex& rhs);
        double_complex& operator+=(double_complex rhs);
        double_complex& operator-=(double_complex rhs);
        double_complex& operator*=(double_complex rhs);
        double_complex& operator/=(double_complex rhs);
    private:
//    double re, im;      exposition only
    };
```

1 The class `double_complex` describes an object that can store the Cartesian components, of type `double`, of a complex number.

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

— `double` *re*, the real component;

— `double` *im*, the imaginary component.

26.1.1.3.1 double_complex constructors [lib.double.complex.cons]

```
    double_complex(double re_arg = 0, im_arg = 0);
```

1 Constructs an object of class `double_complex`, initializing *re* to *re_arg* and *im* to *im_arg*.

```
    double_complex(float_complex& rhs);
```

2 Constructs an object of class `double_complex`, initializing *re* to `(double)real(`*rhs*`)` and *im* to `(double)imag(`*rhs*`)`.

26.1.1.3.2 operator+= [lib.op+=.dc]

```
    double_complex& operator+=(double_complex rhs);
```

1 Adds the complex value *rhs* to the complex value `*this` and stores the sum in `*this`.

2 Returns `*this`.

26.1.1.3.3 operator-= [lib.op-=.dc]

```
double_complex& operator-=(double_complex rhs);
```

1 Subtracts the complex value *rhs* from the complex value *this and stores the difference in *this.

2 Returns *this.

26.1.1.3.4 operator*= [lib.op*=.dc]

```
double_complex& operator*=(double_complex rhs);
```

1 Multiplies the complex value *rhs* by the complex value *this and stores the product in *this.

2 Returns *this.

26.1.1.3.5 operator/= [lib.op/=.dc]

```
double_complex& operator/=(double_complex rhs);
```

1 Divides the complex value *rhs* into the complex value *this and stores the quotient in *this.

2 Returns *this.

26.1.1.4 double_complex operations [lib.double.complex.ops]

26.1.1.4.1 _double_complex [lib.double.complex.ldc]

```
double_complex _double_complex(const long_double_complex& rhs);
```

1 Returns double_complex((double)real(*rhs*), (double)imag(*rhs*)).

26.1.1.4.2 operator+ [lib.op+.dc.dc]

```
double_complex operator+(double_complex lhs);
```

1 Returns double_complex(*lhs*).

```
double_complex operator+(double_complex lhs, double_complex rhs);
double_complex operator+(double_complex lhs, double rhs);
double_complex operator+(double lhs, double_complex rhs);
```

2 Returns double_complex(*lhs*) += *rhs*.

26.1.1.4.3 operator- [lib.op-.dc.dc]

```
double_complex operator-(double_complex lhs);
```

1 Returns double_complex(-real(*lhs*),-imag(*lhs*)).

```
double_complex operator-(double_complex lhs, double_complex rhs);
double_complex operator-(double_complex lhs, double rhs);
double_complex operator-(double lhs, double_complex rhs);
```

2 Returns double_complex(*lhs*) -= *rhs*.

26.1.1.4.4 operator* [lib.op*.dc.dc]

```
double_complex operator*(double_complex lhs, double_complex rhs);
double_complex operator*(double_complex lhs, double rhs);
double_complex operator*(double lhs, double_complex rhs);
```

1 Returns `double_complex(lhs) *= rhs`.

26.1.1.4.5 `operator/` [lib.op/.dc.dc]

```
double_complex operator/(double_complex lhs, double_complex rhs);
double_complex operator/(double_complex lhs, double rhs);
double_complex operator/(double lhs, double_complex rhs);
```

1 Returns `double_complex(lhs) /= rhs`.

26.1.1.4.6 `operator==` [lib.op==.dc.dc]

```
bool operator==(double_complex lhs, double_complex rhs);
```

1 Returns `real(lhs) == real(rhs) && imag(lhs) == imag(rhs)`.

```
bool operator==(double_complex lhs, double rhs);
```

2 Returns `real(lhs) == rhs && imag(lhs) == 0`.

```
bool operator==(double lhs, double_complex rhs);
```

3 Returns `lhs == real(rhs) && imag(rhs) == 0`.

26.1.1.4.7 `operator!=` [lib.op!=.dc.dc]

```
bool operator!=(double_complex lhs, double_complex rhs);
```

1 Returns `real(lhs) != real(rhs) || imag(lhs) != imag(rhs)`.

```
bool operator!=(double_complex lhs, double rhs);
```

2 Returns `real(lhs) != rhs || imag(lhs) != 0`.

```
bool operator!=(double lhs, double_complex rhs);
```

3 Returns `lhs != real(rhs) || imag(rhs) != 0`.

26.1.1.4.8 `operator>>` [lib.ext.dc]

```
istream& operator>>(istream& is, double_complex& x);
```

1 Evaluates the expression:

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

2 where `ch` is an object of type `char` and `re` and `im` are objects of type `double`. If the result is nonzero, the function assigns `double_complex(re, im)` to `x`.

3 The function returns `is`.

26.1.1.4.9 `operator<<` [lib.ins.dc]

```
ostream& operator<<(ostream& os, double_complex x);
```

1 Returns `os << '(' << real(x) << ',' << imag(x) << ')'`.

26.1.1.5 Class `long_double_complex` [lib.long.double.complex]

```
    class long_double_complex {
    public:
        long_double_complex(re_arg = 0, im_arg = 0);
        long_double_complex(const float_complex& rhs);
        long_double_complex(const double_complex& rhs);
        long_double_complex& operator+=(long_double_complex rhs);
        long_double_complex& operator-=(long_double_complex rhs);
        long_double_complex& operator*=(long_double_complex rhs);
        long_double_complex& operator/=(long_double_complex rhs);
    private:
//      long double re, im;        exposition only
    };
```

1 The class `long_double_complex` describes an object that can store the Cartesian components, of type `long double`, of a complex number.

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

— `long double` *`re`*, the real component;

— `long double` *`im`*, the imaginary component.

26.1.1.5.1 `long_double_complex` constructors [lib.long.double.complex.cons]

```
    long_double_complex(long double re_arg = 0, im_arg = 0);
```

1 Constructs an object of class `long_double_complex`, initializing *`re`* to *`re_arg`* and *`im`* to *`im_arg`*.

```
    long_double_complex(float_complex& rhs);
```

2 Constructs an object of class `long_double_complex`, initializing *`re`* to `(long double)real(`*`rhs`*`)` and *`im`* to `(long double)imag(`*`rhs`*`)`.

```
    long_double_complex(double_complex& rhs);
```

3 Constructs an object of class `long_double_complex`, initializing *`re`* to `(long double)real(`*`rhs`*`)` and *`im`* to `(long double)imag(`*`rhs`*`)`.

26.1.1.5.2 `operator+=` [lib.op+=.ldc]

```
    long_double_complex& operator+=(long_double_complex rhs);
```

1 Adds the complex value *`rhs`* to the complex value `*this` and stores the sum in `*this`.

2 Returns `*this`.

26.1.1.5.3 `operator-=` [lib.op-=.ldc]

```
    long_double_complex& operator-=(long_double_complex rhs);
```

1 Subtracts the complex value *`rhs`* from the complex value `*this` and stores the difference in `*this`.

2 Returns `*this`.

26.1.1.5.4 `operator*=` [lib.op*=.ldc]

```
    long_double_complex& operator*=(long_double_complex rhs);
```

1 Multiplies the complex value *`rhs`* by the complex value `*this` and stores the product in `*this`.

2 Returns `*this`.

26.1.1.5.5 `operator/=` [lib.op/=.ldc]

```
long_double_complex& operator/=(long_double_complex rhs);
```

1 Divides the complex value *rhs* into the complex value `*this` and stores the quotient in `*this`.

2 Returns `*this`.

26.1.1.6 `long_double_complex` operations [lib.long.double.complex.ops]

26.1.1.6.1 `operator+` [lib.op+.ldc.ldc]

```
long_double_complex operator+(long_double_complex lhs);
```

1 Returns `long_double_complex(`*lhs*`)`.

```
long_double_complex operator+(long_double_complex lhs,
                              long_double_complex rhs);
long_double_complex operator+(long_double_complex lhs,
                              long double rhs);
long_double_complex operator+(long double lhs,
                              long_double_complex rhs);
```

2 Returns `long_double_complex(`*lhs*`) +=` *rhs*.

26.1.1.6.2 `operator-` [lib.op-.ldc.ldc]

```
long_double_complex operator-(long_double_complex lhs);
```

1 Returns `long_double_complex(-real(`*lhs*`),-imag(`*lhs*`))`.

```
long_double_complex operator-(long_double_complex lhs,
                              long_double_complex rhs);
long_double_complex operator-(long_double_complex lhs,
                              long double rhs);
long_double_complex operator-(long double lhs,
                              long_double_complex rhs);
```

2 Returns `long_double_complex(`*lhs*`) -=` *rhs*.

26.1.1.6.3 `operator*` [lib.op*.ldc.ldc]

```
long_double_complex operator*(long_double_complex lhs,
                              long_double_complex rhs);
```

1 Returns `long_double_complex(`*lhs*`) *=` *rhs*.

```
long_double_complex operator*(long_double_complex lhs,
                              long double rhs);
```

2 Returns `long_double_complex(`*lhs*`) *= long_double_complex(`*rhs*`)`.

```
long_double_complex operator*(long double lhs,
                              long_double_complex rhs);
```

3 Returns `long_double_complex(`*lhs*`) *=` *rhs*.

26.1.1.6.4 **operator/** [lib.op/.ldc.ldc]

```
long_double_complex operator/(long_double_complex lhs,
                              long_double_complex rhs);
long_double_complex operator/(long_double_complex lhs,
                              long double rhs);
long_double_complex operator/(long double lhs,
                              long_double_complex rhs);
```

1 Returns `long_double_complex(`*`lhs`*`) /=` *`rhs`*.

```
bool operator==(long_double_complex lhs, long_double_complex rhs);
```

2 Returns `real(`*`lhs`*`) == real(`*`rhs`*`) && imag(`*`lhs`*`) == imag(`*`rhs`*`)`.

```
bool operator==(long_double_complex lhs, long double rhs);
```

3 Returns `real(`*`lhs`*`) ==` *`rhs`* `&& imag(`*`lhs`*`) == 0`.

```
bool operator==(long double lhs, long_double_complex rhs);
```

4 Returns *`lhs`* `== real(`*`rhs`*`) && imag(`*`rhs`*`) == 0`.

26.1.1.6.5 **operator!=** [lib.op!=.ldc.ldc]

```
bool operator!=(long_double_complex lhs, long_double_complex rhs);
```

1 Returns `real(`*`lhs`*`) != real(`*`rhs`*`) || imag(`*`lhs`*`) != imag(`*`rhs`*`)`.

```
bool operator!=(long_double_complex lhs, long double rhs);
```

2 Returns `real(`*`lhs`*`) !=` *`rhs`* `|| imag(`*`lhs`*`) != 0`.

```
bool operator!=(long double lhs, long_double_complex rhs);
```

3 Returns *`lhs`* `!= real(`*`rhs`*`) || imag(`*`rhs`*`) != 0`.

26.1.1.6.6 **operator>>** [lib.ext.ldc]

```
istream& operator>>(istream& is, long_double_complex& x);
```

1 Evaluates the expression:

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

2 where *`ch`* is an object of type `char` and *`re`* and *`im`* are objects of type `long double`. If the result is nonzero, the function assigns `long_double_complex(`*`re`*`,` *`im`*`)` to *`x`*.

3 The function returns *`is`*.

26.1.1.6.7 **operator<<** [lib.ins.ldc]

```
ostream& operator<<(ostream& os, long_double_complex x);
```

1 Returns *`os`* `<< '(' << real(`*`x`*`) << ',' << imag(`*`x`*`) << ')'`.

26.1.2 Complex number operations [lib.complex.ops]

26.1.2.1 `abs` [lib.abs]

```
float       abs(float_complex x);
double      abs(double_complex x);
long double abs(long_double_complex x);
```

1 Returns the magnitude of *x*.

26.1.2.2 `arg` [lib.arg]

```
float       arg(float_complex x);
double      arg(double_complex x);
long double arg(long_double_complex x);
```

1 Returns the phase angle of *x*.

26.1.2.3 `conj` [lib.conj]

```
float_complex       conj(float_complex x);
double_complex      conj(double_complex x);
long_double_complex conj(long_double_complex x);
```

1 Returns the conjugate of *x*.

26.1.2.4 `cos` [lib.cos]

```
float_complex       cos(float_complex x);
double_complex      cos(double_complex x);
long_double_complex cos(long_double_complex x);
```

1 Returns the cosine of *x*.

26.1.2.5 `cosh` [lib.cosh]

```
float_complex       cosh(float_complex x);
double_complex      cosh(double_complex x);
long_double_complex cosh(long_double_complex x);
```

1 Returns the hyperbolic cosine of *x*.

26.1.2.6 `exp` [lib.exp]

```
float_complex       exp(float_complex x);
double_complex      exp(double_complex x);
long_double_complex exp(long_double_complex x);
```

1 Returns the exponential of *x*.

26.1.2.7 `imag` [lib.imag]

```
float       imag(float_complex x);
double      imag(double_complex x);
long double imag(long_double_complex x);
```

1 Returns the imaginary part of *x*.

26.1.2.8 `log` [lib.log]

```
float_complex       log(float_complex x);
double_complex      log(double_complex x);
long_double_complex log(long_double_complex x);
```

1 Returns the logarithm of *x*.

26.1.2.9 `norm` [lib.norm]

```
float       norm(float_complex x);
double      norm(double_complex x);
long double norm(long_double_complex x);
```

1 Returns the squared magnitude of *x*.

26.1.2.10 `polar` [lib.polar.ld.ld]

```
float_complex polar(float rho, float theta);
double_complex polar(double rho, double theta);
long_double_complex polar(long double rho, long double theta);
```

1 Returns the `complex` value corresponding to a complex number whose magnitude is *rho* and whose phase angle is *theta*.

26.1.2.11 `pow` [lib.pow]

```
float_complex       pow(float_complex x, float_complex y);
float_complex       pow(float_complex x, float_complex y);
float_complex       pow(float_complex x, float y);
float_complex       pow(float_complex x, int y);
float_complex       pow(float x, float_complex y);

double_complex      pow(double_complex x, double_complex y);
double_complex      pow(double_complex x, double y);
double_complex      pow(double_complex x, int y);
double_complex      pow(double x, double_complex y);

long_double_complex pow(long_double_complex x, long_double_complex y);
long_double_complex pow(long_double_complex x, long double y);
long_double_complex pow(long_double_complex x, int y);
long_double_complex pow(long double x, long_double_complex y);
```

1 Returns *x* raised to the power *y*.

26.1.2.12 `real` [lib.real]

```
float       real(float_complex x);
double      real(double_complex x);
long double real(long_double_complex x);
```

1 Returns the real part of *x*.

26.1.2.13 `sin` [lib.sin]

```
float_complex       sin(float_complex x);
double_complex      sin(double_complex x);
long_double_complex sin(long_double_complex x);
```

1 Returns the sine of *x*.

26.1.2.14 `sinh` [lib.sinh]

```
float_complex       sinh(float_complex x);
double_complex      sinh(double_complex x);
long_double_complex sinh(long_double_complex x);
```

1 Returns the hyperbolic sine of *x*.

26.1.2.15 `sqrt` [lib.sqrt]

```
float_complex       sqrt(float_complex x);
double_complex      sqrt(double_complex x);
long_double_complex sqrt(long_double_complex x);
```

1 Returns the square root of *x*.

26.2 Numeric arrays [lib.numarray]

1 Headers:

— `<valarray>`

2 Table 89:

Table 89—Header `<valarray>` synopsis

Type	Name(s)	
Template classes:		
`mask_array`	`slice_array`	`valarray`
Template operators:		
`operator!= (valarray) [3]`	`operator<< (valarray) [3]`	
`operator% (valarray) [3]`	`operator<<=(valarray) [2]`	
`operator%= (valarray) [2]`	`operator<= (valarray) [3]`	
`operator& (valarray) [3]`	`operator== (valarray) [3]`	
`operator&& (valarray) [3]`	`operator> (valarray) [3]`	
`operator&= (valarray) [2]`	`operator>= (valarray) [3]`	
`operator* (valarray) [3]`	`operator>> (valarray) [3]`	
`operator*= (valarray) [2]`	`operator>>=(valarray) [2]`	
`operator+ (valarray) [3]`	`operator^ (valarray) [3]`	
`operator+= (valarray) [2]`	`operator^= (valarray) [2]`	
`operator- (valarray) [3]`	`operator| (valarray) [3]`	
`operator-= (valarray) [2]`	`operator|= (valarray) [2]`	
`operator/ (valarray) [3]`	`operator|| (valarray) [3]`	
`operator/= (valarray) [2]`		
`operator< (valarray) [3]`		
Template functions:		
`abs (valarray)`	`cosh (valarray)`	`sinh (valarray)`
`acos (valarray)`	`exp (valarray)`	`sqrt (valarray)`
`asin (valarray)`	`log (valarray)`	`tan (valarray)`
`atan (valarray)`	`log10(valarray)`	`tanh (valarray)`
`atan2(valarray) [3]`	`pow (valarray) [3]`	
`cos (valarray)`	`sin (valarray)`	
Classes:	`gslice`	`slice`

3 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.[115)]

4 The `valarray` array classes are defined to be free of certain forms of aliasing, thus allowing operations on these classes to be optimized.

5 These library functions are permitted to throw an `bad_alloc` exception if there are not sufficient resources available to carry out the operation. Note that the exception is not mandated.

Box 101

The descriptions of `valarray` and the associated classes which follow lack any discussion of possible exceptions.

6 The templates and classes defined in `<valarray>` have the following public interfaces:

115) If any of the names `valarray, slice_array, gslice_array, mask_array, indirect_array, slice` or `gslice` are introduced into a translation unit by any means other than inclusion of the `<valarray>` header file, the resulting behavior is undefined.

```
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
```

26.2.1 Template class `valarray<T>` [lib.template.valarray]

```
template<class T> class valarray {
public:
    inline valarray();
    inline valarray(enum Uninitialized, size_t);
    valarray(const T&, size_t);
    valarray(const T*, size_t);
    valarray(const valarray&);
    valarray(const slice_array<T>&);
    valarray(const gslice_array<T>&);
    valarray(const mask_array<T>&);
    valarray(const indirect_array<T>&);
    inline ~valarray();

    valarray& operator=(const valarray&);
    valarray& operator=(const slice_array<T>&);
    valarray& operator=(const gslice_array<T>&);
    valarray& operator=(const mask_array<T>&);
    valarray& operator=(const indirect_array<T>&);

    inline size_t length() const;
    inline operator T*();
    inline operator const T*() const;

    inline const T    operator[](size_t);
    inline T&         operator[](size_t);
    const valarray    operator[](slice) const;
    slice_array<T>    operator[](slice);
    const valarray    operator[](const gslice&) const;
    gslice_array<T>   operator[](const gslice&);
    const valarray    operator[](const valarray<bool>&) const;
    mask_array<T>     operator[](const valarray<bool>&);
    const valarray    operator[](const valarray<int>&) const;
    indirect_array<T> operator[](const valarray<int>&);

    const valarray operator+() const;
    const valarray operator-() const;
    const valarray operator~() const;
    const valarray operator!() const;

    valarray<T>& operator*=(const T&);
    valarray<T>& operator/=(const T&);
    valarray<T>& operator%=(const T&);
    valarray<T>& operator+=(const T&);
    valarray<T>& operator-=(const T&);
    valarray<T>& operator^=(const T&);
    valarray<T>& operator&=(const T&);
    valarray<T>& operator|=(const T&);
    valarray<T>& operator<<=(const T&);
    valarray<T>& operator>>=(const T&);
```

Box 102

The friend specifiers are over-specification. I have left them in, pending discussion, because they are a concise summary of non-member operators applicable to this class.

```
friend const valarray<T> operator* (const valarray<T>&, const T&);
friend const valarray<T> operator* (const T&, const valarray<T>&);
friend const valarray<T> operator/ (const valarray<T>&, const T&);
friend const valarray<T> operator/ (const T&, const valarray<T>&);
friend const valarray<T> operator% (const valarray<T>&, const T&);
friend const valarray<T> operator% (const T&, const valarray<T>&);
friend const valarray<T> operator+ (const valarray<T>&, const T&);
friend const valarray<T> operator+ (const T&, const valarray<T>&);
friend const valarray<T> operator- (const valarray<T>&, const T&);
friend const valarray<T> operator- (const T&, const valarray<T>&);
friend const valarray<T> operator^ (const valarray<T>&, const T&);
friend const valarray<T> operator^ (const T&, const valarray<T>&);
friend const valarray<T> operator& (const valarray<T>&, const T&);
friend const valarray<T> operator& (const T&, const valarray<T>&);
friend const valarray<T> operator| (const valarray<T>&, const T&);
friend const valarray<T> operator| (const T&, const valarray<T>&);
friend const valarray<T> operator<<(const valarray<T>&, const T&);
friend const valarray<T> operator<<(const T&, const valarray<T>&);
friend const valarray<T> operator>>(const valarray<T>&, const T&);
friend const valarray<T> operator>>(const T&, const valarray<T>&);
friend const valarray<T> operator&&(const valarray<T>&, const T&);
friend const valarray<T> operator&&(const T&, const valarray<T>&);
friend const valarray<T> operator||(const valarray<T>&, const T&);
friend const valarray<T> operator||(const T&, const valarray<T>&);

valarray<T>& operator*= (const valarray<T>& ab);
valarray<T>& operator/= (const valarray<T>& ab);
valarray<T>& operator%= (const valarray<T>& ab);
valarray<T>& operator+= (const valarray<T>& ab);
valarray<T>& operator-= (const valarray<T>& ab);
valarray<T>& operator^= (const valarray<T>&);
valarray<T>& operator|= (const valarray<T>&);
valarray<T>& operator&= (const valarray<T>&);
valarray<T>& operator<<=(const valarray<T>&);
valarray<T>& operator>>=(const valarray<T>&);

friend const valarray<T> operator* (const valarray<T>&, const valarray<T>&);
friend const valarray<T> operator/ (const valarray<T>&, const valarray<T>&);
friend const valarray<T> operator% (const valarray<T>&, const valarray<T>&);
friend const valarray<T> operator+ (const valarray<T>&, const valarray<T>&);
friend const valarray<T> operator- (const valarray<T>&, const valarray<T>&);
friend const valarray<T> operator^ (const valarray<T>&, const valarray<T>&);
friend const valarray<T> operator| (const valarray<T>&, const valarray<T>&);
friend const valarray<T> operator& (const valarray<T>&, const valarray<T>&);
friend const valarray<T> operator<<(const valarray<T>&, const valarray<T>&);
friend const valarray<T> operator>>(const valarray<T>&, const valarray<T>&);
friend const valarray<T> operator&&(const valarray<T>&, const valarray<T>&);
friend const valarray<T> operator||(const valarray<T>&, const valarray<T>&);
```

```
        friend const valarray<bool> operator==(const valarray<T>&, const T&);
        friend const valarray<bool> operator==(const T&, const valarray<T>&);
        friend const valarray<bool> operator==(const valarray<T>&, const valarray<T>&);
        friend const valarray<bool> operator!=(const valarray<T>&, const T&);
        friend const valarray<bool> operator!=(const T&, const valarray<T>&);
        friend const valarray<bool> operator!=(const valarray<T>&, const valarray<T>&);
        friend const valarray<bool> operator< (const T&, const valarray<T>&);
        friend const valarray<bool> operator< (const T&, const valarray<T>&);
        friend const valarray<bool> operator< (const valarray<T>&, const valarray<T>&);
        friend const valarray<bool> operator> (const valarray<T>&, const T&);
        friend const valarray<bool> operator> (const T&, const valarray<T>&);
        friend const valarray<bool> operator> (const valarray<T>&, const valarray<T>&);
        friend const valarray<bool> operator<=(const valarray<T>&, const T&);
        friend const valarray<bool> operator<=(const T&, const valarray<T>&);
        friend const valarray<bool> operator<=(const valarray<T>&, const valarray<T>&);
        friend const valarray<bool> operator>=(const valarray<T>&, const T&);
        friend const valarray<bool> operator>=(const valarray<T>&, const T&);
        friend const valarray<bool> operator>=(const T&, const valarray<T>&);
        friend const valarray<bool> operator>=(const valarray<T>&, const valarray<T>&);

        const T sum() const;
        void fill(const T&);
        const T min() const;
        const T max() const;

        friend const valarray<T> abs  (const valarray<T>&);
        friend const valarray<T> acos (const valarray<T>&);
        friend const valarray<T> asin (const valarray<T>&);
        friend const valarray<T> atan (const valarray<T>&);
        friend const valarray<T> atan2(const valarray<T>&, const valarray<T>&);
        friend const valarray<T> atan2(const valarray<T>&, const T&);
        friend const valarray<T> atan2(const T&, const valarray<T>&);
        friend const valarray<T> cos  (const valarray<T>&);
        friend const valarray<T> cosh (const valarray<T>&);
        friend const valarray<T> exp  (const valarray<T>&);
        friend const valarray<T> log  (const valarray<T>&);
        friend const valarray<T> log10(const valarray<T>&);
        friend const valarray<T> pow  (const valarray<T>&, const valarray<T>&);
        friend const valarray<T> pow  (const valarray<T>&, const T&);
        friend const valarray<T> pow  (const T&, const valarray<T>&);
        friend const valarray<T> sin  (const valarray<T>&);
        friend const valarray<T> sinh (const valarray<T>&);
        friend const valarray<T> sqrt (const valarray<T>&);
        friend const valarray<T> tan  (const valarray<T>&);
        friend const valarray<T> tanh  (const valarray<T>&);

        const valarray<T> shift(int) const;
        const valarray<T> apply(T func(T)) const;
        const valarray<T> apply(T func(const T&)) const;
        void free();
    private:
    // implementation dependent
    };
```

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.[116)]

[116)] 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

2 An implementation is permitted to qualify any of the functions declared in `<valarray>` as `inline`.

3 A specialization of `valarray` for a type *T* has well-defined behavior if and only if the type *T* satisfies the following requirements:[117)]

— *T* is not an abstract class (it has no pure virtual member functions);

— *T* is not a reference type;

— *T* is not cv-qualified;

— If *T* is a class, it has a public default constructor;

— If *T* is a class, it has a public copy constructor with the signature *`T::T(const T&)`*

— If *T* is a class, it has a public destructor;

— If *T* is a class, it has a public assignment operator whose signature is either

```
T& T::operator=(const T&)
```

or

```
T& T::operator=(T)
```

— If *T* is a class, its assignment operator, copy and default constructors, and destructor must correspond to each other in the following sense: Initialization of raw storage using the default constructor, followed by assignment, is semantically equivalent to initialization of raw storage using the copy constructor. Destruction of an object, followed by initialization of its raw storage using the copy constructor, is semantically equivalent to assignment to the original object. This rule states that there must not be any subtle differences in the semantics of initialization versus assignment. This gives an implementation considerable flexibility in how arrays are initialized.

For example, an implementation is allowed to initialize a `valarray` by allocating storage using the `new` operator (which implies a call to the default constructor for each element) and then assigning each element its value. Or the implementation can allocate raw storage and use the copy constructor to initialize each element. If the distinction between initialization and assignment is important for a class, or if it fails to satisfy any of the other conditions listed above, the programmer should use `dynarray` instead of `valarray` for that class;

— If *T* is a class, it does not overload unary `operator&`.

4 In addition, many member and ***friend*** functions of `valarray<T>` can be successfully instantiated and will exhibit well-defined behavior if and only if *T* satisfies additional requirements specified for each such member or ***friend*** function.

5 For example, it is legitimate to instantiate `valarray<complex>`, but operator > will not be successfully instantiated for `valarray<complex>` operands, since `complex` does not have any ordering operators.

26.2.1.1 `valarray` constructors [lib.valarray.cons]

```
valarray::valarray();
```

1 Constructs an object of class `valarray<T>`, [118)] 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.

building block for designing such classes.

117) In other words, `valarray<T>` should only be instantiated for value types. These include built-in arithmetic types, pointers, the library class `complex`, and instantiations of `valarray` for value types.

118) For convenience, such objects are referred to as ‘‘arrays’’ throughout the remainder of subclause 26.2.

```
valarray::valarray(enum Uninitialized, size_t);
```

2 The array created by this constructor has a length equal to the value of the second argument. The first argument is not used. The elements of the array are constructed using the default constructor for the instantiating type *T*. The extra argument is needed to prevent this constructor from being used by the compiler to silently convert integers to `valarray` objects.

```
valarray::valarray(const T&, size_t);
```

3 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::valarray(const T*, size_t);
```

4 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::valarray(const valarray&);
```

5 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::valarray(const slice_array<T>&);
valarray::valarray(const gslice_array<T>&);
valarray::valarray(const mask_array<T>&);
valarray::valarray(const indirect_array<T>&);
```

6 These conversion constructors convert one of the four reference templates to a `valarray`.

26.2.1.2 `valarray` destructor [lib.valarray.des]

```
valarray::~valarray();
```

26.2.1.3 `valarray` assignment [lib.valarray.op=]

```
valarray& valarray::operator=(const valarray&);
```

1 The assignment operator modifies the length of the `*this` array to be equal to that of the argument array. Each element of the `*this` array is then assigned the value of the corresponding element of the argument array. Assignment is the usual way to change the length of an array after initialization. Assignment results in a distinct array rather than an alias.

```
valarray& valarray::operator=(const slice_array<T>&);
valarray& valarray::operator=(const gslice_array<T>&);
valarray& valarray::operator=(const mask_array<T>&);
valarray& valarray::operator=(const indirect_array<T>&);
```

2 These operators allow the results of a generalized subscripting operation to be assigned directly to a `valarray`.

26.2.1.4 `valarray` length access [lib.valarray::length]

```
size_t valarray::length() const;
```

1 This function returns the number of elements in the `this` array.

26.2.1.5 `valarray` pointer conversion [lib.valarray.ptr]

```
valarray::operator T*();
valarray::operator const T*() const;
```

1 A non-constant array may be converted to a pointer to the instantiating type. A constant array may be converted to a pointer to the instantiating type, qualified by `const`.

2 It is guaranteed that

```
&a[0] == (T*)a
```

for any non-constant `valarray<T> a`. The pointer returned for a non-constant array (whether or not it points to a type qualified by `const`) is valid for the same duration as a reference returned by the `size_t` subscript operator. The pointer returned for a constant array is valid for the lifetime of the array.[119)]

26.2.1.6 `valarray` element access [lib.valarray.access]

```
const T operator[](size_t) const;
T& operator[](size_t);
```

1 When applied to a constant array, the subscript operator returns the value of the corresponding element of the array. When applied to a non-constant array, the subscript operator returns a reference to the corresponding element of the array.

2 Thus, the expression

```
(a[i] = q, a[i]) == q
```

evaluates as true for any non-constant `valarray<T> a`, any `T q`, and for any `size_t i` such that the 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.[120)]

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& 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.

119) This form of access is essential for reusability and cross-language programming.
120) Compilers may take advantage of inlining, constant propagation, loop fusion, tracking of pointers obtained from `operator new`, and other techniques to generate efficient `valarray`s.

26.2.1.7 `valarray` subset operations [lib.valarray.sub]

```
const valarray    operator[](slice) const;
slice_array<T>    operator[](slice);
const valarray    operator[](const gslice&) const;
gslice_array<T>   operator[](const gslice&);
const valarray    operator[](const valarray<bool>&) const;
mask_array<T>     operator[](const valarray<bool>&);
const valarray    operator[](const valarray<int>&) const;
indirect_array<T> operator[](const valarray<int>&);
```

1 Each of these operations returns a subset of the `this` array. The `const-` qualified versions return this subset as a new `valarray`. The non-`const` versions return a class template object which has reference semantics to the original array.

26.2.1.8 `valarray` unary operators [lib.valarray.unary]

```
const valarray valarray::operator+() const;
const valarray valarray::operator-() const;
const valarray valarray::operator~() const;
const valarray valarray::operator!() const;
```

1 Each of these operators may only be instantiated for a type *T* to which the indicated operator can be applied and for which the indicated operator returns a value which is of type &*T* or which may be unambiguously converted to type *T*.

2 Each of these operators returns an array whose length is equal to the length of the `this` array. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding element of the `this` array.

26.2.1.9 `valarray` binary operators with scalars [lib.valarray.binary.scal]

```
const valarray operator* (const valarray&, const T&);
const valarray operator/ (const valarray&, const T&);
const valarray operator% (const valarray&, const T&);
const valarray operator+ (const valarray&, const T&);
const valarray operator- (const valarray&, const T&);
const valarray operator^ (const valarray&, const T&);
const valarray operator& (const valarray&, const T&);
const valarray operator| (const valarray&, const T&);
const valarray operator<<(const valarray&, const T&);
const valarray operator>>(const valarray&, const T&);
const valarray operator&&(const valarray&, const T&);
const valarray operator||(const valarray&, const T&);

const valarray operator* (const T&, const valarray&);
const valarray operator/ (const T&, const valarray&);
const valarray operator% (const T&, const valarray&);
const valarray operator+ (const T&, const valarray&);
const valarray operator- (const T&, const valarray&);
const valarray operator^ (const T&, const valarray&);
const valarray operator& (const T&, const valarray&);
const valarray operator| (const T&, const valarray&);
const valarray operator<<(const T&, const valarray&);
const valarray operator>>(const T&, const valarray&);
const valarray operator&&(const T&, const valarray&);
const valarray operator||(const T&, const valarray&);
```

1 Each of these operators may only be instantiated for a type *T* to which the indicated operator can be applied and for which the indicated operator returns a value which is of type *T* or which can be unambiguously converted to type *T*.

2 Each of these operators returns an array whose length is equal to the length of the array argument. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding element of the array argument and the scalar argument.

26.2.1.10 `valarray` computed assigns with scalars [lib.valarray.cassign.scal]

```
valarray& valarray::operator*= (const T&);
valarray& valarray::operator/= (const T&);
valarray& valarray::operator%= (const T&);
valarray& valarray::operator+= (const T&);
valarray& valarray::operator-= (const T&);
valarray& valarray::operator^= (const T&);
valarray& valarray::operator&= (const T&);
valarray& valarray::operator|= (const T&);
valarray& valarray::operator<<=(const T&);
valarray& valarray::operator>>=(const T&);
```

1 Each of these operators may only be instantiated for a type *T* to which the indicated operator can be applied.

2 Each of these operators applies the indicated operation to each element of the `this` array and the scalar argument.

3 The `this` array is then returned by reference.

4 The appearance of an array on the left hand side of a computed assignment does *not* invalidate references or pointers to the elements of the array.

26.2.1.11 `valarray` binary operations with other arrays [lib.valarray.bin.array]

```
const valarray operator* (const valarray&, const valarray&);
const valarray operator/ (const valarray&, const valarray&);
const valarray operator% (const valarray&, const valarray&);
const valarray operator+ (const valarray&, const valarray&);
const valarray operator- (const valarray&, const valarray&);
const valarray operator^ (const valarray&, const valarray&);
const valarray operator& (const valarray&, const valarray&);
const valarray operator| (const valarray&, const valarray&);
const valarray operator<<(const valarray&, const valarray&);
const valarray operator>>(const valarray&, const valarray&);
const valarray operator&&(const valarray&, const valarray&);
const valarray operator||(const valarray&, const valarray&);
```

1 Each of these operators may only be instantiated for a type *T* to which the indicated operator can be applied and for which the indicated operator returns a value which is of type *T* or which can be unambiguously converted to type *T*.

2 Each of these operators returns an array whose length is equal to the lengths of the argument arrays. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding elements of the argument arrays.

3 If the argument arrays do not have the same length, the behavior is undefined.

26.2.1.12 `valarray` computed assignments with other arrays [lib.valarray.assign.array]

```
valarray& valarray::operator*= (const valarray&);
valarray& valarray::operator/= (const valarray&);
valarray& valarray::operator%= (const valarray&);
valarray& valarray::operator+= (const valarray&);
valarray& valarray::operator-= (const valarray&);
valarray& valarray::operator^= (const valarray&);
valarray& valarray::operator&= (const valarray&);
valarray& valarray::operator|= (const valarray&);
valarray& valarray::operator<<=(const valarray&);
valarray& valarray::operator>>=(const valarray&);
```

1 Each of these operators may only be instantiated for a type *T* to which the indicated operator can be applied. Each of these operators performs the indicated operation on each of its elements and the corresponding element of the argument array.

2 The `this` array is then returned by reference.

3 If the `this` array and the argument array do not have the same length, the behavior is undefined. The appearance of an array on the left hand side of a computed assignment does *not* invalidate references or pointers.

26.2.1.13 `valarray` comparison operators with scalars [lib.valarray.comp.scal]

```
const valarray<bool> operator==(const valarray&, const T&);
const valarray<bool> operator!=(const valarray&, const T&);
const valarray<bool> operator< (const valarray&, const T&);
const valarray<bool> operator> (const valarray&, const T&);
const valarray<bool> operator<=(const valarray&, const T&);
const valarray<bool> operator>=(const valarray&, const T&);
const valarray<bool> operator==(const T&, const valarray&);
const valarray<bool> operator!=(const T&, const valarray&);
const valarray<bool> operator< (const T&, const valarray&);
const valarray<bool> operator> (const T&, const valarray&);
const valarray<bool> operator<=(const T&, const valarray&);
const valarray<bool> operator>=(const T&, const valarray&);
```

1 Each of these operators may only be instantiated for a type *T* to which the indicated operator can be applied and for which the indicated operator returns a value which is of type *bool* or which can be unambiguously converted to type *bool*.

2 Each of these operators returns a *bool* array whose length is equal to the length of the array argument. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding element of the `this` array and the scalar argument.

26.2.1.14 `valarray` comparison operators with other arrays [lib.valarray.comp.array]

```
const valarray<bool> operator==(const valarray&, const valarray&);
const valarray<bool> operator!=(const valarray&, const valarray&);
const valarray<bool> operator< (const valarray&, const valarray&);
const valarray<bool> operator> (const valarray&, const valarray&);
const valarray<bool> operator<=(const valarray&, const valarray&);
const valarray<bool> operator>=(const valarray&, const valarray&);
```

1 Each of these operators may only be instantiated for a type *T* to which the indicated operator can be applied and for which the indicated operator returns a value which is of type *bool* or which can be unambiguously converted to type *bool*.

2 Each of these operators returns a *bool* array whose length is equal to the length of the array arguments. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding elements of the argument arrays.

3 If the two array arguments do not have the same length, the behavior is undefined.

26.2.1.15 `valarray` sum function [lib.valarray.sum]

```
const T valarray::sum() const;
```

This function may only be instantiated for a type *T* to which `operator+=` can be applied. This function returns the sum of all the elements of the array.

1 If the array has length 0, the behavior is undefined. If the array has length 1, `sum` returns the value of element 0. Otherwise, the returned value is calculated by applying `operator+=` to a copy of an element of the array and all other elements of the array in an unspecified order.

26.2.1.16 `valarray` fill function [lib.valarray.fill]

```
void valarray::fill(const T&);
```

This function assigns the value of the argument to all the elements of the `this` array. The length of the array is not changed, nor are any pointers or references to the elements of the array invalidated.

26.2.1.17 `valarray` transcendentals [lib.valarray.transcend]

```
const valarray abs  (const valarray&);
const valarray acos (const valarray&);
const valarray asin (const valarray&);
const valarray atan (const valarray&);
const valarray atan2(const valarray&, const valarray&);
const valarray atan2(const valarray&, const T&);
const valarray atan2(const T&, const valarray&);
const valarray cos  (const valarray&);
const valarray cosh (const valarray&);
const valarray exp  (const valarray&);
const valarray log  (const valarray&);
const valarray log10(const valarray&);
const valarray pow  (const valarray&, const valarray&);
const valarray pow  (const valarray&, const T&);
const valarray pow  (const T&, const valarray&);
const valarray sin  (const valarray&);
const valarray sinh (const valarray&);
const valarray sqrt (const valarray&);
const valarray tan  (const valarray&);
const valarray tanh (const valarray&);
```

1 Each of these functions may only be instantiated for a type *T* to which a unique function with the indicated name can be applied. This function must return a value which is of type *T* or which can be unambiguously converted to type *T*.

26.2.1.18 `valarray` min and max functions [lib.valarray.minmax]

```
const T min(const valarray&);
const T max(const valarray&);
```

1 These functions may only be instantiated for a type *T* to which `operator>` and `operator<` may be applied and for which `operator>` and `operator<` return a value which is of type *bool* or which can be unambiguously converted to type *bool*.

2 These functions return the minimum or maximum value found in the argument array.

3 The value returned for an array of length 0 is undefined. For an array of length 1, the value of element 0 is returned. For all other array lengths, the determination is made using `operator>` and `operator<`, in a manner analogous to the application of `operator+=` for the `sum` function.

26.2.1.19 `valarray` shift function [lib.valarray.shift]

```
const valarray valarray::shift(int) const;
```

1 This function returns an array whose length is identical to the `this` array, but whose element values are shifted the number of places indicated by the argument.

2 For example, if the argument has the value 2, the first two elements of the result will be constructed using the default constructor; the third element of the result will be assigned the value of the first element of the argument; etc.

Box 103

Should a cshift (circular shift) function also be defined? This is a common operation in Fortran.

26.2.1.20 `valarray` mapping functions [lib.valarray.map]

```
const valarray valarray::apply(T func(T)) const;
const valarray valarray::apply(T func(const T&)) const;
```

1 These functions return an array whose length is equal to the `this` array. Each element of the returned array is assigned the value returned by applying the argument function to the corresponding element of the `this` array.

26.2.1.21 `valarray` free function [lib.valarray.free]

```
void valarray::free();
```

1 This function sets the length of an array to zero.[121)]

26.2.2 Class `slice` [lib.class.slice]

```
class slice {
public:
    slice();
    slice(int, int, int);
    int start() const;
    int length() const;
    int stride() const;
private:
//      implementation defined
};
```

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.[122)]

26.2.2.1 `slice` constructors [lib.cons.slice]

```
slice::slice();
slice::slice(int start, int length, int stride);
slice::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.

121) An implementation may reclaim the storage used by the array when this function is called.
122) C++ programs may instantiate this class.

2 For example,

```
slice(3, 8, 2)
```

constructs a slice which selects elements 3, 5, 7, ... 17 from an array.

26.2.2.2 `slice access functions` [lib.slice.access]

```
int slice::start() const;
int slice::length() const;
int slice::stride() const;
```

1 These functions return the start, length, or stride specified by a `slice` object.

26.2.3 Template class `slice_array` [lib.template.slice.array]

```
template <class T> class slice_array {
public:
    void operator=(const valarray<T>&) const;
    void operator*=(const valarray<T>&) const;
    void operator/=(const valarray<T>&) const;
    void operator%=(const valarray<T>&) const;
    void operator+=(const valarray<T>&) const;
    void operator-=(const valarray<T>&) const;
    void operator^=(const valarray<T>&) const;
    void operator&=(const valarray<T>&) const;
     void operator|=(const valarray<T>&) const;
    void operator<<=(const valarray<T>&) const;
    void operator>>=(const valarray<T>&) const;

    void fill(const T&);
private:
    slice_array();
    slice_array(const slice_array&);
    slice_array& operator=(const slice_array&);
//    remainder implementation defined
};
```

1 The `slice_array` template is a helper template used by the `slice` subscript operator

```
slice_array<T> valarray<T>::operator[](slice);
```

It has reference semantics to a subset of an array specified by a `slice` object.

2 For example, the expression

```
a[slice(1, 5, 3)] = b;
```

has the effect of assigning the elements of `b` to a slice of the elements in `a`. For the slice shown, the elements selected from `a` be 1, 4, ..., 13.

3 Note that programmers may not instantiate `slice_array`, since all its constructors are private. It is intended purely as a helper class and should be transparent to the user.

26.2.3.1 `slice_array` constructors [lib.cons.slice.arr]

```
slice_array::slice_array();
slice_array::slice_array(const slice_array&);
```

1 Note that the `slice_array` template has no public constructors. These constructors are declared to be private. These constructors need not be defined.

26.2.3.2 slice_array assignment [lib.slice.arr.assign]

```
void operator=(const valarray<T>&) const;
slice_array& operator=(const slice_array&);
```

1 The second of these two assignment operators is declared private and need not be defined. The first has reference semantics, assigning the values of the argument array elements to selected elements of the valarray<T> object to which the slice_array object refers.

26.2.3.3 slice_array computed assignment [lib.slice.arr.comp.assign]

```
void slice_array::operator*= (const valarray<T>&) const;
void slice_array::operator/= (const valarray<T>&) const;
void slice_array::operator%= (const valarray<T>&) const;
void slice_array::operator+= (const valarray<T>&) const;
void slice_array::operator-= (const valarray<T>&) const;
void slice_array::operator^= (const valarray<T>&) const;
void slice_array::operator&= (const valarray<T>&) const;
void slice_array::operator|= (const valarray<T>&) const;
void slice_array::operator<<=(const valarray<T>&) const;
void slice_array::operator>>=(const valarray<T>&) const;
```

1 These computed assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the valarray<T> object to which the slice_array object refers.

26.2.3.4 slice_array fill function [lib.slice.arr.fill]

```
void fill(const T&);
```

1 This function has reference semantics, assigning the value of its argument to the elements of the valarray<T> object to which the slice_array object refers.

26.2.4 The gslice class [lib.class.gslice]

```
class gslice {
public:
    gslice();
    gslice(int s, const valarray<int>& l, const valarray<int>& d);
    int start() const;
    valarray<int> length() const;
    valarray<int> stride() const;
private:
//  implementation defined
};
```

1 This class represents a generalized slice out of an array. A gslice is defined by a starting offset (s), a set of lengths (l_j), and a set of strides (d_j). The number of lengths must equal the number of strides.

2 A gslice represents a mapping from a set of indices (i_j), equal in number to the number of strides, to a single index k. It is useful for building multidimensional array classes using the valarray template, which is one-dimensional. The set of one-dimensional index values specified by a gslice are

$$k = s + sum13j(i_j d_j)$$

where the multidimensional indices i_j range in value from 0 to l_{ij}-1.

3 For example, the gslice specification

```
start  = 3
length = {2, 4, 3}
stride = {19, 4, 1}
```

yields the sequence of one-dimensional indices

```
k = 3 + (0,1) x 19 = (0,1,2,3) x 4 + (0,1,2) x 1
```

which are ordered as shown in the following table:

```
(i0, i1, i2, k) =
        (0, 0, 0, 3),
        (0, 0, 1, 4),
        (0, 0, 2, 5),
        (0, 1, 0, 7),
        (0, 1, 1, 8),
        (0, 1, 2, 9),
        (0, 2, 0, 11),
        (0, 2, 1, 12),
        (0, 2, 2, 13),
        (0, 3, 0, 15),
        (0, 3, 1, 16),
        (0, 3, 2, 17),
        (1, 0, 0, 22),
        (1, 0, 1, 23),
        ...
        (1, 3, 2, 36)
```

That is, the highest-ordered index turns fastest.

4 It is possible to have degenerate generalized slices in which an address is repeated.

5 For example, if the stride parameters in the previous example are changed to {1, 1, 1}, the first few elements of the resulting sequence of indices will be

```
        (0, 0, 0, 3),
        (0, 0, 1, 4),
        (0, 0, 2, 5),
        (0, 1, 0, 4),
        (0, 1, 1, 5),
        (0, 1, 2, 6),
        ...
```

6 If a degenerate slice is used as the argument to the non-`const` version of `operator[](const gslice&)`, the resulting behavior is undefined.

26.2.4.1 `gslice constructors` [lib.gslice.cons]

```
gslice::gslice();
gslice::gslice(int start, const valarray<int>& lengths, const valarray<int>& strides);
gslice::gslice(const gslice&);
```

1 The default constructor creates a `gslice` which specifies no elements. The constructor with arguments builds a `gslice` based on a specification of start, lengths, and strides, as explained in the previous section.

26.2.4.2 `gslice access functions` [lib.gslice.access]

```
int start() const;
valarray<int> length() const;
valarray<int> stride() const;
```

These access functions return the representation of the start, lengths, or strides specified for the `gslice`.

26.2.5 Template class `gslice_array` [lib.template.gslice.array]

```
template <class T> class gslice_array {
public:
    void operator=(const valarray<T>&) const;
    void operator*=(const valarray<T>&) const;
    void operator/=(const valarray<T>&) const;
    void operator%=(const valarray<T>&) const;
    void operator+=(const valarray<T>&) const;
    void operator-=(const valarray<T>&) const;
    void operator^=(const valarray<T>&) const;
    void operator&=(const valarray<T>&) const;
    void operator|=(const valarray<T>&) const;
    void operator<<=(const valarray<T>&) const;
    void operator>>=(const valarray<T>&) const;

    void fill(const T&);
private:
    gslice_array();
    gslice_array(const gslice_array&);
    gslice_array& operator=(const gslice_array&);
//  remainder implementation defined
};
```

1 This template is a helper template used by the `slice` subscript operator

```
gslice_array<T> valarray<T>::operator[](const gslice&);
```

It has reference semantics to a subset of an array specified by a `gslice` object.

2 Thus, the expression

```
a[gslice(1, length, stride)] = b
```

has the effect of assigning the elements of `b` to a generalized slice of the elements in `a`.

3 Note that programmers may not instantiate `gslice_array`, since all its constructors are private. It is intended purely as a helper class and should be transparent to the user.

26.2.5.1 `gslice_array` constructors [lib.gslice.array.cons]

```
gslice_array::gslice_array();
gslice_array::gslice_array(const gslice_array&);
```

1 The `gslice_array` template has no public constructors. It declares the above constructors to be private. These constructors need not be defined.

26.2.5.2 `gslice_array` assignment [lib.gslice.array.assign]

```
void operator=(const valarray<T>&) const;
gslice_array& operator=(const gslice_array&);
```

1 The second of these two assignment operators is declared private and need not be defined. The first has reference semantics, assigning the values of the argument array elements to selected elements of the `valarray<T>` object to which the `gslice_array` refers.

26.2.5.3 `gslice_array computed assignment` [lib.gslice.array.comp.assign]

```
void gslice_array::operator*= (const valarray<T>&) const;
void gslice_array::operator/= (const valarray<T>&) const;
void gslice_array::operator%= (const valarray<T>&) const;
void gslice_array::operator+= (const valarray<T>&) const;
void gslice_array::operator-= (const valarray<T>&) const;
void gslice_array::operator^= (const valarray<T>&) const;
void gslice_array::operator&= (const valarray<T>&) const;
void gslice_array::operator|= (const valarray<T>&) const;
void gslice_array::operator<<=(const valarray<T>&) const;
void gslice_array::operator>>=(const valarray<T>&) const;
```

1 These computed assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the `valarray<T>` object to which the `gslice_array` object refers.

26.2.5.4 `gslice_array` fill function [lib.gslice.array.fill]

```
void fill(const T&);
```

1 This function has reference semantics, assigning the value of its argument to the elements of the `valarray<T>` object to which the `gslice_array` object refers.

26.2.6 Template class `mask_array` [lib.template.mask.array]

```
template <class T> class mask_array {
public:
    void operator=  (const valarray<T>&) const;
    void operator*= (const valarray<T>&) const;
    void operator/= (const valarray<T>&) const;
    void operator%= (const valarray<T>&) const;
    void operator+= (const valarray<T>&) const;
    void operator-= (const valarray<T>&) const;
    void operator^= (const valarray<T>&) const;
    void operator&= (const valarray<T>&) const;
    void operator|= (const valarray<T>&) const;
    void operator<<=(const valarray<T>&) const;
    void operator>>=(const valarray<T>&) const;

    void fill(const T&);
private:
    mask_array();
    mask_array(const mask_array&);
    mask_array& operator=(const mask_array&);
//  remainder implementation defined
};
```

1 This template is a helper template used by the mask subscript operator

```
mask_array<T> valarray<T>::operator[](const valarray<bool>&);
```

It has reference semantics to a subset of an array specified by a boolean mask. Thus, the expression

```
a[mask] = b;
```

has the effect of assigning the elements of `b` to the masked elements in `a` (those for which the corresponding element in `mask` is `true`.

2 Note that C++ programs may not declare instances of `mask_array`, since all its constructors are private. It is intended purely as a helper class, and should be transparent to the user.

26.2.6.1 `mask_array` constructors [lib.mask.array.cons]

```
mask_array::mask_array();
mask_array::mask_array(const mask_array&);
```

1 The `mask_array` template has no public constructors. It declares the above constructors to be private. These constructors need not be defined.

26.2.6.2 `mask_array` assignment [lib.mask.array.assign]

```
void operator=(const valarray<T>&) const;
mask_array& operator=(const mask_array&);
```

1 The second of these two assignment operators is declared private and need not be defined. The first has reference semantics, assigning the values of the argument array elements to selected elements of the `valarray<T>` object to which it refers.

26.2.6.3 `mask_array` computed assignment [lib.mask.array.comp.assign]

```
void mask_array::operator*= (const valarray<T>&) const;
void mask_array::operator/= (const valarray<T>&) const;
void mask_array::operator%= (const valarray<T>&) const;
void mask_array::operator+= (const valarray<T>&) const;
void mask_array::operator-= (const valarray<T>&) const;
void mask_array::operator^= (const valarray<T>&) const;
void mask_array::operator&= (const valarray<T>&) const;
void mask_array::operator|= (const valarray<T>&) const;
void mask_array::operator<<=(const valarray<T>&) const;
void mask_array::operator>>=(const valarray<T>&) const;
```

1 These computed assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the `valarray<T>` object to which the mask object refers.

26.2.6.4 `mask_array` fill function [lib.mask.array.fill]

```
void fill(const T&);
```

This function has reference semantics, assigning the value of its argument to the elements of the `valarray<T>` object to which the `mask_array` object refers.

26.2.7 Template class `indirect_array` [lib.template.indirect.array]

```
template <class T> class indirect_array {
public:
    void operator=  (const valarray<T>&) const;
    void operator*= (const valarray<T>&) const;
    void operator/= (const valarray<T>&) const;
    void operator%= (const valarray<T>&) const;
    void operator+= (const valarray<T>&) const;
    void operator-= (const valarray<T>&) const;
    void operator^= (const valarray<T>&) const;
    void operator&= (const valarray<T>&) const;
    void operator|= (const valarray<T>&) const;
    void operator<<=(const valarray<T>&) const;
    void operator>>=(const valarray<T>&) const;
```

```
        void fill(const T&);
    private:
        indirect_array();
        indirect_array(const indirect_array&);
        indirect_array& operator=(const indirect_array&);
    //  remainder implementation defined
    };
```

1 This template is a helper template used by the indirect subscript operator

```
    indirect_array<T> valarray<T>::operator[](const valarray<int>&);
```

It has reference semantics to a subset of an array specified by an `indirect_array`. Thus the expression

```
    a[indirect] = b;
```

has the effect of assigning the elements of `b` to the elements in `a` whose indices appear in `indirect`.

2 Note that C++ programs may not declare instances of `indirect_array`, since all its constructors are private. It is intended purely as a helper class, and should be transparent to the user.

26.2.7.1 `indirect_array` constructors [lib.indirect.array.cons]

```
    indirect_array::indirect_array();
    indirect_array::indirect_array(const indirect_array&);
```

The `indirect_array` template has no public constructors. The constructors listed above are private. These constructors need not be defined.

26.2.7.2 `indirect_array` assignment [lib.indirect.array.assign]

```
    void operator=(const valarray<T>&) const;
    indirect_array& operator=(const indirect_array&);
```

1 The second of these two assignment operators is declared private and need not be defined. The first has reference semantics, assigning the values of the argument array elements to selected elements of the `valarray<T>` object to which it refers.

2 If the `indirect_array` specifies an element in the `valarray<T>` object to which it refers more than once, the behavior is undefined.

3 For example,

```
    int addr = {2, 3, 1, 4, 4};
    valarray<int> indirect(addr, 5);
    valarray<double> a(0., 10), b(1., 5);
    array[indirect] = b;
```

results in undefined behavior since element 4 is specified twice in the indirection.

26.2.7.3 `indirect_array` computed assignment [lib.indirect.array.comp.assign]

```
    void indirect_array::operator*= (const valarray<T>&) const;
    void indirect_array::operator/= (const valarray<T>&) const;
    void indirect_array::operator%= (const valarray<T>&) const;
    void indirect_array::operator+= (const valarray<T>&) const;
    void indirect_array::operator-= (const valarray<T>&) const;
    void indirect_array::operator^= (const valarray<T>&) const;
    void indirect_array::operator&= (const valarray<T>&) const;
    void indirect_array::operator|= (const valarray<T>&) const;
    void indirect_array::operator<<=(const valarray<T>&) const;
    void indirect_array::operator>>=(const valarray<T>&) const;
```

1 These computed assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the `valarray<T>` object to which the `indirect_array` object refers.

2 If the `indirect_array` specifies an element in the `valarray<T>` object to which it refers more than once, the behavior is undefined.

26.2.7.4 `indirect_array` fill function [lib.indirect.array.fill]

```
void fill(const T&);
```

1 This function has reference semantics, assigning the value of its argument to the elements of the `valarray<T>` object to which the `indirect_array` object refers.

26.3 Generalized numeric operations [lib.numeric.ops]

1 Headers:

— `<stl numerics (TBD)>`

2 Table 90:

Table 90—Header `<stl numerics (TBD)>` synopsis

Type	Name(s)
Template functions:	
`accumulate [2]`	`inner_product [2]`
`adjacent_difference [2]`	`partial_sum [2]`

26.3.1 Accumulate [lib.accumulate]

```
template <class InputIterator, class T>
T accumulate(InputIterator first, InputIterator last, T init);

template <class InputIterator, class T, class BinaryOperation>
T accumulate(InputIterator first, InputIterator last, T init,
             BinaryOperation binary_op);
```

1 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.[123] `binary_op` is assumed not to cause side effects.

26.3.2 Inner product [lib.inner.product]

```
template <class InputIterator1, class InputIterator2, class T>
T inner_product(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, T init);

template <class InputIterator1, class InputIterator2, class T,
          class BinaryOperation1, class BinaryOperation2>
T inner_product(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, T init,
                BinaryOperation1 binary_op1, BinaryOperation2 binary_op2);
```

[123] `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.

1 Computes its result by initializing the accumulator `acc` with the initial value `init` and then modifying it with

```
acc = acc + (*i1) * (*i2)
```

or

```
acc = binary_op1(acc, binary_op2(*i1, *i2))
```

for every iterator `i1` in the range `[first, last)` and iterator `i2` in the range `[first2, first2 + (last - first))` in order.

2 `binary_op1` and `binary_op2` shall not cause side effects.

26.3.3 Partial sum [lib.partial.sum]

```
    template <class InputIterator, class OutputIterator>
    OutputIterator partial_sum(InputIterator first, InputIterator last,
                               OutputIterator result);

    template <class InputIterator, class OutputIterator, class BinaryOperation>
    OutputIterator partial_sum(InputIterator first, InputIterator last,
                               OutputIterator result, BinaryOperation binary_op);
```

1 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)))
```

2 Returns `result + (last - first)`.

3 Complexity: Exactly `(last - first) - 1` applications of `binary_op` are performed.

4 `binary_op` is expected not to have any side effects. `result` may be equal to `first`.

26.3.4 Adjacent difference [lib.adjacent.difference]

```
    template <class InputIterator, class OutputIterator>
    OutputIterator adjacent_difference(InputIterator first, InputIterator last,
                                       OutputIterator result);

    template <class InputIterator, class OutputIterator, class BinaryOperation>
    OutputIterator adjacent_difference(InputIterator first, InputIterator last,
                                       OutputIterator result, BinaryOperation binary_op);
```

1 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`.

2 `binary_op` shall not have any side effects. `result` may be equal to `first`.

3 Returns `result + (last - first)`.

4 Complexity: Exactly `(last - first) - 1` applications of `binary_op` are performed.

26.4 C Library [lib.c.math]

1 Headers:

— `<cmath>`

— `<cstdlib> abs(), div(), rand(), srand()`

2 Table 91:

Table 91—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`

3 Table 92:

Table 92—Header `<cstdlib>` synopsis

Type	**Name(s)**	
Macros:	`RAND_MAX`	
Types:	`div_t`	`ldiv_t`
Functions:		
`abs`	`labs`	`srand`
`div`	`ldiv`	`rand`

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

SEE ALSO: ISO C subclauses 7.5, 7.10.2, 7.10.6.

27 Input/output library [lib.input.output]

1 This clause describes components that C++ programs may use to perform input/output operations.

2 The following subclauses describe components for all iostreams (27.1), base stream buffer and streams classes (27.2), stream manipulators (27.3), string streams (27.4), and file streams (27.5).

27.1 Iostreams [lib.iostreams]

1 Headers:

— `<ios>`

— `<iostream>`

2 Table 93:

Table 93—Header `<ios>` synopsis

Type	**Name(s)**		
Template class:	`basic_ios`		
Template structs:	`ios_baggage`	`ios_char_baggage`	`ios_pos_baggage`
Template functions:			
`basic_dec`	`basic_noshowbase`	`basic_oct`	`basic_showpos`
`basic_fixed`	`basic_noshowpoint`	`basic_right`	`basic_skipws`
`basic_hex`	`basic_noskipws`	`basic_scientific`	`basic_uppercase`
`basic_internal`	`basic_nouppercase`	`basic_showbase`	
`basic_left`	`basic_nowshowpos`	`basic_showpoint`	
Classes:	`basic_ios<char>`	`basic_ios<wchar_t>`	`ios`
Structs:			
`ios_baggage<char>`		`ios_pos_baggage<streampos>`	
`ios_baggage<wchar_t>`		`ios_pos_baggage<wstreampos>`	
`ios_char_baggage<char>`			
`ios_char_baggage<wchar_t>`			
Functions:			
`dec`	`noshowbase`	`oct`	`showpos`
`fixed`	`noshowpoint`	`right`	`skipws`
`hex`	`noshowpos`	`scientific`	`uppercase`
`internal`	`noskipws`	`showbase`	
`left`	`nouppercase`	`showpoint`	

3 Table 94:

Table 94—Header `<iostream>` synopsis

Type	Name(s)			
Objects:	`cerr`	`cin`	`clog`	`cout`

Box 104

Is it clear that these can be declared using incomplete types? I'm a little unhappy about this name because people are going to assume that iostream is essentially the same as iostream.h.

4 Additional definitions:

— **Character** In this clause, the term ‘‘character’’ means the generalized element of text data. Each text data consist of a sequence of character. So the term does not only means the `char` type object, and the `wchar_t` type object, but also any sort of classes which provides the definitions specified in (_lib.TBD_).

— **Character container type** Character container type is a class or a type which represent a *character*. It is used for one of the template parameter of the template IOStream classes.

— **The template IOStream classes** The template IOStream classes are the template classes which takes two template parameters: `charT` and `baggage`. The class, `charT`, represents the character container class and the class, baggage, represents the baggage structure which provides the definition of the functionality necessary to implement the template IOStream classes.

— **Narrow-oriented IOStream classes** The narrow-oriented IOStream classes are the versions of the template IOStream classes specialized with the character container class, `char`. The traditional IOStream classes aree regarded as the narrow-oriented IOStream classes.

— **Wide-oriented IOStream classes** The wide-oriented IOStream classes are the versions of the template IOStream classes specialized with the character container class, `wchar_t`.

27.1.1 Iostream character traits [lib.iostream.traits]

27.1.1.1 Template struct `ios_baggage` [lib.ios.baggage]

Box 105

At the Kitchener meeting, the Library WG decided to change the names used from ‘‘`baggage`’’ to ‘‘`traits`’’. However, Tom Plum objected to doing this without a formal proposal and vote. These names are therefore unresolved, and subject to change.

Box 106

I don’t find the two level approach of `ios_baggage` useful. I think it will simplify everyone’s (i.e. both our and programmer’s) life and shorten the working paper if we combine all the baggages (or traits) needed by any class in the standard into a single template. Using multiple inheritance we can arrange for everything to be specified exactly once, and only the `wchar_t` version needs to be specialized. The `char` version is generic.

1 Proposal: Combine the following two into one

```
template <class charT> struct char_traits<charT> {

    typedef char char_type;
    typedef int  int_type;

    static char_type to_char_type(int_type c) { return c; }
// etc.
```

```
template <class charT> struct ios_baggage {};

struct ios_baggage<char> {
    typedef ios_char_baggage<char> char_bag;
    typedef ios_pos_baggage<streampos> pos_bag;
};

struct ios_baggage<wchar_t> {
    typedef ios_char_baggage<wchar_t> char_bag;
    typedef ios_pos_baggage<wstreampos> pos_bag;
};
```

2 The template struct `ios_baggage<charT>` is an baggage class which maintains the definitions of the types and functions necessary to implement the template IOStream classes. The template parameter `charT` represents the *character container type* and each specialized version provides the default definitions corresponding to the specialized character container type.

3 All of the types and functions provided in this struct can be classified into two categories, character-container-type-related and positional-information-related. Each of the specialized struct, `ios_baggage<CHAR_T>`, in which `CHAR_T` represents the specialized type of the character container type, typedefs two specialized template structs, `ios_char_baggage<CHAR_T>` and `ios_pos_baggage<POS_T>`, in which `POS_T` represents a specialized version of the *repositioning information class*.

27.1.1.2 Template struct `ios_char_baggage` [lib.ios.char.baggage]

```
  template <class charT> struct ios_char_baggage :
      public string_char_baggage<charT> {};
```

1 Every implementation shall provide the following two specialization versions of the `ios_char_baggage`[124]:

```
struct ios_char_baggage<char> {
    typedef char char_type;
    typedef int  int_type;
```

[124] The two types `wchar_t` and `wint_t` are declared in `<cwchar>`.

```
    static char_type to_char_type(int_type c) { return c; }
    static int_type  to_int_type (char_type c)  { return c; }
    static bool      eq_char_type(char_type c1, char_type c2) { return c1 == c2; }
    static bool      eq_int_type (int_type c1, int_type c2) { return c1 == c2; }
    static int_type  eof()     { return EOF; }
    static int_type  not_eof() { return  ~EOF; }
    static bool      is_eof(int_type c) { return c == EOF; }
    static char_type newline() { return '\n'; }
    static bool      is_whitespace(char_type c, locale::ctype<char> ctype) {
         return ctype.isspace<char>(c); }
    static char_type  eos() { return 0; }
    static size_t     length(const char_type* s) { return strlen(s); }
    static char_type* copy(char_type* dst, const char_type* src, size_t n)
                      { return strcpy(dst,src,size); }
};

struct ios_char_baggage<wchar_t> {
    typedef wchar_t char_type;
    typedef wint_t  int_type;

    static char_type to_char_type(int_type c) { return c; }
    static int_type  to_int_type (char_type c)  { return c; }
    static bool      eq_char_type(char_type c1, char_type c2) { return c1 == c2; }
    static bool      eq_int_type (int_type c1, int_type c2) { return c1 == c2; }
    static int_type  eof()     { return WEOF; }
    static int_type  not_eof() { return ~WEOF; }
    static bool      is_eof(int_type c) { return c == WEOF; }
    static char_type newline() { return L'\n'; }
    static bool      is_whitespace(char_type c, locale::ctype<char_type> ctype) {
         return ctype.isspace<char_type>(c); }
    static char_type  eos() { return 0; }
```

Box 107

The following two functions were not defined in the proposal. They should be give the generic definitions. Is that clear?

```
    static size_t     length(const char_type* s);
    static char_type* copy(char_type* dst, const char_type* src, size_t n);
  };
```

Box 108

There are several types and functions needed for implementing the template IOStream classes. Some of these types and functions depend on the definition of the character container type. The collection of these functions describes the behavior which the implementation of the template IOStream class expects to the character container class.

Those who provide a character container class 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.

No special definition/declaration is provided here. The base class (or struct), `string_char_baggage` provides the common definitions common between the template string classes and the template IOStream classes.

2 The template struct `ios_char_baggage<charT>` is a struct derived from the class `string_char_baggage<charT>`. For each of the character container classes, `CHAR_T`, the corresponding specialized struct `ios_char_baggage<CHAR_T>` prepares the definitions related to the character container type so that it provides all the functionality necessary to implement the template IOStream classes.

3 Each of the specialization of the struct `ios_char_baggage` shall have the following definitions:

```
struct ios_char_baggage<CHAR_T> {
    typedef CHAR_T char_type;
    typedef INT_T  int_type;

    static char_type to_char_type (int_type);
    static int_type  to_int_type (char_type);
    static bool      eq_char_type (char_type, char_type);
    static bool      eq_int_type (int_type, int_type);
    static int_type  eof();
    static int_type  not_eof();
```

Box 109

Jerry Schwarz proposal: Change the declaration of not_eof to

```
    static int_type  not_eof(char_type c);
```

with a generic description that it returns a value other than `eof()`.

4 This is the way the function is used by implementations. It can be implemented efficiently in many cases and it provides an easy way for me to specify return values in the working paper.

```
    static bool      is_eof(int_type);
    static char_type newline();
    static bool      is_whitespace(locale::ctype<char_type> ctype, char_type c);
    static char_type  eos();
    static size_t     length(const char_type* s);
    static char_type* copy(char_type* dst, const char_type* src, size_t n) ;
};
```

Box 110

If the proposal for flattening the baggage classes is accepted, these classes will also need to be flattened. A reasonable structure would be

```
class std_pos_traits {
    // note that this is not a template
    typedef streampos pos_t;
    typedef streamoff off_t;
}

template <class charT>
struct std_traits<charT>
    : public std_char_traits<charT>, public std_pos_traits { }

struct std_traits<wchar_t> : public std_char_traits<wchar_t> {
    typedef wstreampos pos_t;
    typedef wstreamoff off_t;
    // implementations are free to replace any other function provided
    // it satisfies the generic constraints for the given trait.
}
```

27.1.1.2.1 Type `ios_char_baggage::INT_T` [lib.ios.char.baggage::int.t]

```
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.

27.1.1.2.2 Type `ios_char_baggage::char_type` [lib.ios.char.baggage::char.type]

```
char_type
```

1 A synonym of the character container type, `CHAR_T`.

27.1.1.2.3 Type `ios_char_baggage::int_type` [lib.ios.char.baggage::int.type]

```
int_type
```

1 A synonym of the character container type, `INT_T`.

27.1.1.2.4 `ios_char_baggage::to_char_type` [lib.ios.char.baggage::to.char.type]

```
char_type to_char_type(int_type c);
```

1 Converts a valid character value represented in the `int_type` to the corresponding `char_type` value. If *`c`* is the end-of-file value, the return value is unspecified.

27.1.1.2.5 `ios_char_baggage::to_int_type` [lib.ios.char.baggage::to.int.type]

```
int_type to_int_type(char_type c);
```

1 Converts a valid character value represented in the `char_type` to the corresponding `int_type` value.

27.1.1.2.6 `ios_char_baggage::eq_char_type` [lib.ios.char.baggage::eq.char.type]

```
bool eq_char_type(char_type c1, char_type c2);
```

1 Returns `true` if *`c1`* and *`c2`* represent the same character.

27.1.1.2.7 `ios_char_baggage::eq_int_type` [lib.ios.char.baggage::eq.int.type]

```
bool eq_int_type(int_type c1, int_type c2);
```

1 Returns `true` if *`c1`* and *`c2`* represent the same character.

27.1.1.2.8 `ios_char_baggage::eof` [lib.ios.char.baggage::eof]

```
int_type eof();
```

1 Returns an `int_type` value which represents the end-of-file. It is returned by several functions to indicate end-of-file state (no more input from an input sequence or no more output permitted to an output sequence), or to indicate an invalid return value.

27.1.1.2.9 `ios_char_baggage::not_eof` [lib.ios.char.baggage::not.eof]

```
int_type not_eof();
```

1 Returns a certain `int_type` value other than the end-of-file. It is used in `basic_streambuf<charT,baggage>::overflow()`.

27.1.1.2.10 `ios_char_baggage::is_eof` [lib.ios.char.baggage::is.eof]

```
bool is_eof(int_type c);
```

1 Returns `true` if *c* represent the end-of-file.

27.1.1.2.11 `ios_char_baggage::newline` [lib.ios.char.baggage::newline]

```
char_type newline();
```

1 Returns a character value which represent the newline character of the basic character set. It appears as the default parameter of `basic_istream<charT,baggage>::getline()`.

27.1.1.2.12 `ios_char_baggage::is_whitespace` [lib.ios.char.baggage::is.whitespace]

```
bool is_whitespace(char_type c, locale::ctype<char_type> ctype);
```

1 Returns `true` if *c* represents one of the white space characters. The default definition is as if it returns *ctype*`.isspace(c)`.

2 An implementation of the template IOStream classes may use all of the above static member functions in addition to the following three functions provided from the base struct `string_char_baggage<CHAR_T>`.

27.1.1.2.13 `ios_char_baggage::eos` [lib.ios.char.baggage::eos]

```
char_type eos();
```

1 Returns the null character which is used for the terminator of null terminated character strings. The default constructor for the character container type provides the value.

27.1.1.2.14 `ios_char_baggage::length` [lib.ios.char.baggage::length]

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

1 Determines the length of a null terminated character string pointed to by *s*.

27.1.1.2.15 `ios_char_baggage::copy` [lib.ios.char.baggage::copy]

```
char_type* copy(char_type* dest, const char_type* src, size_t n);
```

1 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.

Box 111

The reason that these two member functions, `length` and `copy`, are prepared is to achive efficiency for `char` and `wchar_t` specialized versions. If the character container type require no special construction/destruction operations (or it is a immutable class), the memory copy operation leads us to avoid inefficiency which an explicit copy loop in the implementation of the template IOStream classes give rise to.

27.1.2 Positional information [lib.positional]

Box 112

The description of the following member functions is incomplete. The behavior and specification of the `streampos`/`streamoff` should need more discussion.

27.1.2.1 Template struct `ios_pos_baggage` [lib.ios.pos.baggage]

```
template <class posT> struct ios_pos_baggage {};

struct ios_pos_baggage<streampos> {
    typedef streampos pos_type;
    typedef streamoff off_type;
};

struct ios_pos_baggage<wstreampos> {
    typedef wstreampos pos_type;
    typedef wstreamoff off_type;
};
```

1 The template struct `ios_pos_baggage<posT>` is a struct. For each of the repositional information class, `POS_T`, the corresponding specialized struct `ios_pos_baggage<POS_T>` prepares the typedefs related to the `POS_T`.

2

Box 113

Jerry Schwarz proposal: Rewrite the following sections to clarify that we are talking about generic requirements for members of a user supplied `baggage::pos_bag` and not only `ios_char_baggage` specializations.

Each of the specialization of the struct `ios_pos_baggage` shall have the following definitions:

```
  struct ios_pos_baggage<POS_T> {
      typedef POS_T pos_type;
      typedef OFF_T off_type;
};
```

27.1.2.1.1 Type `ios_pos_baggage::OFF_T` [lib.ios.pos.baggage::off.t]

```
    OFF_T;
```

1 A synonym for one of the signed integral types whose representation has at least as many bits as type `long`.

27.1.2.1.2 Type `ios_pos_baggage::off_type` [lib.ios.pos.baggage::off.type]

```
   off_type;
```

1 A synonym of the `OFF_T` and is used for declaring/implementing the template IOStream class.

27.1.2.1.3 Type `ios_pos_baggage::POS_T` [lib.ios.pos.baggage::pos.t]

```
    POS_T;
```

1 An implementation-defined class for seek operations which describes an object that can store all the information necessary to restore to the position.

27.1.2.1.4 Type `ios_pos_baggage::pos_type` [lib.ios.pos.baggage::pos.type]

```
    pos_type;
```

1 A synonym for the class `POS_T` which is used for declaring/implementing the template IOStream classes.

27.1.2.2 Type OFF_T requirements [lib.off.t.reqmts]

1 The type OFF_T is a synonym for one if the signed basic integral types T1 whose representation at least as many bits as type long. It is used to represent:

— a signed displacement, measured in characters, from a specified position within a sequence.

— an absolute position within a sequence.

Box 114

Jerry Schwarz proposal: Revrite the following so that it is clear that certain operations are possible on baggage::char_bag::pos_type but that it need not be a class. One of the advantages of the templatizing approach is that implementations can make ios_pos_baggage<char>::pos_type the same as their current streampos.

27.1.2.3 Type POS_T requirements [lib.pos.t.reqmts]

1 Every class that can apply to the template parameter posT in the template struct ios_pos_baggage shall have following definitions:

```
    typedef T1 OFF_T;
```

Box 115

Jerry Schwarz proposal: Replace POS_T::offset by fPOS_T::operator OFF_T. The point is to allow POS_T to be a non-class.

```
class POS_T {
public:
    POS_T(int off);
    POS_T(OFF_T off = 0);
    OFF_T  offset() const;
    OFF_T  operator- (POS_T& rhs);
    POS_T& operator+=(OFF_T off);
    POS_T& operator-=(OFF_T off);
    POS_T  operator+ (OFF_T off);
    POS_T  operator- (OFF_T off);
    bool   operator==(POS_T& rhs) const;
    bool   operator!=(POS_T& rhs) const;
};
```

Box 116

The asymmetries implied by making operator+, operator-, operator== and operator!= members rather than global operators are unpleasant.

Box 117

This subclause need more discussion. How do we treat the case if the external source/sink stream does not ensure to accept POS_T, OFF_T object to which some arithmetic operations performed?

2 The class POS_T describes an object that can store all the information necessary to restore one or more types of sequences to a previous stream position. Every POS_T class has the corresponding OFF_T type and the set of applicable stream classes.

— **Repositional Streams and Arbitrary-positional Streams** There are two types of stream: repositional and arbitrary-positional.

Box 118
There are also non-positional streams in which no seeking is possible.

3 With a *repositional stream*, we can seek to only the position where we previously encountered. On the other hand, with an *arbitrary-positional* stream, we can seek to any integral position within the length of the stream.

4 For a stream buffer which is corresponding to a repositional stream (but not a arbitrary-positional stream), all we can do is either to fetch the current position of the stream buffer or to specify the previous position which we have already fetched from the stream buffer.

5 Every arbitrary-positional stream is a repositional.

6 If a repositional stream returns a `POS_T` object, and some arithmetic operations (`operator+=`, `operator-`, `operator+=`, `operator-=`) are applied, the behavior of the stream after restoring the position with the modified `POS_T` object is undefined. It means that a `POS_T` object whose parent stream is repositional shall not apply any arithmetic operations.

7 It is not ensured that in case the validity of a certain `POS_T` object is broken, the error shall be informed. Or there is no way to check the validity of a certain `POS_T` object.

— **Invalid POS_T value** The stream operations whose return type is `POS_T` may return `POS_T((OFF_T)(-1))` as signal to some error occurs. This return value is called the *invalid POS_T value*. The constructor `POS_T::POS_T((OFF_T)(-1))` constructs the invalid `POS_T` value, which is available only for comparing to the return value of such member functions.

27.1.2.3.1 `POS_T` constructors [lib.pos.t.cons]

```
POS_T(int off);
POS_T(OFF_T off = 0);
```

1 [Almost same as 27.1.2.8.1]

2 Constructs an object of type `POS_T`, initializing *`pos`* to zero and *`fp`* to the stream position at the beginning of the sequence, with the conversion state at the beginning of a new multibyte sequence in the initial shift state.[125] The constructor then evaluates the expression `*this += off`.

27.1.2.3.2 `POS_T::offset` [lib.ios.pos.t.offset]

```
OFF_T  offset() const;
```

1 [Almost same as _lib.streampos.offset_]

2 Determines the value of type `OFF_T` that represents the stream position stored in *`pos`* and *`fp`*, if possible, and returns that value. Otherwise, returns `(OFF_T)(-1)`. For a sequence requiring a conversion state, even a representable value of type `OFF_T` need not supply sufficient information to restore the stored stream position.

27.1.2.3.3 `POS_T::operator-` [lib.ios.pos.t.op-.pos.t]

```
OFF_T  operator- (POS_T& rhs);
```

1 [Almost same as _lib.streampos.op-_]

2 Determines the value of type `POS_T` that represents the difference in stream positions between `*this` and *`rhs`*, if possible, and returns that value. (If `*this` is a stream position nearer the beginning of the sequence than *`rhs`*, the difference is negative.) Otherwise, returns `(streamoff)(-1)`. For a sequence

[125] The next character to read or write is the first character in the sequence.

that does not represent stream positions in uniform units, even a representable value need not be meaningful.

3 If the parent stream is not arbitrary-positional, the value of the operation becomes invalid so it may not apply the parent stream.

27.1.2.3.4 `POS_T::operator+=` [lib.ios.pos.t.op+=]

```
POS_T& operator+=(OFF_T off);
```

1 [Almost same as _lib.streampos.op+=_]

2 Adds *off* to the stream position stored in *pos* and *fp*, if possible, and replaces the stored values. Otherwise, the function stores an invalid stream position in *pos* and *fp*. For a sequence that does not represent stream positions in uniform units, the resulting stream position need not be meaningful.

3 Returns `*this`.

4 If the parent stream is not arbitrary-positional, the value of the operation becomes invalid so it may not apply the parent stream.

27.1.2.3.5 `POS_T::operator-=` [lib.ios.pos.t.op-=]

```
POS_T& operator-=(OFF_T off);
```

1 [Almost same as _lib.streampos.op-=_]

2 Subtracts *off* from the stream position stored in *pos* and *fp*, if possible, and replaces the stored value. Otherwise, the function stores an invalid stream position in *pos* and *fp*. For a sequence that does not represent stream positions in uniform units, the resulting stream position need not be meaningful.

3 Returns `*this`.

4 If the parent stream is not arbitrary-positional, the value of the operation becomes invalid so it may not apply the parent stream.

27.1.2.3.6 `POS_T::operator+` [lib.ios.pos.t.op+]

```
POS_T  operator+ (OFF_T off);
```

1 [Almost same as _lib.streampos.op+_]

2 Returns `POS_T(*this) += off`.

27.1.2.3.7 `POS_T::operator-` [lib.ios.pos.t.op-.off.t]

```
POS_T  operator- (OFF_T off);
```

1 [Almost same as _lib.streampos.op-_]

2 Returns `POS_T(*this) -= off`.

27.1.2.3.8 `POS_T::operator==` [lib.ios.pos.t.op==]

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

1 [Almost same as _lib.streampos.op==_]

2 Compares the stream position stored in `*this` to the stream position stored in *rhs*, and returns `true` if the two correspond to the same position within a file or if both store an invalid stream position.

27.1.2.3.9 POS_T::operator!= [lib.ios.pos.t.op!=]

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

1 [Almost same as _lib.streampos.op!=_]

2 Returns `true` if `!(*this == rhs)`.

27.1.2.4 Type streamoff [lib.streamoff]

```
        typedef T2 streamoff;
```

1 The type `streamoff` is a synonym for one of the signed basic integral types *T1* whose representation has at least as many bits as type `long`. The corresponding `POS_T` class is the class, `streampos`. It is used to represent:

— a signed displacement, measured in bytes, from a specified position within a sequence;

2 If a `streamoff` object has a value other than the parent stream returns (for example, assigned an arbitrary integer), the value may not apply any streams.

3 Any `streamoff` object can be converted to a `streampos` object is ensured. But no validity of the result `streampos` object is ensured, whether the `streamoff` object is valid or else.

27.1.2.5 Type streamsize [lib.streamsize]

```
        typedef T3 streamsize;
```

1 The type `streamsize` is a synonym for one of the signed basic integral types. It is used to represent the number of characters transferred in an I/O operation, or the size of I/O buffers.[126)]

27.1.2.6 Class wstreamoff [lib.wstreamoff]

1 [Almost the same description as in streamoff]

27.1.2.7 Class streampos [lib.streampos]

1 In this subclause, the type name *fpos_t* is a synonym for the type `fpos_t` defined in `<cstdio>` (27.5).

```
class streampos {
public:
    streampos(streamoff off = 0);
    streamoff  offset() const;
    streamoff  operator-(streampos& rhs) const;
    streampos& operator+=(streamoff off);
    streampos& operator-=(streamoff off);
    streampos  operator+(streamoff off) const;
    streampos  operator-(streamoff off) const;
    bool       operator==(const streampos& rhs) const;
    bool       operator!=(const streampos& rhs) const;
private:
//  streamoff pos;          exposition only
//  fpos_t fp;              exposition only
};
```

126) `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.

The `streamsize` type should probably be the signed type corresponding to `size_t` (which is what Posix.2 calls `ssize_t`).

27.1.2.8 Class `wstreampos` [lib.wstreampos]

1 [Almost the same description as in streampos]

2 The class `streampos` describes an object that can store all the information necessary to restore an arbitrary sequence, controlled by the Standard C++ library, to a previous *stream position* and *conversion state*.[127)] For the sake of exposition, the data it stores is presented here as:

— `streamoff` *`pos`*, specifies the absolute position within the sequence;

— *`fpos_t fp`*, specifies the stream position and conversion state in the implementation-dependent form required by functions declared in `<cstdio>`.

3 It is unspecified how these two member objects combine to represent a stream position.

27.1.2.8.1 `streampos` constructor [lib.streampos.cons]

Box 119

One of the advantages of the template approach is that `streampos` can does not have to hold the state information required for wide oriented streams.

1 Jerry Schwarz proposal: Replace this section with: `streaampos` satisfies the generic constraints on a `POS_T`.

```
streampos(streamoff off = 0);
```

2 Constructs an object of class `streampos`, initializing *`pos`* to zero and *`fp`* to the stream position at the beginning of the sequence, with the conversion state at the beginning of a new multibyte sequence in the initial shift state.[128)] The constructor then evaluates the expression `*this +=` *`off`*.

27.1.2.8.2 `streampos::offset` [lib.streampos::offset]

```
streamoff offset() const;
```

1 Determines the value of type `streamoff` that represents the stream position stored in *`pos`* and *`fp`*, if possible, and returns that value. Otherwise, returns `(streamoff)(-1)`.

2 For a sequence requiring a conversion state, even a representable value of type `streamoff` need not supply sufficient information to restore the stored stream position.

27.1.2.8.3 `streampos::operator-` [lib.streampos::op-.sp]

```
streamoff operator-(streampos& rhs) const;
```

1 Determines the value of type `streamoff` that represents the difference in stream positions between `*this` and *`rhs`*, if possible, and returns that value. (If `*this` is a stream position nearer the beginning of the sequence than *`rhs`*, the difference is negative.) Otherwise, returns `(streamoff)(-1)`.

2 For a sequence that does not represent stream positions in uniform units, even a representable value need not be meaningful.

127) 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.
128) The next character to read or write is the first character in the sequence.

27.1.2.8.4 `streampos::operator+=` [lib.streampos::op+=]

```
streampos& operator+=(streamoff off);
```

1 Adds *off* to the stream position stored in *pos* and *fp*, if possible, and replaces the stored values. Otherwise, the function stores an invalid stream position in *pos* and *fp*.

2 For a sequence that does not represent stream positions in uniform units, the resulting stream position need not be meaningful.

3 Returns `*this`.

27.1.2.8.5 `streampos::operator-=` [lib.streamos::op-=]

```
streampos& operator-=(streamoff off);
```

1 Subtracts *off* from the stream position stored in *pos* and *fp*, if possible, and replaces the stored value. Otherwise, the function stores an invalid stream position in *pos* and *fp*.

2 For a sequence that does not represent stream positions in uniform units, the resulting stream position need not be meaningful.

3 Returns `*this`.

27.1.2.8.6 `streampos::operator+` [lib.streampos::op+]

```
streampos operator+(streamoff off) const;
```

1 Returns `streampos(*this) +=` *off*.

27.1.2.8.7 `streampos::operator-` [lib.streampos::op-.off]

```
streampos operator-(streamoff off) const;
```

1 Returns `streampos(*this) -=` *off*.

27.1.2.8.8 `streampos::operator==` [lib.streampos::op==]

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

1 Compares the stream position stored in `*this` to the stream position stored in *rhs*, and returns `true` if the two correspond to the same position within a file or if both store an invalid stream position.

27.1.2.8.9 `streampos::operator!=` [lib.op!=.streampos]

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

1 Returns `true` if `!(*this ==` *rhs*`)`.

27.1.3 Base class ios and manipulators [lib.ios.base]

1 The header `<ios>` defines three types that controls input from and output to character sequence.

27.1.3.1 Template class `basic_ios` [lib.ios]

```
template<class charT, class baggage = ios_baggage<charT> >
class basic_ios {
    typedef basic_ios<charT,baggage> ios_type;
public:
    typedef charT char_type;
    typedef baggage::char_bag::int_type int_type;
    typedef baggage::pos_bag::pos_type  pos_type;
    typedef baggage::pos_bag::off_type  off_type;
    int_type  eof()     { return baggage::char_bag::eof(); }
    char_type newline() { return baggage::char_bag::newline(); }
```

Box 120

Note: An idea to declare `eof` as a constant, that is,

```
  static const IEOF = baggage::char_bag::eof(),
```

is rejected because it cause a constraint that `charT` and `int_type` should be integral types.

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

    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 {
    public:
        Init();
       ~Init();
    private:
//  static int init_cnt;          exposition only
    };

    basic_ios(basic_streambuf<charT,baggage>* sb_arg);
    virtual ~basic_ios();
    operator bool() const
    bool operator!() const
    ios_type& copyfmt(const ios_type& rhs);
    basic_ostream<charT,baggage>* tie() const;
    basic_ostream<charT,baggage>* tie(basic_ostream<charT,baggage>* tiestr_arg);
    basic_streambuf<charT,baggage>* rdbuf() const;
    basic_streambuf<charT,baggage>* rdbuf(basic_streambuf<charT,baggage>* sb_arg);
```

Box 121

Note: template parameter coupling between `basic_ios` and `basic_streambuf`: Both `basic_ios` and `basic_streambuf` corresponding to it should take the same template parameter `<charT,baggage>`. We need not allow to make a couple of `basic_ios<wchar_t>` and `basic_streambuf<char>`.

```
    iostate rdstate() const;
    void clear(iostate state_arg = goodbit);
    void setstate(iostate state_arg);
    bool good() const;
    bool eof()  const;
    bool fail() const;
    bool bad()  const;

    iostate exceptions() const;
    void exceptions(iostate except_arg);
    fmtflags flags() const;
    fmtflags flags(fmtflags fmtfl_arg);
    fmtflags setf(fmtflags fmtfl_arg);
    fmtflags setf(fmtflags fmtfl_arg, fmtflags mask);
    void unsetf(fmtflags mask);
```

Box 122

Specifying fill character. It is represented as `int_type`.

```
    int_type fill() const;
    int_type fill(int_type ch);
    int precision() const;
    int precision(int prec_arg);
    int width() const;
    int width(int wide_arg);
    locale imbue(const locale& loc_arg);
    locale getloc() const;
    static int xalloc();
    long&  iword(int index_arg);
    void*& pword(int index_arg);

protected:
    basic_ios();
    void init(basic_streambuf<charT,baggage>* sb_arg);
private:
//  basic_streambuf<charT,baggage>* sb;          exposition only
//  basic_ostream<charT,baggage>* tiestr;        exposition only
//  iostate state;        exposition only
//  iostate except;       exposition only
//  fmtflags fmtfl;       exposition only
//  int prec;             exposition only
//  int wide;             exposition only
//  char fillch;          exposition only
//  locale loc;           exposition only
//  static int index;     exposition only
//  int* iarray;          exposition only
//  void** parray;        exposition only
};
```

Box 123

Lots of things that should be the same in all instantiations of `basic_ios` are made members. In particular it seems that `failure, fmtflags, iostate, openmode, seekdir`, and all the constants of these types should be the same in different instances.

Jerry Schwarz proposal:

```
class ios_constants {
    class failure .... ;
    typedef T1 fmtflags;
    ....; // bits of fmtflags;
    typedef T2 iostate;
    ....; // bits of iostate
    typedef T3 openmode;
    ....; // bits of openmode
    typedef T4 seekdir;
    ....; // bits of seekdir
};

template<class charT, class traits = std_traits<charT> >
class basic_ios : public ios_constants {
    ....
};
```

```
    class ios : public basic_ios<char> {}

  class wios : public basic_ios<wchar_t> {}
```

Box 124

Jerry Schwarz proposal: Replace with

```
    typedef basic_ios<char>    ios ;
    typedef basic_ios<wchar_t> wios;
```

1 The template class `basic_ios<charT,baggage>` serves as a base class for the classes `basic_istream<charT,baggage>` and `basic_ostream<charT,baggage>`.

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`.

4 `basic_ios` defines several member types:

— a class `failure` derived from `exception`;

— a class `Init`;

Box 125

Jerry Schwarz proposal: Eliminate **`basic_ios::Init`** and replace it by a requirement (in _lib.iostreams_) that the standard streams should be initialized before startup. We had a discussion of this in San Diego and I think this was the consensus. The "nifty counter" trick is generally regarded as a bad way to cause initialization of the standard streams because of the overheads.

— three bitmask types, `fmtflags`, `iostate`, and `openmode`;

— an enumerated type, `seekdir`.

5 It maintains several kinds of data:

— a pointer to a *stream buffer*, an object of class `basic_streambuf<charT,baggage>`, that controls sources (input) and sinks (output) of *character* sequences; the parameter `charT` specifies the type of character;

— 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.

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

— `basic_streambuf<charT,baggage>*` *`sb`*, points to the stream buffer;

— `basic_ostream<charT,baggage>*` *`tiestr`*, points to an output sequence that is *tied* to (synchronized with) an input sequence controlled by the stream buffer;

— `iostate` *`state`*, holds the control state of the stream buffer;

— `iostate` *`except`*, holds a mask that determines what elements set in `state` cause exceptions to be thrown;

— `fmtflags` *`fmtfl`*, holds format control information for both input and output;

— `int` *`wide`*, specifies the field width (number of characters) to generate on certain output conversions;

— `int` *`prec`*, specifies the precision (number of digits after the decimal point) to generate on certain output conversions;

— `char` *`fillch`*, specifies the character to use to pad (fill) an output conversion to the specified field width;

— `locale` *`loc`*, specifies the locale in which to perform locale-dependent input and output operations;

— `static int` *`index`*, specifies the next available unique index for the integer or pointer arrays maintained for the private use of the program, initialized to an unspecified value;

— `int*` *`iarray`*, points to the first element of an arbitrary-length integer array maintained for the private use of the program;

— `void**` *`parray`*, points to the first element of an arbitrary-length pointer array maintained for the private use of the program.

27.1.3.1.1 Class `basic_ios::failure` [lib.ios::failure]

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

1 The class `failure` defines the base class for the types of all objects thrown as exceptions, by functions in the iostreams library, to report errors detected during stream buffer operations.

27.1.3.1.1.1 `basic_ios::failure` constructor [lib.basic.ios::failure.cons]

```
    failure(const char* what_arg = 0);
```

1 Constructs an object of class `failure`, initializing the base class with `exception(`*`what_arg`*`)`.

Box 126
Jerry Schwarz proposal: Allow an implementation defined default value.

27.1.3.1.1.2 `basic_ios::failure::~failure` destructor [lib.basic.ios::failure.des]

```
    virtual ~failure();
```

1 Destroys an object of class `failure`.

27.1.3.1.1.3 `basic_ios::failure::what` [lib.ios::failure::what]

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

1 Behaves the same as `exception::what()`.

27.1.3.1.2 Type `basic_ios::fmtflags` [lib.ios::fmtflags]

```
    typedef T1 fmtflags;
```

1 The type `fmtflags` is a bitmask type (indicated here as *`T1`*). Setting its elements has the effects indicated in Table 95:

Table 95—`fmtflags` effects

Element	Effect(s) if set
`boolalpha`	insert and extract `bool` type in alphabetic format
`dec`	converts integer input or generates integer output in decimal base
`fixed`	generate floating-point output in fixed-point notation;
`hex`	converts integer input or generates integer output in hexadecimal base;
`internal`	adds fill characters at a designated internal point in certain generated output;
`left`	adds fill characters on the left (initial positions) of certain generated output;
`oct`	converts integer input or generates integer output in octal base;
`right`	adds fill characters on the right (final positions) of certain generated output;
`scientific`	generates floating-point output in scientific notation;
`showbase`	generates a prefix indicating the numeric base of generated integer output;
`showpoint`	generates a decimal-point character unconditionally in generated floating-point output;
`showpos`	generates a `+` sign in non-negative generated numeric output;
`skipws`	skips leading white space before certain input operations;
`unitbuf`	flushes output after each output operation;
`uppercase`	replaces certain lowercase letters with their uppercase equivalents in generated output.

2 Type `fmtflags` also defines the constants indicated in Table 96:

Table 96—`fmtflags` constants

Constant	Allowable values
`adjustfield`	`left | right | internal`
`basefield`	`dec | oct | hex`
`floatfield`	`scientific | fixed`

27.1.3.1.3 Type `basic_ios::iostate` [lib.ios::iostate]

```
typedef T2 iostate;
```

1 The type `iostate` is a bitmask type (indicated here as *`T2`*). It contains the elements indicated in Table 97:

Table 97—`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.

27.1.3.1.4 Type `basic_ios::openmode` [lib.ios::openmode]

```
typedef T3 openmode;
```

1 The type `openmode` is a bitmask type (indicated here as *`T3`*). It contains the elements indicated in Table 98:

Table 98—`openmode` effects

Element	Effect(s) if set
`app`	seek to end-of-file before each write to the file
`ate`	open a file and seek to end-of-file immediately after opening the file
`binary`	perform input and output in binary mode (as opposed to text mode)
`in`	open a file for input
`out`	open a file for output
`trunc`	truncate an existing file when opening it

27.1.3.1.5 Type `basic_ios::seekdir` [lib.ios::seekdir]

```
typedef T4 seekdir;
```

1 The type `seekdir` is an enumerated type (indicated here as *`T4`*). It contains the elements indicated in Table 99:

Table 99—`seekdir` effects

Element	Meaning
`beg`	request a seek (positioning for subsequent input or output within a sequence) relative to the beginning of the stream
`cur`	request a seek relative to the current position within the sequence
`end`	request a seek relative to the current end of the sequence

27.1.3.1.6 Class `basic_ios::Init` [lib.ios::Init]

Box 127

Jerry Schwarz proposal: Eliminate **`basic_ios::Init`** and replace it by a requirement (in _lib.iostreams_) that the standard streams should be initialized before startup. We had a discussion of this in San Diego and I think this was the consensus. The "nifty counter" trick is generally regarded as a bad way to cause initialization of the standard streams because of the overheads.

```
class Init {
public:
    Init();
   ~Init();
private:
//  static int init_cnt;          exposition only
};
```

1 The class `Init` describes an object whose construction ensures the construction of the four objects declared in `<iostream>` that associate file stream buffers with the standard C streams provided for by the functions declared in `<cstdio>` (27.5).

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

— `static int` *`init_cnt`*, counts the number of constructor and destructor calls for class `Init`, initialized to zero.

27.1.3.1.6.1 `basic_ios::Init` constructor [lib.basic.ios::init.cons]

```
Init();
```

1 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 four objects `cin` (27.1.4.1), `cout` (27.1.4.2), `cerr` (27.1.4.3), and `clog` (27.1.4.4). In any case, the function then adds one to the value stored in *`init_cnt`*.

27.1.3.1.6.2 `basic_ios::Init` destructor [lib.basic.ios::init.des]

```
~Init();
```

1 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()`.

27.1.3.1.7 `basic_ios::basic_ios` constructor [lib.basic.ios.sb.cons]

```
basic_ios(basic_streambuf<charT,baggage>* sb_arg);
```

1 Constructs an object of class `basic_ios`, assigning initial values to its member objects by calling `init(`*`sb_arg`*`)`.

27.1.3.1.8 `basic_ios` destructor [lib.basic.ios.des]

```
virtual ~basic_ios();
```

1 Destroys an object of class `basic_ios`.

27.1.3.1.9 `basic_ios::operator bool` [lib.ios::operator.bool]

```
operator bool() const
```

1 Returns a non-null pointer (whose value is otherwise unspecified) if `failbit | badbit` is set in *`state`*.

27.1.3.1.10 `basic_ios::operator!` [lib.ios::operator!]

```
bool operator!() const
```

1 Returns `true` if `failbit | badbit` is set in *`state`*.

27.1.3.1.11 `basic_ios::copyfmt` [lib.ios::copyfmt]

```
basic_ios<charT,baggage>& copyfmt(const basic_ios<charT,baggage>& rhs);
```

1 Assigns to the member objects of `*this` the corresponding member objects of *`rhs`*, except that:

— *`sb`* and *`state`* are left unchanged;

— *`except`* is altered last by calling `exception(`*`rhs.except`*`)`.

— The contents of arrays pointed at by `pword` and `iword` are copied not the pointers themselves.[129)]

2 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.

3 Returns `*this`.

27.1.3.1.12 `basic_ios::tie` [lib.ios::tie]

```
basic_ostream<charT,baggage>* tie() const;
```

1 Returns *`tiestr`*.

```
basic_ostream<charT,baggage>* tie(basic_ostream<charT,baggage>* tiestr_arg);
```

129) This suggests an infinite amount of copying, but the implementation can keep track of the maximum element of the arrays that is non-zero.

2 Assigns *tiestr_arg* to *tiestr* and then returns the previous value stored in *tiestr*.

27.1.3.1.13 basic_ios::rdbuf [lib.ios::rdbuf]

```
basic_streambuf<charT,baggage>* rdbuf() const;
```

1 Returns *sb*.

```
basic_streambuf<charT,baggage>* rdbuf(basic_streambuf<charT,baggage>* sb_arg);
```

2 Assigns *sb_arg* to *sb*, then calls `clear()`.

Box 128

Note: need to modify so as to describe the occurence of imbueing *loc*`::codecnv into the argument stream buffer.`

3 Returns the previous value stored in *sb*.

27.1.3.1.14 basic_ios::rdstate [lib.ios::rdstate]

```
iostate rdstate() const;
```

1 Returns *state*.

27.1.3.1.15 basic_ios::clear(iostate) [lib.ios::clear.basic.ios]

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

1 Assigns *state_arg* to *state*. If *sb* is a null pointer, the function then sets `badbit` in *state*. If *state* `&` *except* is zero, returns. Otherwise, the function throws an object *fail* of class *failure*, constructed with argument values that are implementation-defined.

27.1.3.1.16 basic_ios::setstate(iostate) [lib.ios::setstate.basic.ios]

```
void setstate(iostate state_arg);
```

1 Calls `clear(`*state* `|` *state_arg*`)`.

27.1.3.1.17 basic_ios::good [lib.ios::good]

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

1 Returns `true` if *state* is zero.

27.1.3.1.18 basic_ios::eof [lib.ios::eof]

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

1 Returns `true` if `eofbit` is set in *state*.

27.1.3.1.19 basic_ios::fail [lib.ios::fail]

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

1 Returns `true` if `failbit` or `badbit` is set in *state*.[130)]

130) Checking `badbit` also for `fail()` is historical practice.

27.1.3.1.20 `basic_ios::bad` [lib.ios::bad]

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

1 Returns `true` if `badbit` is set in *`state`*.

27.1.3.1.21 `basic_ios::exceptions` [lib.ios::exceptions]

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

1 Returns *`except`*.

```
void exceptions(iostate except_arg);
```

2 Assigns *`except_arg`* to *`except`*, then calls `clear(`*`state`*`)`.

27.1.3.1.22 `basic_ios::flags` [lib.ios::flags]

```
fmtflags flags() const;
```

1 Returns *`fmtfl`*.

```
fmtflags flags(fmtflags fmtfl_arg);
```

2 Assigns *`fmtfl_arg`* to *`fmtfl`* and then returns the previous value stored in *`fmtfl`*.

27.1.3.1.23 `basic_ios::setf(fmtflags)` [lib.ios::setf]

```
fmtflags setf(fmtflags fmtfl_arg);
```

1 Sets *`fmtfl_arg`* in *`fmtfl`* and then returns the previous value stored in *`fmtfl`*.

```
fmtflags setf(fmtflags fmtfl_arg, fmtflags mask);
```

2 Clears *`mask`* in *`fmtfl`*, sets *`fmtfl_arg & mask`* in *`fmtfl`*, and then returns the previous value stored in *`fmtfl`*.

27.1.3.1.24 `basic_ios::unsetf(fmtflags)` [lib.ios::unsetf]

```
void unsetf(fmtflags mask);
```

1 Clears *`mask`* in *`fmtfl`*.

27.1.3.1.25 `basic_ios::fill` [lib.ios::fill]

```
int_type fill() const;
```

1 Returns *`fillch`*.

```
int_type fill(int_type fillch_arg);
```

2 Assigns *`fillch_arg`* to *`fillch`* and then returns the previous value stored in *`fillch`*.

27.1.3.1.26 `basic_ios::precision` [lib.ios::precision]

```
int precision() const;
```

1 Returns *`prec`*.

```
int precision(int prec_arg);
```

2 Assigns *`prec_arg`* to *`prec`* and then returns the previous value stored in *`prec`*.

27.1.3.1.27 `basic_ios::width` [lib.ios::width]

```
int width() const;
```

1 Returns *`wide`*.

```
int width(int wide_arg);
```

2 Assigns *`wide_arg`* to *`wide`* and then returns the previous value stored in *`wide`*.

27.1.3.1.28 `basic_ios::imbue` [lib.ios::imbue]

```
locale imbue(const locale loc_arg);
```

1 Imbues the `basic_ios` with the `locale` object, *`loc`*. `In case the member pointer` *`sb`* `of the basic_streambuf<charT,ICB> has already initialized, the function also imbues the object pointed to by` *`sb`*.

2 Assigns *`loc_arg`* to *`loc`* and then returns the previous value stored in *`loc`*.

27.1.3.1.29 `basic_ios::getloc` [lib.ios::getloc]

```
locale getloc() const;
```

1 Returns the classic `"C"` locale if no locale has been imbued. Otherwise, returns *`loc`*.

27.1.3.1.30 `basic_ios::xalloc` [lib.ios::xalloc]

```
static int xalloc();
```

1 Returns *`index`*`++`.

27.1.3.1.31 `basic_ios::iword` [lib.ios::iword]

```
long& iword(int idx);
```

1 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.

2 Returns *`iarray`*`[`*`idx`*`]`. After a subsequent call to `iword(int)` for the same object, the earlier return value may no longer be valid.[131)]

27.1.3.1.32 `basic_ios::pword` [lib.ios::pword]

```
void* & pword(int idx);
```

1 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.

2 Returns *`parray`*`[`*`idx`*`]`. After a subsequent call to `pword(int)` for the same object, the earlier return value may no longer be valid.

131) 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.

27.1.3.1.33 `basic_ios` constructor [lib.basic.ios.cons]

```
basic_ios();
```

1 Constructs an object of class `basic_ios`, assigning initial values to its member objects by calling `init(0)`.

27.1.3.1.34 `basic_ios::init` [lib.basic.ios::init]

```
void init(basic_streambuf<charT,baggage>* sb_arg);
```

1 The postconditions of this function are indicated in Table 100:

Table 100—`init` effects

Element	Value
sb	sb_arg
tiestr	a null pointer
state	`goodbit` if *sb_arg* is not a null pointer, otherwise `badbit`.
except	`goodbit`
fmtfl	`skipws | dec`
wide	zero
prec	6
fillch	the space character
loc	new `locale()`, which means the default value is the current global locale;[132)
iarray	a null pointer
parray	a null pointer

Box 129

Note: the default locale value shall be `global` but not transparent because the locality of the stream buffer will be unchanged between its lifetime.

27.1.3.2 Standard `basic_ios` manipulators [lib.std.ios.manip]

27.1.3.2.1 `fmtflags` manipulators [lib.fmtflags.manip]

```
template<class charT, class baggage>
    basic_boolalpha<charT,baggage>& boolalpha(basic_ios<charT,baggage>& str);
```

1 Calls *str*`.setf(basic_ios::boolalpha)` and then returns *str*.[133)

```
template<class charT, class baggage>
    basic_ios<charT,baggage>& noboolalpha(basic_ios<charT,baggage>& str);
```

2 Calls *str*`.unsetf(basic_ios::boolalpha)` and then returns *str*.

```
template<class charT, class baggage>
    basic_ios<charT,baggage>& showbase(basic_ios<charT,baggage>& str);
```

132) Usually, `locale::classic()`.

3 Calls *str*`.setf(basic_ios::showbase)` and then returns *str*.

```
    template<class charT, class baggage>
        basic_ios<charT,baggage>& noshowbase(basic_ios<charT,baggage>& str);
```

4 Calls *str*`.unsetf(basic_ios::showbase)` and then returns *str*.

27.1.3.2.2 `basic_showpoint` [lib.basic.showpoint]

```
    template<class charT, class baggage>
        basic_ios<charT,baggage>& showpoint(basic_ios<charT,baggage>& str);
```

1 Calls *str*`.setf(basic_ios::showpoint)` and then returns *str*.

```
    template<class charT, class baggage>
        basic_ios<charT,baggage>& noshowpoint(basic_ios<charT,baggage>& str);
```

2 Calls *str*`.unsetf(basic_ios::showpoint)` and then returns *str*.

```
    template<class charT, class baggage>
        basic_ios<charT,baggage>& showpos(basic_ios<charT,baggage>& str);
```

3 Calls *str*`.setf(basic_ios::showpos)` and then returns *str*.

```
    template<class charT, class baggage>
        basic_ios<charT,baggage>& noshowpos(basic_ios<charT,baggage>& str);
```

4 Calls *str*`.unsetf(basic_ios::showpos)` and then returns *str*.

```
    template<class charT, class baggage>
        basic_ios<charT,baggage>& skipws(basic_ios<charT,baggage>& str);
```

5 Calls *str*`.setf(basic_ios::skipws)` and then returns *str*.

```
    template<class charT, class baggage>
        basic_ios<charT,baggage>& noskipws(basic_ios<charT,baggage>& str);
```

6 Calls *str*`.unsetf(basic_ios::skipws)` and then returns *str*.

```
    template<class charT, class baggage>
        basic_ios<charT,baggage>& uppercase(basic_ios<charT,baggage>& str);
```

7 Calls *str*`.setf(basic_ios::uppercase)` and then returns *str*.

```
    template<class charT, class baggage>
        basic_ios<charT,baggage>& nouppercase(basic_ios<charT,baggage>& str);
```

8 Calls *str*`.unsetf(basic_ios::uppercase)` and then returns *str*.

27.1.3.2.3 `adjustfield` manipulators [lib.adjustfield.manip]

```
    template<class charT, class baggage>
        basic_ios<charT,baggage>& internal(basic_ios<charT,baggage>& str);
```

1 Calls *str*`.setf(basic_ios::internal, basic_ios::adjustfield)` and then returns *str*.

```
    template<class charT, class baggage>
        basic_ios<charT,baggage>& left(basic_ios<charT,baggage>& str);
```

2 Calls *str*`.setf(basic_ios::left, basic_ios::adjustfield)` and then returns *str*.

```
    template<class charT, class baggage>
        basic_ios<charT,baggage>& right(basic_ios<charT,baggage>& str);
```

3 Calls *`str`*`.setf(basic_ios::right, basic_ios::adjustfield)` and then returns *`str`*.

27.1.3.2.4 `basefield` manipulators [lib.basefield.manip]

```
template<class charT, class baggage>
    basic_ios<charT,baggage>& dec(basic_ios<charT,baggage>& str);
```

1 Calls *`str`*`.setf(basic_ios::dec, basic_ios::basefield)` and then returns *`str`*.[133)]

```
template<class charT, class baggage>
    basic_ios<charT,baggage>& hex(basic_ios<charT,baggage>& str);
```

2 Calls *`str`*`.setf(basic_ios::hex, basic_ios::basefield)` and then returns *`str`*.

```
template<class charT, class baggage>
    basic_ios<charT,baggage>& oct(basic_ios<charT,baggage>& str);
```

3 Calls *`str`*`.setf(basic_ios::oct, basic_ios::basefield)` and then returns *`str`*.

27.1.3.2.5 `floatfield` manipulators [lib.floatfield.manip]

```
template<class charT, class baggage>
    basic_ios<charT,baggage>& fixed(basic_ios<charT,baggage>& str);
```

1 Calls *`str`*`.setf(basic_ios::fixed, basic_ios::floatfield)` and then returns *`str`*.

```
template<class charT, class baggage>
    basic_ios<charT,baggage>& scientific(basic_ios<charT,baggage>& str);
```

2 Calls *`str`*`.setf(basic_ios::scientific, basic_ios::floatfield)` and then returns *`str`*.

27.1.4 Standard iostream objects [lib.iostream.objects]

1 The header `<iostream>` declares four objects that associate objects of class `basic_stdiobuf` with the standard C streams provided for by the functions declared in `<cstdio>` (27.5).

Box 130

Jerry Schwarz proposal: Replace the following paragraph by: The four objects are constructed and associations are established before dynamic initialization of file scope variables is begun.

The four objects are constructed, and the associations are established, the first time an object of class `basic_ios<charT,baggage>::Init` is constructed. The four objects are *not* destroyed during program execution.[134)]

Box 131

Note: Need to determine whether we treat `cin`, `cout`, `cerr`, and `clog` as wide-oriented or not. We can allow to change narrow-wide if only no I/O operations occurs. So the default is wide-oriented and easily modify narrow-oriented? -- mjv

133) The function signature `dec(basic_ios<charT,baggage>&)` 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`.

134) Constructors and destructors for static objects can access these objects to read input from `stdin` or write output to `stdout` or `stderr`.

Box 132
They can't be wide-oriented. They have the wrong type. -- jss

Box 133
Jerry Schwarz proposal: Add `wcin, wcout, werr` and `wclog.`

Box 134
Jerry Schwarz proposal: Except as noted below the state of the format state of the iostream objects after initialization is the default state established by `ios::init()`.

27.1.4.1 Object `cin` [lib.cin]

```
istream cin;
```

1 The object `cin` controls input from an unbuffered stream buffer associated with the object `stdin`, declared in `<cstdio>`.

2 After the object `cin` is initialized, `cin.tie()` returns `cout`.

27.1.4.2 Object `cout` [lib.cout]

```
ostream cout;
```

1 The object `cout` controls output to an unbuffered stream buffer associated with the object `stdout`, declared in `<cstdio>` (27.5).

27.1.4.3 Object `cerr` [lib.cerr]

```
ostream cerr;
```

1 The object `cerr` controls output to an unbuffered stream buffer associated with the object `stderr`, declared in `<cstdio>` (27.5).

2 After the object `cerr` is initialized, `cerr.flags() & unitbuf` is nonzero.

27.1.4.4 Object `clog` [lib.clog]

```
ostream clog;
```

1 The object `clog` controls output to a stream buffer associated with the object `stderr`, declared in `<cstdio>` (27.5).

Box 135
I think this is overspecification. The destination of `clog` ought to be implementation defined. --jss

27.2 Default stream buffers and streams [lib.default.iostreams]

1 Headers:

— `<streambuf>`

— `<istream>`

— `<ostream>`

2 Table 101:

Table 101—Header `<streambuf>` synopsis

Type	Name(s)	
Type:	`streamoff`	
Template class:	`basic_streambuf`	
Classes:		
`basic_streambuf<char>`	`streambuf`	`wstreambuf`
`basic_streambuf<wchar_t>`	`streampos`	`wstreampos`

3 Table 102:

Table 102—Header `<istream>` synopsis

Type	Name(s)
Template classes:	
`basic_istream`	`istreambuf_iterator`
Template operators:	
`operator!= (istreambuf_iterator)`	`operator== (istreambuf_iterator)`
Template functions:	
`basic_ws`	`value_type (istreambuf_iterator)`
`distance_type (istreambuf_iterator)`	
`iterator_category (istreambuf_iter)`	
Classes:	
`basic_istream<char>`	`istream`
`basic_istream<wchar_t>`	`wistream`
Function: `ws`	

4 Table 103:

Table 103—Header `<ostream>` synopsis

Type	Name(s)
Template classes:	
`basic_ostream`	`ostreambuf_iterator`
Template operators:	
`operator!= (ostreambuf_iterator)`	`operator== (ostreambuf_iterator)`
Template functions:	
`basic_ends`	`value_type (ostreambuf_iterator)`
`basic_flush`	`iterator_category (ostreambuf_iter)`
Classes:	
`basic_ostream<char>`	`ostream`
`basic_ostream<wchar_t>`	`wostream`
Functions: `endl`	`ends` `flush`

27.2.1 Stream buffers [lib.stream.buffers]

1 The header `<streambuf>` defines types that control input from and output to *character* sequences.

27.2.1.1 Stream buffer requirements [lib.streambuf.reqts]

1 The following templates defined in the header `<streambuf>` have already been specified in subclause 27.1.1.1:

```
    template class ios_baggage,
    template class ios_char_baggage,
    template class ios_pos_baggage;
```

2 The following defined in the header `<streambuf>` have already been specified in subclauses 27.1.1.2 and 27.1.2.1:

```
    class ios_char_baggage<char>,
    class ios_char_baggage<wchar_t>,
    class ios_pos_baggage<streampos>,
    class ios_pos_baggage<wstreampos>,
    Type streamoff,
    Type streampos,
    Type wstreamoff,
    Class wstreampos.
```

27.2.1.2 Template class `basic_streambuf<charT,baggage>` [lib.streambuf]

```
template<class charT, class baggage = ios_baggage<charT> >
class basic_streambuf {
public:
    typedef charT char_type;
    typedef baggage::char_bag::int_type int_type;
    typedef baggage::pos_bag::pos_type  pos_type;
    typedef baggage::pos_bag::off_type  off_type;
```

Box 136

In order to simplify descriptions and as a convenience for programmers.

1 Jerry Schwarz proposal:

```
    typedef basic_ios<char> ios;
```

and use of the defined type as appropriate elsewhere in the working paper.

```
    int_type  eof()     { return baggage::char_bag::eof(); }
    char_type newline() { return baggage::char_bag::newline(); }

public:
    virtual ~basic_streambuf();
    pos_type pubseekoff(off_type off,
                        basic_ios<charT,baggage>::seekdir way,
                        basic_ios<charT,baggage>::openmode which
                            = basic_ios<charT,baggage>::in
                            | basic_ios<charT,baggage>::out);
    pos_type pubseekpos(pos_type sp,
                        basic_ios<charT,baggage>::openmode which
                            = basic_ios<charT,baggage>::in
                            | basic_ios<charT,baggage>::out);
    basic_streambuf<char_type,baggage>*
             pubsetbuf(char_type* s, streamsize n);
```

Box 137

The following was not part of the proposal. Should we keep it? -- mjv

Box 138

Yes. I believe this was an inadvertant omission due to the proposal being based on a draft that did not contain `in_avail`. -- jss.

```
    int      in_avail();
    int      pubsync();
    int_type sbumpc();
    int_type sgetc();
    int      sgetn(char_type* s, streamsize n);
    int_type snextc();
    int_type sputbackc(char_type c);
```

Box 139

The following two were not part of the proposal. Should we keep them? -- mjv

Box 140

Yes. I believe this was an inadvertant omission due to the proposal being based on a draft that did not contain `sungetc` . -- jss.

```
    int      sungetc();
    int      sputc(int c);
    int_type sputn(const char_type* s, streamsize n);

protected:
    basic_streambuf();
    char_type* eback() const;
    char_type* gptr() const;
    char_type* egptr() const;
    void       gbump(int n);
    void       setg(char_type* gbeg_arg,
                    char_type* gnext_arg,
                    char_type* gend_arg);
    char_type* pbase() const;
    char_type* pptr() const;
    char_type* epptr() const;
    void       pbump(int n);
    void       setp(char_type* pbeg_arg,
                    char_type* pend_arg);
    virtual int_type overflow (int_type c = eof());
    virtual int_type pbackfail(int_type c = eof());
```

Box 141

The following was not part of the proposal. Should we keep it? -- mjv

Box 142

Yes. It is the public interface to `in_avail`. -- jss

```
    virtual int      showmany();
    virtual int_type underflow();
    virtual int_type uflow();
    virtual streamsize xsgetn(char_type* s, streamsize n);
    virtual streamsize xsputn(const char_type* s, streamsize n);
    virtual pos_type seekoff(off_type off,
            basic_ios<charT,baggage>::seekdir way,
            basic_ios<charT,baggage>::openmode which = in | out);
    virtual pos_type seekpos(pos_type sp,
            basic_ios<charT,baggage>::openmode which = in | out);
    virtual basic_streambuf<char_type,baggage>*
              setbuf(char_type* s, streamsize n);
    virtual int sync();
```

Box 143

The following two member functions are introduced to support raw byte I/O, whose behaviors are implementation-defined.

Box 144

Private discussions in Kitchener concluded that these functions cannot be implemented in any generic fashion and that their usefulness is doubtful.

2 Jerry Schwarz proposal: Remove `write_byte` and `read_byte` here and throughout the working paper.

```
    virtual streamsize write_byte(const char* buf, streamsize len);
    virtual streamsize read_byte (char* buf, streamsize len);
    private:
//  char_type* gbeg;     exposition only
//  char_type* gnext;    exposition only
//  char_type* gend;     exposition only
//  char_type* pbeg;     exposition only
//  char_type* pnext;    exposition only
//  char_type* pend;     exposition only
    };

    class streambuf : public basic_streambuf<char> {};

    class wstreambuf : public basic_streambuf<wchar_t> {};
```

3 The class template `basic_streambuf<charT,baggage>` 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.

4 The class `streambuf` is an instantiation of the template class `basic_streambuf` specialized by the type `char`.

5 The class wstreambuf is an instantiation of the template class `basic_streambuf` specialized by the type `wchar_t`.

6 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.

7 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).

8 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).

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

— `char_type*` *gbeg*, the beginning pointer for the input sequence;

— `char_type*` *gnext*, the next pointer for the input sequence;

— `char_type*` *gend*, the end pointer for the input sequence;

— `char_type*` *pbeg*, the beginning pointer for the output sequence;

— `char_type*` *pnext*, the next pointer for the output sequence;

— `char_type*` *pend*, the end pointer for the output sequence.

27.2.1.2.1 `basic_streambuf` destructor [lib.basic.streambuf.des]

```
virtual ~basic_streambuf();
```

1 Destroys an object of class `basic_streambuf`.

27.2.1.2.2 `basic_streambuf::pubseekoff` [lib.streambuf::pubseekoff]

```
pos_type pubseekoff(off_type off,
    basic_ios<charT,baggage>::seekdir way,
    basic_ios<charT,baggage>::openmode which
        = basic_ios<charT,baggage>::in | basic_ios<charT,baggage>::out);
```

1 Returns `seekoff(`*`off`*`, `*`way`*`, `*`which`*`)`.

27.2.1.2.3 `basic_streambuf::pubseekpos` [lib.streambuf::pubseekpos]

```
pos_type pubseekpos(pos_type sp,
    basic_ios<charT,baggage>::openmode which
        = basic_ios<charT,baggage>::in | basic_ios<charT,baggage>::out);
```

1 Returns `seekpos(`*`sp`*`, `*`which`*`)`.

27.2.1.2.4 `basic_streambuf::pubsetbuf` [lib.streambuf::pubsetbuf]

```
basic_streambuf<char_type,baggage>*
    pubsetbuf(char_type* s, streamsize n);
```

1 Returns `setbuf(`*`s`*`, `*`n`*`)`.

Box 145
The following was not part of the proposal. Should we keep it?

27.2.1.2.5 `basic_streambuf::in_avail` [lib.streambuf::in.avail]

```
int in_avail();
```

1 If the input sequence does not have a read position available, returns `showmany()`. Otherwise, returns *`gend`*` - `*`gnext`*.

27.2.1.2.6 `basic_streambuf::pubsync` [lib.streambuf::pubsync]

```
int pubsync();
```

1 Returns `sync()`.

27.2.1.2.7 `basic_streambuf::sbumpc` [lib.streambuf::sbumpc]

```
int_type sbumpc();
```

1 If the input sequence does not have a read position available, returns `uflow()`. Otherwise, returns `(char_type)*`*`gnext`*`++`.

27.2.1.2.8 `basic_streambuf::sgetc` [lib.streambuf::sgetc]

```
int_type sgetc();
```

1 If the input sequence does not have a read position available, returns `underflow()`. Otherwise, returns `(char_type)*`*`gnext`*.

27.2.1.2.9 `basic_streambuf::sgetn` [lib.streambuf::sgetn]

```
int sgetn(char_type* s, streamsize n);
```

1 Returns `xsgetn(`*`s`*`, `*`n`*`)`.

27.2.1.2.10 `basic_streambuf::snextc` [lib.streambuf::snextc]

```
int_type snextc();
```

1 Calls `sbumpc()` and, if that function returns `eof()`, returns `eof()`. Otherwise, returns `sgetc()`.

Box 146
Note: `baggage::eos()` ... the definition of `EOF`

27.2.1.2.11 `basic_streambuf::sputbackc` [lib.streambuf::sputbackc]

```
int_type sputbackc(char_type c);
```

1 If the input sequence does not have a putback position available, or if *`c`* `!=` *`gnext`*`[-1]`, returns `pbackfail(`*`c`*`)`. Otherwise, returns `(char_type)*--`*`gnext`*.

Box 147
The following was not part of the proposal. Should we keep it?

27.2.1.2.12 `basic_streambuf::sungetc` [lib.streambuf::sungetc]

```
int sungetc();
```

1 If the input sequence does not have a putback position available, returns `pbackfail()`. Otherwise, returns `(char_type)*--`*`gnext`*.

Box 148
The following was not part of the proposal. Should we keep it? --mjv

Box 149
Absolutely `sputc` is a fundamental member of the interface -- jss

27.2.1.2.13 `basic_streambuf::sputc` [lib.streambuf::sputc]

```
int sputc(int c);
```

1 If the output sequence does not have a write position available, returns `overflow(`*`c`*`)`. Otherwise, returns `(*`*`pnext`*`++ = `*`c`*`)`.

27.2.1.2.14 `basic_streambuf::sputn` [lib.streambuf::sputn]

```
int_type sputn(const char_type* s, streamsize n);
```

1 Returns `xsputn(`*`s`*`, `*`n`*`)`.

27.2.1.2.15 `basic_streambuf` constructor [lib.basic.streambuf.cons]

```
basic_streambuf();
```

1 Constructs an object of class `basic_streambuf<charT,baggage>` and initializes: [135)]

— all its pointer member objects to null pointers,

— the *loc* member object to the return value of `locale::global()`.

Box 150
Once the *loc* member is initialized its locale-dependent behavior does not change until the next imbueing of the locale.

27.2.1.2.16 `basic_streambuf::eback` [lib.streambuf::eback]

```
char_type* eback() const;
```

1 Returns *gbeg*.

27.2.1.2.17 `basic_streambuf::gptr` [lib.streambuf::gptr]

```
char_type* gptr() const;
```

1 Returns *gnext*.

27.2.1.2.18 `basic_streambuf::egptr` [lib.streambuf::egptr]

```
char_type* egptr() const;
```

1 Returns *gend*.

27.2.1.2.19 `basic_streambuf::gbump` [lib.streambuf::gbump]

```
void gbump(int n);
```

1 Assigns *gnext + n* to *gnext*.

27.2.1.2.20 `basic_streambuf::setg` [lib.streambuf::setg]

```
void setg(char_type* gbeg_arg,
    char_type* gnext_arg,
    char_type* gend_arg);
```

1 Assigns *gbeg_arg* to *gbeg*, *gnext_arg* to *gnext*, and *gend_arg* to *gend*.

27.2.1.2.21 `basic_streambuf::pbase` [lib.streambuf::pbase]

```
char_type* pbase() const;
```

1 Returns *pbeg*.

135) The default constructor is protected for class `basic_streambuf` to assure that only objects for classes derived from this class may be constructed.

27.2.1.2.22 `basic_streambuf::pptr` [lib.streambuf::pptr]

```
char_type* pptr() const;
```

1 Returns *pnext*.

27.2.1.2.23 `basic_streambuf::epptr` [lib.streambuf::epptr]

```
char_type* epptr() const;
```

1 Returns *pend*.

27.2.1.2.24 `basic_streambuf::pbump` [lib.streambuf::pbump]

```
void pbump(int n);
```

1 Assigns *pnext + n* to *pnext*.

27.2.1.2.25 `basic_streambuf::setp` [lib.streambuf::setp]

```
void setp(char_type* pbeg_arg, char_type* pend_arg);
```

1 Assigns *pbeg_arg* to *pbeg*, *pbeg_arg* to *pnext*, and *pend_arg* to *pend*.

27.2.1.2.26 `basic_streambuf::overflow` [lib.streambuf::overflow]

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

1 Consumes some initial subsequence of the characters of the *pending sequence*. The pending sequence is defined as the concatenation of

a) if *pbeg* is `NULL` then the empty sequence otherwise, `pnext - pbeg` characters beginning at `pbeg`.

b) if *c* `== eof()` then the empty sequence otherwise, the sequence consisting of *c*.

2 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.

3 Every overiding definition of this virtual function shall obey the following constraints:

1) The effect of consuming a character on the associated output sequence is specified[136)]

2) Let `r` be the number of characters in the pending sequence not consumed. If `r` is non-zerof then *pbeg* and *pnext* must be set so that:

pnext - pbeg == r

and the *r* characters starting at *pbeg* are the associated output stream. In case *r* is zero (all characters of the pending sequence have been consumed) then either *pbeg* is set to `NULL`, or *pbeg* and *pnext* 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 *pbeg* and *pnext* according to the above rules.

4 Returns `eof()` or throws an exception if the function fails.

5 Otherwise, returns some value other than `eof()`[137)] to indicate success.

136) That is, for each class derived from an instance of `basic_streambuf` in this clause, a specification of how consume a character effects the associated output sequence is given. There is no requirement on a user defined class.

137) Typically, *c*.

6 Default behavior: returns `eof()`.

Box 151

Should be pure virtual. -- mjv

Box 152

I disagree. Accidental attempts to create a `streambuf` are eliminated by making the constructors protected. By giving definitions to all the virtuals in `streambuf` we make it possible to derive from it by only supplying definitions for the virtuals that are needed. For example, you do not have to define `overflow` in a `streambuf` that is only intended for input. I have a similar response to all the other places where there is a box asking this question although IUm only making this response in one place. -- jss

27.2.1.2.27 `basic_streambuf::pbackfail` [lib.streambuf::pbackfail]

```
virtual int_type pbackfail(int c = eof());
```

Box 153

Check vs. lib.basic.filebuf::pbackfail

Box 154

Jerry Schwarz proposal: Replace this section by the following and make corresponding changes to descriptions of the overriding functions defined elsewhere in the working paper.

1 The public functions of `basic_streambuf` call this virtual only when `gnext` is null, `gnext==gbeg` or `*gnext!=c`. Other calls shall also satisfy that constraint.

2 The *pending sequence* is defined as for `underflow` (in 27.2.1.2.29) with the modifications that

— If `c==eof()` then the input sequence is backed up one character before the pending sequence is determined.

— If `c!=eof()` then `c` is prepended. Whether the input sequence is backed up or modified in any other way is unspecified.

3 On return, the constraints of `gnext`, `gbeg`, and `pnext` are the same as for `underflow` (in 27.2.1.2.29)

4 Returns `eof()` to indicate failure. Failure may occur because the input sequence could not be backed up, or if for some other reason the pointers could not be set consistent with the constraints.

5 Returns some value other than `eof()` to indicate success.

6 Default behavior: returns `eof()`

7 Puts back the character designated by *`c`* to the input sequence, if possible, in one of five ways:

— If *`c`* `!= eof()`, if either the input sequence has a putback position available or the function makes a putback position available, and if `(charT)`*`c`* `== (charT)`*`gnext`*`[-1]`, assigns *`gnext`* `- 1` to *`gnext`*.

Returns `(char_type)`*`c`*.

— If *`c`* `!= eof()`, if either the input sequence has a putback position available or the function makes a putback position available, and if the function is permitted to assign to the putback position, assigns *`c`* to `*--`*`gnext`*.

Returns `(char_type)`*c*.

— If *c* `!= eof()`, if no putback position is available, and if the function can put back a character directly to the associated input sequence, puts back *c* directly to the associated input sequence.

Returns `(char_type)`*c*.

— If *c* `== eof()` and if either the input sequence has a putback position available or the function makes a putback position available, assigns *gnext* `- 1` to *gnext*.

Returns `(char_type)`*c*.

— If *c* `== eof()`, if no putback position is available, if the function can put back a character directly to the associated input sequence, and if the function can determine the character *x* immediately before the current position in the associated input sequence, puts back *x* directly to the associated input sequence.

Returns a value other than `eof()`.

8 Returns `eof()` to indicate failure.

9 Default behavior: returns `eof()`.

Box 155
Should be pure virtual.

10 Notes:

11 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. How (or whether) the function makes a putback position available, puts back a character directly to the input sequence, or determines the character immediately before the current position in the associated input sequence is defined separately for each class derived from `basic_streambuf` in this clause.

Box 156
The following was not part of the proposal. Should we keep it?

27.2.1.2.28 `basic_streambuf::showmany`[138)] [lib.streambuf::showmany]

```
virtual int showmany();
```

1 Returns a count of the guaranteed number of characters that can be read from the input sequence before a call to `uflow()` or `underflow()` returns `eof()`. A positive return value of indicates that the next such call will not return `eof()`.[139)]

2 Default behavior: returns zero.

27.2.1.2.29 `basic_streambuf::underflow` [lib.streambuf::underflow]

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

1 The public members of `basic_streambuf` call this virtual only if `gnext` is null or `gnext >= gend`

138) The morphemes of `showmany` are "es-how-manyS, not "show-many".
139) The next such call might fail by throwing an exception. The intention is that the next call will return ‘‘immediately.’’

2 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.

3 The pending sequence of characters is defined as the concatenation of:

a) If *`gnext`* is non-`NULL`, then the *`gend - gnext`* characters starting at *`gnext`*, otherwise the empty sequence.

b) Some sequence (possibly empty) of characters read from the input sequence.

4 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.

5 The *backup sequence* is defined as the concatenation of:

a) If *`gbeg`* is null then empty,

b) Otherwise the *`gnext - gbeg`* characters beginning at *`gbeg`*.

6 The function sets up the *`gnext`* and *`gend`* satisfying one of:

a) If the pending sequence is non-empty, *`gend`* is non-null and *`gend - gnext`* characters starting at *`gnext`* are the characters in the pending sequence

b) If the pending sequence is empty, either *`gnext`* is null or *`gnext`* and *`gend`* are set to the same non-`NULL` pointer.

7 If *`gbeg`* and *`gnext`* 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 *`gnext - gbeg`* characters, then the *`gnext - gbeg`* characters starting at `gbeg` agree with the last *`gnext - gbeg`* characters of the backup sequence.

b) Or the *`n`* characters starting at *`gnext - n`* agree with the backup sequence (where *`n`* is the length of the backup sequence)

8 Returns `eof()` to indicate failure.

9 Default behavior: returns `eof()`.

Box 157
Should be pure virtual.

27.2.1.2.30 `basic_streambuf::uflow` [lib.streambuf::uflow]

```
virtual int_type uflow();
```

1 The constraints are the same as for `underflow` (27.2.1.2.29) 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.

2 Default behavior: Calls `underflow(eof())`. If `underflow` returns `eof()`, returns `eof()`. Otherwise, do *`gbump(-1)`* and returns `(char_type)*`*`gnext`*.

Box 158
Jerry Schwarz proposal: Returns `not_eof(c)`.

27.2.1.2.31 `basic_streambuf::xsgetn` [lib.streambuf::xsgetn]

```
virtual streamsize xsgetn(char_type* s, streamsize n);
```

1 Assigns up to *n* characters to successive elements of the array whose first element is designated by *s*. The characters assigned are read from the input sequence as if by repeated calls to `sbumpc()`. Assigning stops when either *n* characters have been assigned or a call to `sbumpc()` would return `eof()`.

2 Returns the number of characters assigned.[140)]

27.2.1.2.32 `basic_streambuf::xsputn` [lib.streambuf::xsputn]

```
virtual streamsize xsputn(const char_type* s, streamsize n);
```

1 Writes up to *n* characters to the output sequence as if by repeated calls to `sputc(c)`. The characters written are obtained from successive elements of the array whose first element is designated by *s*. Writing stops when either *n* characters have been written or a call to `sputc(c)` would return `eof()`.

2 Returns the number of characters written.

27.2.1.2.33 `basic_streambuf::seekoff` [lib.streambuf::seekoff]

```
virtual pos_type seekoff(off_type off,
    basic_ios<charT,baggage>::seekdir way,
    basic_ios<charT,baggage>::openmode which
        = basic_ios<charT,baggage>::in | basic_ios<charT,baggage>::out);
```

1 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.

2 Default behavior: returns an object of class `pos_type` that stores an invalid stream position.

Box 159
Should be pure virtual.

27.2.1.2.34 `basic_streambuf::seekpos` [lib.streambuf::seekpos]

```
virtual pos_type seekpos(pos_type sp,
    basic_ios<charT,baggage>::openmode which = in | out);
```

1 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.

2 Default behavior: returns an object of class `pos_type` that stores an invalid stream position.

Box 160
Should be pure virtual.

140) Classes derived from `basic_streambuf` can provide more efficient ways to implement `xsgetn` and `xsputn` by overriding these definitions in the base class.

27.2.1.2.35 `basic_streambuf::setbuf` [lib.streambuf::setbuf]

```
virtual basic_streambuf*
    setbuf(char_type* s, streamsize n);
```

1 Performs an operation that is defined separately for each class derived from `basic_streambuf` in this clause.

Box 161
Should be pure virtual.

2 Default behavior: returns `this`.

27.2.1.2.36 `basic_streambuf::sync` [lib.streambuf::sync]

```
virtual int sync();
```

1 Synchronizes the controlled sequences with the arrays. That is, if `pbeg` is non-null the characters between `pbeg` and `pnext` are written to the controlled sequence, and if `gnext` is non-null the characters between `gnext` and `gend` are restored to the input sequence. The pointers may then be reset as appropriate.

2 Returns -1 on failure. What constitutes failure is determined by each derived class.

3 Default behavior: returns zero.

27.2.1.2.37 `basic_streambuf::read_byte` [lib.streambuf::read.byte]

Box 162
Jerry Schwarz proposal: Delete `read_byte`. It cannot be given any generic definition.

```
streamsize read_byte(char* s, streamsize n);
```

Assigns up to *n* bytes to successive elements of an array whose first element is designated by *s*. How the characters written are obtained is the implementation-defined. Assigning stops when either *n* characters have been assigned or end-of-file occurs on the input sequence.

1 Returns the number of characters assigned.

Box 163
The reason that the behavior of `read_byte` is unspecified (or implementation-defined) is because we cannot assume the nature of the external source/sink stream. If the external source/sink stream is a byte stream, we have a chance to specify more clear description about the behavior. However, there is no insurance that the external stream be a byte sequence. In case the stream is a wide character stream. How we should convert it to a byte sequence, use ‘‘narrow’’ functions, every wide character breaks an element of bytes, or converted into a multibyte character sequence? Because even library users can customize the C++ iostream, any kind of external source/sink stream may be applied and any kind of conversion will be challenged. So, the Standard should not provide any specification about the nature of the conversion.

27.2.1.2.38 `basic_streambuf::write_byte` [lib.streambuf::write.byte]

Box 164
Jerry Schwarz proposal: Delete `write_byte.` It cannot be given any generic definition.

```
    streamsize write_byte(const char* s, streamsize n);
```

1 Writes up to *n* bytes to the output sequences. The bytes written are obtained from successive elements of the array whose first element is designated by *s*. How the bytes written are converted to the external source/sink stream is implementation-defined. Writing stops when either *n* characters have been written or a write fails.

2 Returns the number of characters written.

27.2.2 Input streams [lib.input.streams]

1 The header `<istream>` defines a type and a function signature that control input from a stream buffer.

27.2.2.1 Template class `basic_istream` [lib.istream]

```
template <class charT, class baggage = ios_baggage<charT> >
class basic_istream : virtual public basic_ios<charT,baggage> {
public:
    typedef charT char_type;
    typedef baggage::char_bag::int_type int_type;
    typedef baggage::pos_bag::pos_type  pos_type;
    typedef baggage::pos_bag::off_type  off_type;
    int_type  eof()     { return baggage::char_bag::eof(); }
    char_type newline() { return baggage::char_bag::newline(); }
private:
// for abbreviation:
    typedef basic_istream<char_type,baggage> istream_type;
    typedef basic_ios<charT,baggage> ios_type;

public:
    basic_istream(basic_streambuf<charT,baggage>* sb);
    virtual ~basic_istream();
    bool ipfx(bool noskipws = 0);
    void isfx();

    istream_type& operator>>(istream_type& (*pf)(istream_type&))
    istream_type& operator>>(ios_type& (*pf)(ios_type&))
    istream_type& operator>>(char_type* s);
```

Box 165
The following two were not part of the proposal. Should we keep them? -- mjv

Box 166
They cannot be kept because their types are not legal. You can't make an arbitrary `char_type` signed or unsigned. However they are very much a part of the current `iostream` interface, and I suspect are used extensively, so they must be supported. The only way I can see to deal with this is to make `basic_iostream<char>` a specialization. I'm not making this an "Jerry Schwarz proposal" because I am not convinced that this is the best solution. -- jss

```
    istream_type& operator>>(unsigned char_type* s)
    istream_type& operator>>(signed char_type* s);
    istream_type& operator>>(char_type& c);
```

Box 167
The following two were not part of the proposal. Should we keep them? -- mjv

Box 168
The same considerations as applied to `operator>>` apply here. -- jss

```
istream_type& operator>>(unsigned char_type& c)
istream_type& operator>>(signed char_type& c)
istream_type& operator>>(bool& n);
istream_type& operator>>(short& n);
istream_type& operator>>(unsigned short& n);
istream_type& operator>>(int& n);
istream_type& operator>>(unsigned int& n);
istream_type& operator>>(long& n);
istream_type& operator>>(unsigned long& n);
istream_type& operator>>(float& f);
istream_type& operator>>(double& f);
istream_type& operator>>(long double& f);
istream_type& operator>>(void*& p);
istream_type& operator>>(basic_streambuf<char_type,baggage>& sb);

int_type get();
istream_type& get(char_type* s, streamsize n,
                  char_type delim = newline());
```

Box 169
The following two were not part of the proposal. Should we keep them?

Box 170
The same considerations as applied to `operator>>` apply here. -- jss

```
istream_type& get(unsigned char_type* s, streamsize n,
                  char_type delim = newline())
```

Box 171
The following two were not part of the proposal. Should we keep them?

```
istream_type& get(signed char_type* s, streamsize n,
                  char_type delim = newline())
istream_type& get(char_type& c);
istream_type& get(unsigned char_type& c);
istream_type& get(signed char_type& c);
istream_type& get(basic_streambuf<char_type,baggage>& sb,
                  char_type delim = newline());
istream_type& getline(char_type* s, streamsize n,
                      char_type delim = newline());
```

Box 172
The following two were not part of the proposal. Should we keep them?

Box 173
The same considerations as applied to `operator>>` apply here. -- jss

```
    istream_type& getline(unsigned char_type* s, streamsize n,
                          char_type delim = newline())
    istream_type& getline(signed char_type* s, streamsize n,
                          char_type delim = newline())
    istream_type& ignore(streamsize n = 1, int_type delim = eof());
    istream_type& read(char_type* s, streamsize n);
```

Box 174
The following two were not part of the proposal. Should we keep them?

Box 175
The same considerations as applied to `operator>>` apply here. -- jss

```
    istream_type& read(unsigned char_type* s, streamsize n)
    istream_type& read(signed char_type* s, streamsize n)
```

Box 176
A new member function for supporting the raw byte I/O in which it performs `rdbuf().read_byte(s,n)`

```
    istream_type& read_byte(char* s, streamsize n);
```

Box 177
The following was not part of the proposal. Should we keep it? -- mjv

Box 178
Yes, it is the interface to `in_avail` **-- jss**

```
    int readsome(char_type* s, int n);
    int peek();
    istream_type& putback(char_type c);
    istream_type& unget();
    streamsize gcount() const;
    int sync();
private:
//  streamsize chcount;  exposition only
};
```

Box 179

We talked about this in Kitchener and it seems that deriving `istream` from an instance of `basic_istream<char>` doesn't work unless it duplicates all the members with return type modified to `istream` rather than `basic_istream<char>`. Rather than do that we could "bite the bullet" and make them typedefs. The consequence of that any existing code that declares them to be (incomplete) classes without including `<iostream.h>` will break. Both of these are unpleasant. But some change is required, so as a placeholder Jerry Schwarz proposal:

```
typedef basic_istream<char> istream;
typedef basic_istream<wchar_t> wistream;
```

```
class istream : public basic_istream<char> {};

class wistream : public basic_istream<wchar_t> {};
```

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.

Box 180

[To Be Filled]

2 For the sake of exposition, the data maintained by an object of class `basic_istream` is presented here as:

— `int` *`chcount`*, stores the number of characters extracted by the last unformatted input member function called for the object.

3 Two groups of member function signatures share common properties: the *formatted input functions* (or *extractors*) and the *unformatted input functions.* Both groups of input functions are described as if they obtain (or *extract*) input *characters* by calling *`sb`*`.sbumpc()` or *`sb`*`.sgetc()`. They may use other public members of `istream` except that they do not invoke any tuals members of *`sb`* except `uflow`.

4 If *`sb`*`.sbumpc()` or *`sb`*`.sgetc()` returns `eof()`, then the input function, except as explicitly noted otherwise, completes its actions and does `setstate(eofbit)` before returning.

5 If one of these called functions throws an exception, then unless explicitly noted otherwise the input function calls `setstate(badbit)` and if `badbit` is on in `sb.exception()` rethrows the exception without completing its actions.

27.2.2.1.1 `basic_istream` constructor [lib.basic.istream.cons]

```
basic_istream(basic_streambuf<charT,baggage>* sb);
```

1 Constructs an object of class `basic_istream`, assigning initial values to the base class by calling `basic_ios::init(`*`sb`*`)`, then assigning zero to *`chcount`*.

27.2.2.1.2 `basic_istream` destructor [lib.basic.istream.des]

```
virtual ~basic_istream();
```

1 Destroys an object of class `basic_istream`. Does not perform any operations of *`sb`*.

27.2.2.1.3 `basic_istream::ipfx` [lib.istream::ipfx]

```
    bool ipfx(bool noskipws = 0);
```

1 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.[141)] 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.

2 To decide if the character *`c`* is a whitespace character, the function performs as if it executes the following code fragment:

```
  locale::ctype<charT> ctype = getloc().use<locale::ctype<charT> >();
  if (baggage::char_bag::is_whitespace (c, ctype)!=0)
 // c is a whitespace character.
```

3 If, after any preparation is completed, `good()` is `true`, returns `true`. Otherwise, it calls `setstate(failbit)` and returns `false`.[142)]

141) The call `tie()->flush()` does not necessarily occur if the function can determine that no synchronization is necessary.
142) The function signatures `ipfx(int)` and `isfx()` can also perform additional implementation-dependent operations.

Box 181

The proposal will say as follows:

The function `basic_istream<charT,baggage>::ipfx()` uses the function, `bool baggage::char_bag::is_whitespace(charT, const locale*)` in the baggage structure to determine whether the next input character is whitespace or not.

A typical implementation of the `ipfx` function may be as follows:

```
template <class charT, class baggage = ios_baggage<charT> >
int basic_istream<charT,baggage>::ipfx() {
    ...
// skipping whitespace according to a constraint function,
// is_whitespace
    intT c;
    typedef locale::ctype<charT> ctype_type;
    ctype_type& ctype = getloc().use<ctype_type>();
    while ((c = rdbuf()->snextc()) != eof()) {
    if (!baggage::char_bag::is_whitespace (c,ctype)==0) {
        rdbuf()->sputbackc (c);
        break;
    }
    }
    ...
 }
```

In case we use `ios_baggage<char>` or `ios_baggage<wchar_t>`, the behavior of the constraint function `baggage::char_bag::is_whitespace()` is as if it invokes:

```
locale::ctype<charT>& ctype = getloc().use<locale::ctype<charT> >();
ctype.is(locale::ctype<charT>::SPACE, c);
```

otherwise, the behavior of the function `baggage::char_bag::is_whitespace()` is unspecified.

Those who want to use locale-independent whitespace predicate can specify their definition of `is_whitespace` in their new `ios_char_baggage` as follows:

```
struct my_baggage : public ios_baggage<char> {
    typedef my_char_baggage char_bag;
};

struct my_char_baggage : public ios_char_baggage<char> {
    static bool is_whitespace (char c, const locale::ctype<charT>& ctype) {
    ....(my own implementation)...
    }
};
```

27.2.2.1.4 basic_istream::isfx [lib.istream::isfx]

```
void isfx();
```

1 Returns.

27.2.2.1.5 `basic_istream::sync` [lib.istream::sync]

```
int sync();
```

1 If `rdbuf()` is a null pointer, returns `eof()`. Otherwise, calls `rdbuf()->pubsync()` and, if that function returns `eof()`, calls `setstate(badbit)` and returns `eof()`. Otherwise, returns zero.

27.2.2.2 Formatted input functions [lib.istream.formatted]

27.2.2.2.1 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.

Box 182

The numeric conversion behaviors of the following extractors are locale-dependent.

```
istream_type::operator>>(short& val);
istream_type::operator>>(unsigned short& val);
istream_type::operator>>(int& val);
istream_type::operator>>(unsigned int& val);
istream_type::operator>>(long& val);
istream_type::operator>>(unsigned long& val);
istream_type::operator>>(float& val);
istream_type::operator>>(double& val);
istream_type::operator>>(long double& val);
```

As in the case of the inserters, these extractors depend on the Nathan Myers's `locale::num_get<>` object to perform parsing the input stream data. The conversion occurs as if it performed the following code fragment:

```
HOLDTYPE tmp;
locale::num_get<charT>& fmt = loc.use< locale::num_get<charT> >();
fmt.get (iter, *this, loc, tmp);
if ((val = (TYPE)tmp) != tmp)
// set fail bit...
```

In the above fragment, `loc` stands for the private member of the `basic_ios` class, `TYPE` stands for the type of the argument of the extractor, and `HOLDTYPE` is as follows;

— for `short`, `int` and `long`, `HOLDTYPE` is `long`;

— for `unsigned short`, `unsigned int` and `unsigned long`, `HOLDTYPE` is `unsigned long`.

— for `float`, `double`, `HOLDTYPE is double.`

— for `long double`, `HOLDTYPE` is `long double`.

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.

3 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.

4 The behavior of such functions is described in terms of the conversion specification for an equivalent call to the function `fscanf()`[143] operating with the global locale set to *`loc`*, with the following alterations:

— The formatted input function extracts characters from a stream buffer, rather than reading them from an input file.[144]

— If `flags() & skipws` is zero, the function does not skip any leading white space. In that case, if the next input character is white space, the scan fails.

— If the converted data value cannot be represented as a value of the specified scalar data type, a scan failure occurs.

Box 183

Can the current `locale::num_put/num_get` facet handle `basefield` specification? Needs more discussion.

5 If the scan fails for any reason, the formatted input function calls `setstate(failbit)`.

6 For conversion to an integral type other than a character type, the function determines the integral conversion specifier as indicated in Table 104:

Table 104—Integer conversions

State	`stdio` equivalent
`(flags() & basefield) == oct`	`%o`

`(flags() & basefield) == hex`
`(flags() & uppercase) != 0`
	`%x`
	`%X`

`(flags() & basefield) == 0`
 	
`Otherwise,`
	`%i`

`signed` integral type	`%d`
`unsigned` integral type	`%u`

Box 184

Is this table clear with regards to f5%x **vs.** `%X` **? -- jss**

27.2.2.2.2 `basic_istream::operator>>` [lib.istream::extractors]

```
    istream_type& operator>>(istream_type& (*pf)(istream_type&))
```

1 Returns `(*`*`pf`*`)(*this)`.[145]

143) The signature `fscanf(FILE*, const char*, ...)` is declared in `<cstdio>` (27.5)
144) The stream buffer can, of course, be associated with an input file, but it need not be.
145) See, for example, the function signature `ws(basic_istream&)`.

```
    istream_type& operator>>(ios_type& (*pf)(ios_type&))
```

2 Calls `(*pf)(*this)`, then returns `*this`.[146)]

```
    istream_type& operator>>(char_type* s);
```

3 Extracts characters and stores them into successive locations of an array whose first element is designated by *s*. If `width()` is greater than zero, the maximum number of characters stored *n* is `width()`; otherwise it is `INT_MAX`.[147)]

4 Characters are extracted and stored until any of the following occurs:

— `n - 1` characters are stored;

— end-of-file occurs on the input sequence;

— `baggage::char_bag::is_whitespace(c,ctype)` is nonzero for the next available input character *c*. In the above code fragment, the argument *ctype* is acquired by `getloc().use<locale::ctype<charT> >()`.

5 If the function stores no characters, it calls `setstate(failbit)`. In any case, it then stores a null character into the next successive location of the array and calls `width(0)`.

6 Returns `*this`.

Box 185
The following two were not part of the proposal. Should we keep them?

```
    istream_type& operator>>(unsigned char_type* s)
```

7 Returns `operator>>((char_type*)s)`.

```
    istream_type& operator>>(signed char_type* s);
```

8 Returns `operator>>((char_type*)s)`.

```
    istream_type& operator>>(char_type& c);
```

9 Extracts a character, if one is available, and stores it in *c*. Otherwise, the function calls `setstate(failbit)`.

10 Returns `*this`.

Box 186
The following two were not part of the proposal. Should we keep them?

```
    istream_type& operator>>(unsigned char_type& c)
```

11 Returns `operator>>((char&)c)`.

```
    istream_type& operator>>(signed char_type& c)
```

12 Returns `operator>>((char&)c)`.

```
    istream_type& operator>>(bool& n);
```

146) See, for example, the function signature `dec(basic_ios<charT,baggage>&)`.
147) `MAX_INT` is defined in `<climits>` (27.5).

13 Converts a signed short integer, if one is available, and stores it in *x*. Behaves as if:

```
if (flags() & ios::boolalpha) {
  getloc().extract(*this, n);
} else {
  int x;
  *this >> x;
  if (x == 0)
    n = false;
  else if (x == 1)
    n = true;
  else
    ; // indicate failure
}
return *this;
```

Box 187

Locale extraction (`getloc().extract()`) of the string is something like:

```
istream i;
string bool_false = ...; // locale dependent
string bool_true  = ...;
string s;
i >> s;
if (s == bool_false)
  n = false;
else if (s == bool_true)
  n = true;
else
  ; // indicate failure
```

The strings for the default locale are `"false"` and `"true"`.

14 Returns `*this`.

```
istream_type& operator>>(short& n);
```

15 Converts a signed short integer, if one is available, and stores it in *n*.

16 Returns `*this`.

```
istream_type& operator>>(unsigned short& n);
```

17 Converts an unsigned short integer, if one is available, and stores it in *n*.

18 Returns `*this`.

```
istream_type& operator>>(int& n);
```

19 Converts a signed integer, if one is available, and stores it in *n*.

20 Returns `*this`.

```
istream_type& operator>>(unsigned int& n);
```

21 Converts an unsigned integer, if one is available, and stores it in *n*.

22 Returns `*this`.

```
istream_type& operator>>(long& n);
```

23 Converts a signed long integer, if one is available, and stores it in *n*.

24 Returns `*this`.

```
    istream_type& operator>>(unsigned long& n);
```

25 Converts an unsigned long integer, if one is available, and stores it in *n*.

26 Returns `*this`.

```
    istream_type& operator>>(float& f);
```

27 Converts a `float`, if one is available, and stores it in *f*.

28 Returns `*this`.

```
    istream_type& operator>>(double& f);
```

29 Converts a `double`, if one is available, and stores it in *f*.

30 Returns `*this`.

```
    istream_type& operator>>(long double& f);
```

31 Converts a `long double`, if one is available, and stores it in *f*.

32 Returns `*this`.

```
    istream_type& operator>>(void*& p);
```

33 Converts a pointer to `void`, if one is available, and stores it in *p*.

34 Returns `*this`.

```
    istream_type& operator>>(basic_streambuf<charT,baggage>& sb);
```

35 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

36 If the function inserts no characters, it calls `setstate(failbit)`. If failure was due to catching an exception thrown while extracting characters from *sb* and `failbit` is on in `except` then the caught exception is rethrown.

37 Returns `*this`.

27.2.2.3 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.

27.2.2.3.1 `basic_istream::get` [lib.istream::get]

```
    int get();
```

1 Extracts a character *c*, if one is available. The function then returns `(unsigned char)`*c*. Otherwise, the function calls `setstate(failbit)` and then returns `eof()`.

```
    istream_type& get(char_type*  s, streamsize n,
                      char_type delim = newline());
```

2 Extracts characters and stores them into successive locations of an array whose first element is designated by *s*. 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).

3 If the function stores no characters, it calls `setstate(failbit)`. In any case, it then stores a null character into the next successive location of the array.

4 Returns `*this`.

Box 188
The following two were not part of the proposal. Should we keep them?

```
    istream_type& get(unsigned char_type* s, streamsize n,
                      char_type delim = newline())
```

5 Returns `get((char_type*)`*s*`,` *n*`,` *delim*`)`.

```
    istream_type& get(signed char_type* s, streamsize n,
                      char_type delim = newline())
```

6 Returns `get((char_type*)`*s*`,` *n*`,` *delim*`)`.

```
    istream_type& get(char_type& c);
```

7 Extracts a character, if one is available, and assigns it to *c*. Otherwise, the function calls `setstate(failbit)`.

8 Returns `*this`.

```
    istream_type& get(unsigned char_type& c);
```

9 Returns `get((char&)`*c*`)`.

Box 189
The following two were not part of the proposal. Should we keep them?

```
    istream_type& get(signed char_type& c);
```

10 Returns `basic_istream::get((char&)`*c*`)`.

```
    istream_type& get(basic_streambuf<char_type,baggage>& sb,
                      char_type delim = newline());
```

11 Extracts characters 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);

— *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).

12 If the function inserts no characters, it calls `setstate(failbit)`.

13 Returns `*this`.

27.2.2.3.2 `basic_istream::getline` [lib.istream::getline]

```
istream_type& getline(char_type* s, streamsize n,
                      char_type delim = newline());
```

1 Extracts characters and stores them into successive locations of an array whose first element is designated by *s*. 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);[148)]

3) *n* - 1 characters are stored (in which case the function calls `setstate(failbit)`).

2 These conditions are tested in the order shown.[149)]

3 If the function extracts no characters, it calls `setstate(failbit)`.[150)]

4 In any case, it then stores a null character into the next successive location of the array.

5 Returns `*this`.

6 Example:

```
#include <iostream>
using std;

const int line_buffer_size = 100;

int main()
{
  char buffer[line_buffer_size];
  int line_number = 0;

  while (cin.getline(buffer, line_buffer_size) || cin.gcount()) {
    int count = cin.gcount();
    if (cin.eof())
      cout << "Partial final line";   // cin.fail() is false
    else if (cin.fail()) {
      cout << "Partial long line";
      cin.clear(cin.rdstate() & ~ios::failbit);
    } else {
      count--;        // Don’t include ’\n’ in count
      cout << "Line " << ++line_number;
    }
    cout << " (" << count << " chars): " << buffer << endl;
  }
}
```

Box 190
The following two were not part of the proposal. Should we keep them?

148) Since the final input character is ‘‘extracted,’’ it is counted in the `gcount()`, even though it is not stored.
149) This allows an input line which exactly fills the buffer, without setting `failbit`. This is different behavior than the historical AT&T implementation.
150) This implies an empty input line will not cause `failbit` to be set.

```
istream_type& getline(unsigned char_type* s, streamsize n,
                      char_type delim = newline())
```

7 Returns `getline((char_type*)s, n, delim)`.

```
istream_type& getline(signed char_type* s, streamsize n,
                      char_type delim = newline())
```

8 Returns `getline((char_type*)s, n, delim)`.

27.2.2.3.3 `basic_istream::ignore` [lib.istream::ignore]

```
istream_type& ignore(int n = 1, int_type delim = eof());
```

1 Extracts characters and discards them. Characters are extracted until any of the following occurs:

— if *n* `!= INT_MAX`, *n* characters are extracted

— end-of-file occurs on the input sequence (in which case the function calls `setstate(eofbit)`);

— *c* `==` *delim* for the next available input character *c* (in which case *c* is extracted).[151)]

2 The last condition will never occur if *delim* `== eof()`.

3 Returns `*this`.

27.2.2.3.4 `basic_istream::read` [lib.istream::read]

```
istream_type& read(char_type* s, streamsize n);
```

1 Extracts characters and stores them into successive locations of an array whose first element is designated by *s*. 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)`).

2 Returns `*this`.

Box 191
The following two were not part of the proposal. Should we keep them?

```
istream_type& read(unsigned char_type* s, streamsize n)
```

3 Returns `read((char_type*)s, n)`.

```
istream_type& read(signed char_type* s, streamsize n)
```

4 Returns `read((char_type*)s, n)`.

27.2.2.3.5 `basic_istream::read_byte` [lib.istream::read.byte]

```
istream_type<charT,baggage>& read_byte(char* s, streamsize n);
```

1 Extracts bytes by invoking:

```
rdbuf()->read_byte(s,n);
```

151) The macro `INT_MAX` is defined in `<climits>`.

2 In case end-of-file occurs on the input character sequence, the function calls `setstate(failbit)`.

3 Returns `*this`.

Box 192
The following was not part of the proposal. Should we keep it?

27.2.2.3.6 `basic_istream::readsome` [lib.istream::readsome]

```
int readsome(char_type* s, int n);
```

1 Extracts characters and stores them into successive locations of an array whose first element is designated by *s*. The function first determines *navail*, the value returned by calling `in_avail()`. If *navail* is `1`, the function calls `setstate(eofbit)` and returns zero.

2 Otherwise, the function determines the number of characters to extract *m* as the smaller of *n* and *navail*, and returns `read(`*s*`, `*m*`)`.

27.2.2.3.7 `basic_istream::peek` [lib.istream::peek]

```
int peek();
```

1 Returns `eof()` if `good()` is false. Otherwise, returns `rdbuf()->sgetc()`.

27.2.2.3.8 `basic_istream::putback` [lib.istream::putback]

```
istream_type& putback(char_type c);
```

1 Calls `rdbuf->sputbackc(`*c*`)`. If that function returns `eof()`, calls `setstate(badbit)`.

2 Returns `*this`.

27.2.2.3.9 `basic_istream::unget` [lib.istream::unget]

```
istream_type& unget();
```

1 Calls `rdbuf->sungetc()`. If that function returns `eof()`, calls `setstate(badbit)`.

2 Returns `*this`.

27.2.2.3.10 `basic_istream::gcount` [lib.istream::gcount]

```
streamsize gcount() const;
```

1 Returns *chcount*.

27.2.2.4 Standard `basic_istream` manipulators [lib.basic.istream.manip]

```
template<class charT, class baggage>
    basic_istream<charT,baggage>& ws(basic_istream<charT,baggage>& is);
```

1 Saves a copy of *is.fmtflags*, then clears *is*`.skipws` in *is.fmtflags*. Then calls *is*`.ipfx()` and *is*`.isfx()`, and restores *is.fmtflags* to its saved value.[152)]

2 Returns *is*.

152) The effect of `cin >> ws` is to skip any white space in the input sequence controlled by `cin`.

27.2.3 Template class `istreambuf_iterator` [lib.istreambuf.iterator]

Box 193

As part of the original proposals, the class `istreambuf_iterator` is defined in the header `<istream>`.

It really should be defined as part of Clause 24.4.

```
    template<class charT, class baggage = ios_baggage<charT> >
    class istreambuf_iterator {
    public:
        typedef charT char_type;
        typedef baggage baggage_type;
        typedef baggage::int_type int_type;
        typedef basic_streambuf<charT,baggage> streambuf;
        typedef basic_istream<charT,baggage> istream;

        class proxy {
            charT keep_;
            streambuf* sbuf_;
            proxy (charT c, streambuf* sbuf)
              : keep_(c), sbuf_(sbuf) {}
        public:
            charT operator*() { return keep_; }
            friend class istreambuf_iterator;
        };

    public:
        istreambuf_iterator();
        istreambuf_iterator(istream& s);
        istreambuf_iterator(streambuf* s);
        istreambuf_iterator(const proxy& p);
        charT operator*();
        istreambuf_iterator<charT,baggage>& operator++();
        proxy operator++(int);
        bool equal(istreambuf_iterator& b);
    private:
//      streambuf* sbuf_;          exposition only
    };
```

1 The template class `istreambuf_iterator` reads successive *characters* from the streambuf for which it was constructed.

2 After it is constructed, and every time `operator++` is used, the iterator reads and stores a value of *character*. If the end of stream is reached (streambuf::sgetc() returns `baggage::char_bag::eof()`), the iterator becomes equal to the *end of stream* iterator value. The default constructor `istreambuf_iterator()` and the constructor `istreambuf_iterator(0)` always construct an end of stream iterator object, which is the only legitimate iterator to be used for the end condition.

3 The result of `operator*()` on an end of stream is undefined. For any other iterator value a `const char_type&` is returned. It is impossible to store things into input iterators.

4 Note that in the input iterators, ++ operators are not *equality preserving*, that is, `i == j` does not guarantee at all that `++i == ++j`. Every time ++ is used a new value is used.

5 The practical consequence of this fact is that an `istreambuf_iterator` object can be used only for *one-pass algorithms*, which actually makes perfect sense, since for multi-pass algorithms it is always more appropriate to use in-memory data structures. Two end of stream iterators are always equal. An end of stream iterator is not equal to a non-end of stream iterator. Two non-end of stream iterators are equal when

they are constructed from the same stream.

27.2.3.1 Template class `istreambuf_iterator::proxy` [lib.istreambuf.iterator::proxy]

```
template <class charT, class baggage = ios_baggage<charT> >
class istream_iterator::proxy {
    charT keep_;
    streambuf* sbuf_;
    proxy(charT c, streambuf* sbuf) :
        keep_(c), sbuf_(sbuf) {}
public:
    charT operator*() { return keep_; }
    friend class istreambuf_iterator;
};
```

1 Class `istream_iterator<charT,baggage>::proxy` provides a temporal placeholder as the return value of the post-increment operator (`operator++`). It keeps the character pointed to by the previous value of the iterator for some possible future access to get the character.

27.2.3.2 `istreambuf_iterator` constructors [lib.istreambuf.iterator.cons]

```
istreambuf_iterator();
```

1 Constructs the end-of-stream iterator.

```
istreambuf_iterator(basic_istream<charT,baggage>& s);
```

2 Constructs the `istream_iterator` pointing to the `basic_streambuf` object `*(s.rdbuf())`.

```
istreambuf_iterator(const proxy& p);
```

3 Constructs the `istreambuf_iterator` pointing to the `basic_streambuf` object related to the `proxy` object *p*.

27.2.3.3 `istreambuf_iterator::operator*` [lib.istreambuf.iterator::op*]

```
charT operator*()
```

1 Extract one character pointed to by the `streambuf *sbuf_`.

27.2.3.4 `istreambuf_iterator::operator++` [lib.istreambuf.iterator::op++]

```
istreambuf_iterator<charT,baggage>&
    istreambuf_iterator<charT,baggage>::operator++();
```

1 Advances the iterator and returns the result

```
proxy istreambuf_iterator<charT,baggage>::operator++(int);
```

2 Advances the iterator and returns the `proxy` object keeping the character pointed to by the previous iterator.

27.2.3.5 `istreambuf_iterator::equal` [lib.istreambuf.iterator::equal]

```
bool equal(istreambuf_iterator<charT,baggage>& b);
```

1 Returns `true` if the iterators are equal. Equality is defined as follows:

— If both *a* and *b* are end-of-stream iterators, `a == b`.

— If either *a* or *b* is an end-of-stream iterator, if the other points end-of-file, `a == b`, otherwise `a != b`.

— If both *a* and *b* are not end-of-stream, the two `streambuf` pointed to by the both iterators are compared.

27.2.3.6 `iterator_category` [lib.iterator.category.i]

```
input_iterator iterator_category(const istreambuf_iterator& s);
```

1 Returns the category of the iterator *s*.

27.2.3.7 `operator==` [lib.istreambuf.iterator::op==]

```
template <class charT, class baggage = ios_baggage<charT> >
bool operator==(istreambuf_iterator<charT, baggage>& a,
                istreambuf_iterator<charT, baggage>& b);
```

1 Returns `a.equal(b)`.

27.2.3.8 `operator!=` [lib.istreambuf.iterator::op!=]

```
template <class charT, class baggage = ios_baggage<charT> >
bool operator!=(istreambuf_iterator<charT, baggage>& a,
               istreambuf_iterator<charT, baggage>& b);
```

1 Returns `!a.equal(b)`.

27.2.4 Output streams [lib.output.streams]

1 The header `<ostream>` defines a type and several function signatures that control output to a stream buffer.

27.2.4.1 Template class `basic_ostream` [lib.ostream]

```
template <class charT, class baggage = ioc_baggage<charT> >
class basic_ostream : virtual public basic_ios<charT,baggage> {
public:
    typedef charT char_type;
    typedef baggage::char_bag::int_type int_type;
    typedef baggage::pos_bag::pos_type  pos_type;
    typedef baggage::pos_bag::off_type  off_type;
    int_type  eof()     { return baggage::char_bag::eof(); }
    char_type newline() { return baggage::char_bag::newline(); }
private:
    typedef basic_ostream<charT,baggage> ostream_type;
public:
    basic_ostream(basic_streambuf<char_type,baggage>* sb);
    virtual ~basic_ostream();
    bool opfx();
    void osfx();

    ostream_type& operator<<(ostream_type& (*pf)(ostream_type&));
    ostream_type& operator<<(ios_type& (*pf)(ios_type&));
    ostream_type& operator<<(const char_type* s);
    ostream_type& operator<<(char_type c);
```

Box 194

The following two were not part of the proposal. Should we keep them?

```
    ostream_type& operator<<(unsigned char_type c);
    ostream_type& operator<<(signed char_type c);
    ostream_type& operator<<(bool n);
    ostream_type& operator<<(short n);
    ostream_type& operator<<(unsigned short n);
    ostream_type& operator<<(int n);
    ostream_type& operator<<(unsigned int n);
    ostream_type& operator<<(long n);
    ostream_type& operator<<(unsigned long n);
    ostream_type& operator<<(float f);
    ostream_type& operator<<(double f);
    ostream_type& operator<<(long double f);
    ostream_type& operator<<(void* p);
    ostream_type& operator<<(basic_streambuf<char_type,baggage>& sb);

    int put(char_type c);
    ostream_type& write(const char_type* s, streamsize n);
```

Box 195
The following two were not part of the proposal. Should we keep them?

```
    ostream_type& write(const unsigned char_type* s, streamsize n);
    ostream_type& write(const signed char_type* s, streamsize n);
    ostream_type& write_byte (const char* s, streamsize n);
    ostream_type& flush();
};

    class ostream : public basic_ostream<char> {};

    class wostream : public basic_ostream<wchar_t> {};
```

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.

Box 196
[To Be Filled]

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 `sb.sputc(int)`. They may use other public members of `basic_ostream` except that they do not invoke any tuals members of *sb* except `overflow`. If the called function throws an exception, the output function calls `setstate(badbit)` and if `badbit` is on in `except` rethrows the exception.

27.2.4.1.1 basic_ostream constructor [lib.basic.ostream.sb.cons]

```
    basic_ostream(basic_streambuf<charT,baggage>* sb);
```

1 Constructs an object of class `basic_ostream`, assigning initial values to the base class by calling `basic_ios<charT,baggage>::init(sb)`.

27.2.4.1.2 `basic_ostream` destructor [lib.basic.ostream.des]

```
virtual ~basic_ostream();
```

1 Destroys an object of class `basic_ostream`. Does not perform any operations on `sb`.

27.2.4.1.3 `basic_ostream::opfx` [lib.ostream::opfx]

```
bool opfx();
```

1 If `good()` is nonzero, prepares for formatted or unformatted output. If `tie()` is not a null pointer, calls `tie()->flush()`.[153)]

Box 197
Note: Need to append the `locale` dependency on appropriate extractors.

2 Returns `good()`.[154)]

27.2.4.1.4 `basic_ostream::osfx` [lib.ostream::osfx]

```
void osfx();
```

1 If `flags() & unitbuf` is nonzero, calls `flush()`.

27.2.4.1.5 `basic_ostream::flush` [lib.ostream::flush]

```
basic_ostream& flush();
```

1 If `rdbuf()` is not a null pointer, calls `rdbuf()->pubsync()`. If that function returns `eof()`, calls `setstate(badbit)`.

2 Returns `*this`.

27.2.4.2 Formatted output functions [lib.ostream.formatted]

27.2.4.2.1 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.

153) The call `tie()->flush()` does not necessarily occur if the function can determine that no synchronization is necessary.
154) The function signatures `opfx()` and `osfx()` can also perform additional implementation-dependent operations.

Box 198

The following specification description has not reflected locale-dependency of the behavior of the integral type/float type inserters.

The numeric conversion behaviors of the following inserters are locale-dependent:

```
basic_ostream<charT,baggage>::operator<<(short val);
basic_ostream<charT,baggage>::operator<<(unsigned short val);
basic_ostream<charT,baggage>::operator<<(int val);
basic_ostream<charT,baggage>::operator<<(unsigned int val);
basic_ostream<charT,baggage>::operator<<(long val);
basic_ostream<charT,baggage>::operator<<(unsigned long val);
basic_ostream<charT,baggage>::operator<<(float val);
basic_ostream<charT,baggage>::operator<<(double val);
basic_ostream<charT,baggage>::operator<<(long double val);
```

According to the Nathan Myers's locale object draft [X3J16/94-0064R1,WG21/N0451R1], the class `locale::num_get<>` and `locale::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:

```
locale::num_put<charT>& fmt = loc.use< locale::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.

2 Some formatted output functions endeavor to generate the requested output by converting a value from some scalar or NTBS type to text form and inserting the converted text in the output sequence.

Box 199

Needs work: NTBS.

The behavior of such functions is described in terms of the conversion specification for an equivalent call to the function `fprintf`,[155] operating with the global locale set to *`loc`*, with the following alterations:

— The formatted output function inserts *characters* in a stream buffer, rather than writing them to an output file.[156]

— 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).

3 If the operation fails for any reason, the formatted output function calls `setstate(badbit)`.

4 For conversion from an integral type other than a character type, the function determines the integral conversion specifier as indicated in Table 105:

155) The signature `fprintf(FILE*, const char_type*, ...)` is declared in `<cstdio>` (27.5).
156) The stream buffer can, of course, be associated with an output file, but it need not be.

Table 105—Integer conversions

State	`stdio` equivalent
`(flags() & basefield) == oct`	`%o`
`(flags() & basefield) == hex`	`%x`
`(flags() & uppercase) != 0`	`%X`
`Otherwise,`	
`signed` integral type	`%d`
`unsigned` integral type	`%u`

Box 200

Is this table clear with regards to f5%x **vs. f5%X** ? -- jss

5 For conversion from a floating-point type, the function determines the floating-point conversion specifier as indicated in Table 106:

Box 201

Can the current `locale::num_put/num_get` facet handle `basefield` specification? Needs more discussion.

Table 106—Floating-point conversions

State	`stdio` equivalent
`(flags() & floatfield) == fixed`	`%f`
`(flags() & floatfield) == scientific`	`%e`
`(flags() & uppercase) != 0`	`%E`
`Otherwise,`	
	`%g`
`(flags() & uppercase) != 0`	`%G`

Box 202

Is this table clear with regards to f5%e **vs.** `%E` **? -- jss**

6 The conversion specifier has the following additional qualifiers prepended as indicated in Table 107:

Table 107—Floating-point conversions

Type(s)	State	`stdio` equivalent
an integral type other than a character type	`(flags() & showpos) != 0`	`+`
	`(flags() & showbase) != 0`	`#`
a floating-point type	`(flags() & showpos) != 0`	`+`
	`(flags() & showpoint) != 0`	`#`

— For any conversion, if `width()` is nonzero, then a field width is specified in the conversion specification. The value is `width()`.

— For conversion from a floating-point type, if `flags() & fixed` is nonzero or if `precision()` is greater than zero, then a precision is specified in the conversion specification. The value is `precision()`.

7 Moreover, for any conversion, padding with the fill character returned by `fill()` behaves as follows:

— If `(flags() & adjustfield) == right`, no flag is prepended to the conversion specification, indicating right justification (any padding occurs before the converted text). A fill character occurs wherever `fprintf` generates a space character as padding.

— If `(flags() & adjustfield) == internal`, the flag `0` is prepended to the conversion specification, indicating internal justification (any padding occurs within the converted text). A fill character occurs wherever `fprintf` generates a `0` as padding.[157)]

8 Otherwise, the flag `-` is prepended to the conversion specification, indicating left justification (any padding occurs after the converted text). A fill character occurs wherever `fprintf` generates a space character as padding.

9 Unless explicitly stated otherwise for a particular inserter, each formatted output function calls `width(0)` after determining the field width.

27.2.4.2.2 `basic_ostream::operator<<` [lib.ostream.inserters]

```
ostream_type& operator<<(ostream_type& (*pf)(ostream_type&))
```

1 Returns `(*pf)(*this)`.[158)]

```
ostream_type& operator<<(ios_type& (*pf)(ios_type&))
```

2 Calls `(*(basic_ios<charT,baggage>*)pf)(*this)`, then returns `*this`.[159)]

```
ostream_type& operator<<(const char_type* s);
```

3 Converts the NTBS *s* with the conversion specifier `s`.

4 Returns `*this`.

```
ostream_type& operator<<(char_type c);
```

5 Converts the `char_type` *c* with the conversion specifier `c` and a field width of zero. The stored field width (`basic_ios<charT,baggage>::`*wide*) is *not* set to zero.

6 Returns `*this`.

Box 203
The following two were not part of the proposal. Should we keep them?

```
ostream_type& operator<<(unsigned char_type c)
```

7 Returns `operator<<((char_type)c)`.

```
ostream_type& operator<<(signed char_type c)
```

157) The conversion specification `#o` generates a leading `0` which is *not* a padding character.
158) See, for example, the function signature `endl(basic_ostream&)`.
159) See, for example, the function signature `::dec(basic_ios<charT,baggage>&)`.

8 Returns `operator<<((char_type)`*c*`)`.

```
ostream_type& operator<<(bool n);
```

9 Behaves as if:

```
{
  if (flags() & ios::boolalpha) {
    getloc().insert(*this, n);
  } else {
    *this << int(n);
  }
}
```

10 Returns `*this`.

```
ostream_type& operator<<(short n);
```

11 Converts the signed short integer *n* with the integral conversion specifier preceded by `h`.

12 Returns `*this`.

```
ostream_type& operator<<(unsigned short n);
```

13 Converts the unsigned short integer *n* with the integral conversion specifier preceded by `h`.

14 Returns `*this`.

```
ostream_type& operator<<(int n);
```

15 Converts the signed integer *n* with the integral conversion specifier.

16 Returns `*this`.

```
ostream_type& operator<<(unsigned int n);
```

17 Converts the unsigned integer *n* with the integral conversion specifier.

18 Returns `*this`.

```
ostream_type& operator<<(long n);
```

19 Converts the signed long integer *n* with the integral conversion specifier preceded by `l`.

20 Returns `*this`.

```
ostream_type& operator<<(unsigned long n);
```

21 Converts the unsigned long integer *n* with the integral conversion specifier preceded by `l`.

22 Returns `*this`.

```
ostream_type& operator<<(float f);
```

23 Converts the `float` *f* with the floating-point conversion specifier.

24 Returns `*this`.

```
ostream_type& operator<<(double f);
```

25 Converts the `double` *f* with the floating-point conversion specifier.

26 Returns `*this`.

```
ostream_type& operator<<(long double f);
```

27 Converts the `long double` *f* with the floating-point conversion specifier preceded by `L`.

28 Returns `*this`.

```
ostream_type& operator<<(void* p);
```

29 Converts the pointer to `void` *p* with the conversion specifier `p`.

30 Returns `*this`.

```
ostream_type& operator<<(basic_streambuf<charT,baggage>& sb);
```

31 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); LI an exception occurs while getting a character from *sb* (in which case, the exception is rethrown).

32 If the function inserts no characters or if it stopped because an exception was thrown while extracting a character, it calls `setstate(failbit)`. If an exception was thrown while extracting a character and `failbit` is on in `excedptions` the caught exception is rethrown.

33 Returns `*this`.

27.2.4.3 Unformatted output functions [lib.ostream.unformatted]

1 Each unformatted output function begins execution by calling `opfx()`. If that function returns nonzero, the function endeavors to generate the requested output. In any case, the unformatted output function ends by calling `osfx()`, then returning the value specified for the unformatted output function.

27.2.4.3.1 `basic_ostream::put` [lib.ostream::put]

```
int put(char_type c);
```

1 Inserts the character *c*, if possible. Then returns `(unsigned char)`*c*.

2 Otherwise, calls `setstate(badbit)` and returns `eof()`.

27.2.4.3.2 `basic_ostream::write` [lib.ostream::write.str]

```
basic_ostream& write(const char_type* s, streamsize n);
```

1 Obtains characters to insert from successive locations of an array whose first element is designated by *s*. 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)`).

2 Returns `*this`.

Box 204
The following two were not part of the proposal. Should we keep them?

```
basic_ostream& write(const unsigned char_type* s, streamsize n)
```

3 Returns `write((const char_type*)`*s*`,` *n*`)`.

```
basic_ostream& write(const signed char_type* s, streamsize n)
```

4 Returns `write((const char_type*)s, n)`.

27.2.4.3.3 `basic_ostream::write_byte` [lib.ostream::write.byte]

```
    streamsize write_byte(const char* s, streamsize n);
```

1 Inserts bytes by invoking:

```
    rdbuf()->write_byte(s,n);
```

2 In case writing characters fails, calls `setstate(failbit)`.

3 Returns the number of bytes written.

27.2.4.4 Standard `basic_ostream` manipulators [lib.basic.ostream.manip]

27.2.4.4.1 `endl` [lib.endl]

```
    basic_ostream<charT,baggage>& endl(basic_ostream<charT,baggage>& os);
```

1 Calls `os.put(baggage::newline())`, then `os.flush()`.

2 Returns *os*.[160)]

27.2.4.4.2 `ends` [lib.ends]

```
    basic_ostream<charT,baggage>& ends(basic_ostream<charT,baggage>& os);
```

1

Box 205
[To Be Filled]

2 Returns *os*.[161)]

27.2.4.4.3 `flush` [lib.flush]

```
    basic_ostream<charT,baggage>& flush(basic_ostream<charT,baggage>& os);
```

1 Calls `os.flush()`.

Box 206
[To Be Filled]

2 Returns *os*.

27.2.5 Template class `ostreambuf_iterator` [lib.ostreambuf.iterator]

160) 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.

161) The effect of executing `ostr << ends` is to insert a null character in the output sequence controlled by *ostr*. If *ostr* is an object of class `basic_strstreambuf`, the null character can terminate an NTBS constructed in an array object.

Box 207

Because the class `locale::num_put<>` depends on the class, `ostreambuf_iterator` as the fundamental access way to the output stream with some efficiency, the class `ostreambuf_iterator` is defined in the header `<ostream>`.

It really should be defined as part of Clause 24.4.

```
    template <class charT, class baggage = ios_char_baggage<charT> >
    class ostreambuf_iterator {
    public:
        typedef charT char_type;
        typedef baggage baggage_type;
        typedef basic_streambuf<charT,baggage> streambuf;
        typedef basic_ostream<charT,baggage>   ostream;

    public:
        ostreambuf_iterator() : sbuf_(0) {}
        ostreambuf_iterator(ostream& s) : sbuf_(s.rdbuf()) {}
        ostreambuf_iterator(streambuf* s) : sbuf_(s) {}
        ostreambuf_iterator& operator*() { return *this; }
        ostreambuf_iterator& operator++() { return *this; }
        ostreambuf_iterator& operator++(int) { return *this; }
        ostreambuf_iterator& operator=(charT c) {
            sbuf_->sputc(baggage::to_int_type(c));
        }
        bool equal(ostreambuf_iterator& b) {
            return sbuf_ == b.sbuf_;
        }
    private:
//      streambuf* sbuf_;           exposition only
    };
```

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.

```
    output_iterator iterator_category (const ostreambuf_iterator&) {
      return output_iterator();
    }

  template<class charT, class baggage = ios_char_baggage<charT> >
  bool operator==(ostreambuf_iterator<charT,baggage>& a,
                  ostreambuf_iterator<charT,baggage>& b) {
          return a.equal (b);
  }

  template<class charT, class baggage = ios_char_baggage<charT> >
  bool operator!=(ostreambuf_iterator<charT,baggage>& a,
                  ostreambuf_iterator<charT,baggage>& b) {
          return !a.equal (b);
  }
```

27.3 Stream manipulators [lib.manipulators]

1 Headers:

— `<iomanip>`

2 Table 108:

Table 108—Header `<iomanip>` synopsis

Type	Name(s)	
Template classes:		
`basic_imanip`	`imanip`	`wimanip`
`basic_omanip`	`omanip`	`womanip`
`basic_smanip`	`smanip`	`wsmanip`
Template operators:		
`operator<< (omanip)`	`operator>> (imanip)`	
`operator<< (smanip)`	`operator>> (smanip)`	
Template functions:		
`basic_resetiosflags`	`basic_setfill`	`basic_setprecision`
`basic_setbase`	`basic_setiosflags`	
Functions:		
`resetiosflags`	`setfill`	`setprecision`
`setbase`	`setiosflags`	`setw`

3 The header `<iomanip>` defines three template classes and several related functions that use these template classes to provide extractors and inserters that alter information maintained by class `basic_ios` and its derived classes. It also defines several instantiations of these template classes and functions.

27.3.1 Input manipulators [lib.input.manip]

Box 208

Manipulators have gotten way out of hand. The standard mainpulators should be provided, but the general mechanism isn't needed and a simpler way to define them should be possible.

27.3.1.1 Template class `basic_imanip` [lib.template.basic.imanip]

```
template<class T, class charT, class baggage = ios_char_baggage<charT> >
class basic_imanip {
public:
    typedef charT char_type;
    typedef baggage::char_bag::int_type int_type;
    typedef baggage::pos_bag::pos_type  pos_type;
    typedef baggage::pos_bag::off_type  off_type;
    int_type eof()      { return baggage::char_bag::eof(); }
    char_type newline() { return baggage::char_bag::newline(); }
public:
    basic_imanip(
      basic_ios<charT,baggage>& (*pf_arg)(basic_ios<charT,baggage>&, T), T);
//  basic_ios<charT,baggage>& (*pf)(basic_ios<charT,baggage>&, T);        exposition only
//  T manarg;     exposition only
    };

    template <class T> class imanip<T> : public basic_imanip<T,char> {};

    template <class T> class wimanip<T> : public basic_imanip<T,wchar_t> {};
```

1 The template class `basic_imanip<T,charT,baggage>` describes an object that can store a function pointer and an object of type *T*. The designated function accepts an argument of this type `T`.

Box 209
[To Be Filled]

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

— `basic_ios<charT,baggage>& (*pf)(basic_ios<charT,baggage>&, T)`, the function pointer;

— `T manarg`, the object of type *T*.

27.3.1.1.1 `basic_imanip<T>` constructor [lib.basic.imanip.basic.ios.cons]

```
    basic_imanip(basic_ios<charT,baggage>&
                   (*pf_arg)(basic_ios<charT,baggage>&, T), T manarg_arg);
```

1 Constructs an object of class `basic_imanip<T>`, initializing `pf` to *`pf_arg`* and `manarg` to *`manarg_arg`*.

27.3.1.1.2 `operator>>` [lib.ext.basic.imanip]

```
    template<class T, class charT, class baggage>
    basic_istream<charT,baggage>&
        operator>>(basic_istream<charT,baggage>& is,
                   const basic_imanip<T,charT,baggage>& a);
```

1 Calls `(*a.pf)(is, a.manarg)` and catches any exception the function call throws. If the function catches an exception, it calls `is.setstate(basic_ios::failbit)` (the exception is not rethrown).

2 Returns *`is`*.

27.3.2 Template class `basic_omanip` [lib.template.basic.omanip]

```
template<class T, class charT, class baggage = ios_char_baggage<charT> >
class basic_omanip {
public:
    basic_omanip(
      basic_ios<charT,baggage>& (*pf_arg)(basic_ios<charT,baggage>&, T), T);
//  basic_ios<charT,baggage>& (*pf)(basic_ios<charT,baggage>&, T);        exposition only
//  T manarg;     exposition only
};

    template <class T> class omanip<T> : public basic_omanip<T,char> {};

    template <class T> class womanip<T> : public basic_omanip<T,wchar_t> {};
```

1 The template class `basic_omanip<T>` describes an object that can store a function pointer and an object of type *T*. The designated function accepts an argument of this type `T`.

Box 210
[To Be Filled]

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

— `basic_ios<charT,baggage>& (*pf)(basic_ios<charT,baggage>&, T)`, the function pointer;

— *T manarg*, the object of type *T*.

27.3.2.1 basic_omanip constructor [lib.basic.omanip.basic.ios.cons]

```
basic_omanip(
   basic_ios<charT,baggage>& (*pf_arg)(basic_ios<charT,baggage>&,T),
   T manarg_arg);
```

1 Constructs an object of class basic_omanip<*T*>, initializing pf to *pf_arg* and manarg to *manarg_arg*.

27.3.2.2 operator<< [lib.ins.basic.omanip]

```
template<class T, class charT, class baggage>
basic_ostream<charT,baggage>&
    operator<<(basic_ostream<charT,baggage>& os,
               const basic_omanip<T,charT,baggage>& a);
```

1 Calls (**a.pf*)(*os*, *a.manarg*) and catches any exception the function call throws. If the function catches an exception, it calls *os*.setstate(basic_ios::failbit) (the exception is not rethrown).

2 Returns *os*.

27.3.3 Template class basic_smanip [lib.template.basic.smanip]

```
    template<class T, class charT, class baggage = ios_baggage<charT> >
    class basic_smanip {
    public:
        typedef charT char_type;
        typedef baggage::char_bag::int_type int_type;
        typedef baggage::pos_bag::pos_type  pos_type;
        typedef baggage::pos_bag::off_type  off_type;
        int_type  eof()     { return baggage::char_bag::eof(); }
        char_type newline() { return baggage::char_bag::newline(); }
    public:
        basic_smanip(
          basic_ios<charT,baggage>& (*pf_arg)(basic_ios<charT,baggage>&, T),
          T);
//      basic_ios<charT,baggage>& (*pf)(basic_ios<charT,baggage>&, T);   exposition only
//      T manarg;          exposition only
    };

    template <class T> class smanip<T> : public basic_smanip<T,char> {};

    template <class T> class wsmanip<T> : public basic_smanip<T,wchar_t> {};
```

1 The template class basic_smanip<T,charT,baggage> describes an object that can store a function pointer and an object of type *T*. The designated function accepts an argument of this type T.

Box 211
[To Be Filled]

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

— basic_ios<charT,baggage>& (**pf*)(basic_ios<charT,baggage>&, *T*), the function pointer;

— *T manarg*, the object of type *T*.

27.3.3.1 `basic_smanip` constructor [lib.basic.smanip.basic.ios.cons]

```
basic_smanip(
  basic_ios<charT,baggage>& (*pf_arg)(basic_ios<charT,baggage>&, T),
  T manarg_arg);
```

1 Constructs an object of class `basic_smanip<T>`, initializing `pf` to `pf_arg` and `manarg` to `manarg_arg`.

27.3.3.2 `operator>>` [lib.ext.basic.smanip]

```
template<class T, class charT, class baggage>
basic_istream<charT,baggage>&
    operator>>(basic_istream<charT,baggage>& is,
               const basic_smanip<T,charT,baggage>& a);
```

1 Calls `(*a.pf)(is, a.manarg)` and catches any exception the function call throws. If the function catches an exception, it calls `is.setstate(basic_ios::failbit)` (the exception is not rethrown).

2 Returns `is`.

27.3.3.3 `operator<<` [lib.ins.basic.smanip]

```
template<class T, class charT, class baggage>
basic_ostream<charT,baggage>&
    operator<<(basic_ostream<charT,baggage>& os,
               const basic_smanip<T,charT,baggage>& a);
```

1 Calls `(*a.pf)(os, a.manarg)` and catches any exception the function call throws. If the function catches an exception, it calls `os.setstate(basic_ios::failbit)` (the exception is not rethrown).

2 Returns `os`.

27.3.4 Standard manipulators [lib.std.manip]

27.3.4.1 `basic_resetiosflags` [lib.basic.resetiosflags]

```
template<class charT, class baggage>
basic_smanip<basic_ios<charT,baggage>::fmtflags>
    basic_resetiosflags(basic_ios<charT,baggage>::fmtflags mask);
```

1 Returns `basic_smanip<basic_ios<charT,baggage>::fmtflags>(&f, mask)`, where `f` can be defined as:[162)]

```
template<class charT, class baggage>
basic_ios<charT,baggage>& f(basic_ios<charT,baggage>& str,
                            basic_ios<charT,baggage>::fmtflags mask)
{ // reset specified flags
  str.setf((basic_ios<charT,baggage>::fmtflags)0, mask);
  return str;
}
```

162) The expression `cin >> resetbasic_iosflags(basic_ios<charT,baggage>::skipws)` clears `basic_ios<charT,baggage>::skipws` in the format flags stored in the `basic_istream` object `cin` (the same as `cin >> noskipws`), and the expression `cout << resetbasic_iosflags(basic_ios<charT,baggage>::showbase)` clears `basic_ios<charT,baggage>::showbase` in the format flags stored in the `basic_ostream` object `cout` (the same as `cout << noshowbase`).

27.3.4.2 `basic_setiosflags` [lib.basic.setiosflags]

```
template<class charT, class baggage>
basic_smanip<basic_ios<charT,baggage>::fmtflags>
    basic_setiosflags(basic_ios<charT,baggage>::fmtflags mask);
```

1 Returns `basic_smanip<basic_ios<charT,baggage>::fmtflags>(&f,mask)`, where *f* can be defined as:

```
template<class charT, class baggage>
basic_ios<charT,baggage>& f(basic_ios<charT,baggage>& str,
                            basic_ios<charT,baggage>::fmtflags mask)
{ // set specified flags
  str.setf(mask);
  return str;
}
```

27.3.4.3 `basic_setbase` [lib.basic.setbase]

```
template<class charT, class baggage>
basic_smanip<int> basic_setbase(int base);
```

1 Returns `basic_smanip<int>(&f, base)`, where *f* can be defined as:

```
template<class charT, class baggage>
basic_ios<charT,baggage>& f(basic_ios<charT,baggage>& str, int base)
{ // set basefield
  str.setf(n ==  8 ? basic_ios<charT,baggage>::oct :
                           n == 10 ? basic_ios<charT,baggage>::dec :
                           n == 16 ? basic_ios<charT,baggage>::hex :
                           basic_ios<charT,baggage>::fmtflags(0),
                           basic_ios<charT,baggage>::basefield);
  return str;
}
```

27.3.4.4 `basic_setfill` [lib.basic.setfill]

```
basic_smanip<int> basic_setfill(int c);
```

1 Returns `basic_smanip<int>(&f, c)`, where *f* can be defined as:

```
template<class charT, class baggage>
basic_ios<charT,baggage>& f(basic_ios<charT,baggage>& str, int c)
{ // set fill character
  str.fill(c);
  return str;
}
```

27.3.4.5 `basic_setprecision` [lib.basic.setprecision]

```
template<class charT, class baggage>
basic_smanip<int> basic_setprecision(int n);
```

1 Returns `basic_smanip<int>(&f, n)`, where *f* can be defined as:

```
template<class charT, class baggage>
basic_ios<charT,baggage>& f(basic_ios<charT,baggage>& str, int n)
{ // set precision
  str.precision(n);
  return str;
}
```

27.3.4.6 `basic_setw` [lib.basic.setw]

```
template<class charT, class baggage>
basic_smanip<int> basic_setw(int n);
```

1 Returns `basic_smanip<int>(&f, n)`, where *f* can be defined as:

```
template<class charT, class baggage>
basic_ios<charT,baggage>& f(basic_ios<charT,baggage>& str, int n)
{ // set width
  str.width(n);
  return str;
}
```

Box 212
[To Be Filled]

27.4 String-based streams [lib.string.streams]

1 Headers:

— `<sstream>`

— `<strstream>`

— `<cstdlib>` ascii <-> numeric conversions

2 Table 109:

Table 109—Header `<sstream>` synopsis

Type	Name(s)	
Template classes:		
`basic_istringstream`	`basic_ostringstream`	`basic_stringbuf`
Classes:	`stringbuf`	`wstringbuf`
	`istringstream`	`wistringstream`
	`ostringstream`	`wostringstream`

3 Table 110:

Table 110—Header `<strstream>` synopsis

Box 213

I believe that `strstream` is obsolete and should not be included in the standard. The interface is klunky (a "technical term" meaning the description is longer than it ought to be) and has been frequently criticized for making memory management hard. `sstream`(or a class that uses a general STL iterator) should be a replacement for it, not an addition. I am not making an editorial proposal because I realize that this is likely to be controversial. -- jss

Type	Name(s)		
Template classes:	`basic_istrstream`	`basic_ostrstream`	`basic_strstreambuf`
Classes:	`istrstream`	`ostrstream`	`strstreambuf`

4 Table 111:

Table 111—Header `<cstdlib>` synopsis

Type	Name(s)	
Functions:		
	`atoi`	`strtod`
	`atol`	`strtol`

5 *SEE ALSO:* ISO C subclause 7.10.1.

27.4.1 `char*` streams [lib.str.strstreams]

1 The header `<strstream>` defines three types that associate stream buffers with character array objects and assist reading and writing such objects.

27.4.1.1 Template class `basic_strstreambuf` [lib.strstreambuf]

```
template <class charT, class baggage = ios_baggage<charT> >
class basic_strstreambuf : public basic_streambuf<charT,baggage> {
public:
    typedef charT char_type;
    typedef baggage::char_bag::int_type int_type;
    typedef baggage::pos_bag::pos_type  pos_type;
    typedef baggage::pos_bag::off_type  off_type;
    int_type  eof()     { return baggage::char_bag::eof(); }
    char_type newline() { return baggage::char_bag::newline(); }
```

Box 214

Note: null character constraint is needed in ios_baggage.

```
public:
    basic_strstreambuf(streamsize alsize_arg = 0);
    basic_strstreambuf(void* (*palloc_arg)(size_t),
        void (*pfree_arg)(void*));
    basic_strstreambuf(char_type* gnext_arg, streamsize n,
        char_type* pbeg_arg = 0);
```

Box 215
The following two were not part of the proposal. Should we keep them?

```
    basic_strstreambuf(unsigned char_type* gnext_arg, streamsize n,
        unsigned char_type* pbeg_arg = 0);
    basic_strstreambuf(signed char_type* gnext_arg, streamsize n,
        signed char_type* pbeg_arg = 0);
    basic_strstreambuf(const char_type* gnext_arg, streamsize n);
```

Box 216
The following two were not part of the proposal. Should we keep them?

```
    basic_strstreambuf(const unsigned char_type* gnext_arg, streamsize n);
    basic_strstreambuf(const signed char_type* gnext_arg, streamsize n);
    virtual ~basic_strstreambuf();
    void freeze(bool = 1);
    char_type* str();
    int pcount();
protected:
//  virtual int_type overflow (int_type c = eof());     inherited
//  virtual int_type pbackfail(int_type c = eof());     inherited
```

Box 217
The following was not part of the proposal. Should we keep it?

```
//  virtual int      showmany();    inherited
//  virtual int_type underflow();    inherited
//  virtual int_type uflow();    inherited
//  virtual streamsize xsgetn(char_type* s, streamsize n);    inherited
//  virtual streamsize xsputn(const char_type* s, streamsize n);    inherited
//  virtual pos_type seekoff(off_type off,
//          basic_ios<charT,baggage>::seekdir way,
//          basic_ios<charT,baggage>::openmode which
//            = basic_ios<charT,baggage>::in | basic_ios<charT,baggage>::out);    inherited
//  virtual pos_type seekpos(pos_type sp,
//          basic_ios<charT,baggage>::openmode which
//            = basic_ios<charT,baggage>::in | basic_ios<charT,baggage>::out);    inherited
//  virtual basic_streambuf<char_type,baggage>*
//      setbuf(char_type* s, streamsize n);    inherited
//  virtual int sync(); inherited
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 `basic_strstreambuf<charT,baggage>` is derived from `basic_streambuf<charT,baggage>` to associate 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 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 `basic_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.

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

— *`strstate strmode`*, the attributes of the array object associated with the `basic_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.

4 Each object of class `basic_strstreambuf<charT,baggage>` 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`*.

27.4.1.1.1 `basic_strstreambuf` constructors [lib.basic.strstreambuf.cons]

```
basic_strstreambuf(streamsize alsize_arg = 0);
```

1 Constructs an object of class `basic_strstreambuf`, initializing the base class with `basic_streambuf()`. The postconditions of this function are indicated in Table 112:

Table 112—`basic_strstreambuf(streamsize)` effects

Element	Value
`strmode`	*`dynamic`*
`alsize`	*`alsize_arg`*
`palloc`	a null pointer
`pfree`	a null pointer

```
basic_strstreambuf(void* (*palloc_arg)(size_t), void (*pfree_arg)(void*));
```

2 Constructs an object of class `basic_strstreambuf`, initializing the base class with `basic_streambuf()`. The postconditions of this function are indicated in Table 113:

Table 113—`basic_strstreambuf(void* (*)(size_t),void (*)(void*)` effects

Element	Value
`strmode`	*`dynamic`*
`alsize`	an unspecified value
`palloc`	*`palloc_arg`*
`pfree`	*`pfree_arg`*

```
basic_strstreambuf(char_type* gnext_arg,
    streamsize n,
    char_type *pbeg_arg = 0);
```

3 Constructs an object of class `basic_strstreambuf`, initializing the base class with `basic_streambuf()`. The postconditions of this function are indicated in Table 114:

Table 114—`basic_strstreambuf(charT*,streamsize,charT*)` effects

Element	Value
strmode	0
alsize	an unspecified value
palloc	a null pointer
pfree	a null pointer

4 *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)`.

Box 218
Needs work, regarding `charT` length.

— If `n < 0`, *N* is `INT_MAX`. [163)]

5 If `pbeg_arg` is a null pointer, the function executes:

```
setg(gnext_arg, gnext_arg, gnext_arg + N);
```

6 Otherwise, the function executes:

```
setg(gnext_arg, gnext_arg, pbeg_arg);
setp(pbeg_arg,  pbeg_arg + N);
```

Box 219
The following two were not part of the proposal. Should we keep them?

```
basic_strstreambuf(unsigned char_type* gnext_arg,
                   streamsize n,
                   unsigned char_type* pbeg_arg = 0);
```

7 Behaves the same as `basic_strstreambuf((char_type*)gnext_arg, n, (char_type*)pbeg_arg)`.

```
basic_strstreambuf(signed char_type* gnext_arg,
                   streamsize n,
                   signed char_type* pbeg_arg = 0);
```

163) The function signature `strlen(const char*)` is declared in `<cstring>` (21.2). The macro `INT_MAX` is defined in `<climits>` (_lib.support.limits_).

8 Behaves the same as `basic_strstreambuf((char_type*)`*`gnext_arg`*`,` *`n`*`, (char_type*)`*`pbeg_arg`*`)`.

```
basic_strstreambuf(const char_type* gnext_arg, streamsize n);
```

9 Behaves the same as `basic_strstreambuf((char_type*)`*`gnext_arg`*`,` *`n`*`)`, except that the constructor also sets *`constant`* in *`strmode`*.

Box 220
The following two were not part of the proposal. Should we keep them?

```
basic_strstreambuf(const unsigned char_type* gnext_arg, streamsize n);
```

10 Behaves the same as `basic_strstreambuf((const char_type*)`*`gnext_arg`*`,`*`n`*`)`.

```
basic_strstreambuf(const signed char_type* gnext_arg, streamsize n);
```

11 Behaves the same as `basic_strstreambuf((const char_type*)`*`gnext_arg`*`,`*`n`*`)`.

27.4.1.1.2 `basic_strstreambuf` destructor [lib.basic.strstreambuf.des]

```
virtual ~basic_strstreambuf();
```

1 Destroys an object of class `basic_strstreambuf`. The function frees the dynamically allocated array object only if *`strmode & allocated`* is nonzero and *`strmode & frozen`* is zero. (Subclause 3strstreambuf::overflow_ describes how a dynamically allocated array object is freed.)

27.4.1.1.3 `basic_strstreambuf::freeze` [lib.strstreambuf::freeze]

```
void freeze(bool freezefl = 1);
```

1 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`.

27.4.1.1.4 `basic_strstreambuf::str` [lib.strstreambuf::str]

```
char_type* str();
```

1 Calls `freeze()`, then returns the beginning pointer for the input sequence, *`gbeg`*.[164)]

27.4.1.1.5 `basic_strstreambuf::pcount` [lib.strstreambuf::pcount]

```
int pcount() const;
```

1 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`*.

27.4.1.1.6 `basic_strstreambuf::overflow` [lib.strstreambuf::overflow]

```
//  virtual int_type overflow(int_type c = eof());      inherited
```

164) The return value can be a null pointer.

Box 221

This needs to be rewritten in terms of consuming characters to be consistent with the revised protocol for `overflow`.

1 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_type)`*c*.

— If *c* `== eof()`, there is no character to append.

Returns a value other than `eof()`.

2 Returns `eof()` to indicate failure.

3 Notes:

4 The function can alter the number of write positions available as a result of any call.

5 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.[165] If *palloc* is not a null pointer, the function calls `(*`*palloc*`)(`*n*`)` to allocate the new dynamic array 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*.

6 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*.

7 If *strmode* `&` *dynamic* is zero, or if *strmode* `&` *frozen* is nonzero, the function cannot extend the array (reallocate it with greater length) to make a write position available.

27.4.1.1.7 `basic_strstreambuf::pbackfail` [lib.strstreambuf::pbackfail]

```
//  virtual int_type pbackfail(int_type c = eof());     inherited
```

Box 222

This needs to be rewritten in terms of consuming characters to be consistent with the revised protocol for `pbackfail`..

1 Puts back the character designated by *c* to the input sequence, if possible, in one of three ways:

— If *c* `!= eof()`, if the input sequence has a putback position available, and if `(char_type)`*c* `== char_type)`*gnext*`[-1]`, assigns *gnext* `- 1` to *gnext*.

Returns `(char_type)`*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_type)`*c*.

— If *c* `== eof()` and if the input sequence has a putback position available, assigns *gnext* `- 1` to *gnext*.

[165] An implementation should consider `alsize` in making this decision.

Returns `(char_type)`*`c`*.

2 Returns `eof()` to indicate failure.

3 Notes:

> **Box 223**
> Cannot distinguish success and failure if `c == EOF`.

4 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.

> **Box 224**
> The following was not part of the proposal. Should we keep it?

27.4.1.1.8 `basic_strstreambuf::showmany` [lib.strstreambuf::showmany]

`//  virtual int showmany();` ***inherited***

1 Behaves the same as `basic_streambuf::showmany(int)`.

27.4.1.1.9 `basic_strstreambuf::underflow` [lib.strstreambuf::underflow]

`//  virtual int_type underflow();` ***inherited***

> **Box 225**
> This needs to be rewritten in terms of consuming characters to be consistent with the revised protocol for `underflow`..

1 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_type)*`*`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_type)*`*`gnext`*.

2 Returns `eof()` to indicate failure.

3 Notes:

4 The function can alter the number of read positions available as a result of any call.

27.4.1.1.10 `basic_strstreambuf::uflow` [lib.strstreambuf::uflow]

`//  virtual int_type uflow();` ***inherited***

1 Behaves the same as `basic_streambuf::uflow()`.

27.4.1.1.11 `basic_strstreambuf::xsgetn` [lib.strstreambuf::xsgetn]

```
//  virtual streamsize xsgetn(char_type* s, streamsize n);
```
inherited

1 Behaves the same as `basic_streambuf::xsgetn(s,n)`.

27.4.1.1.12 `basic_strstreambuf::xsputn` [lib.strstreambuf::xsputn]

```
//  virtual int xsputn(const char_type* s, streamsize n);
```
inherited

1 Behaves the same as `basic_streambuf::xsputn(s,n)`.

27.4.1.1.13 `basic_strstreambuf::seekoff` [lib.strstreambuf::seekoff]

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

Box 226
Check vs. 27.4.2.1.11

1 Alters the stream position within one of the controlled sequences, if possible, as indicated in Table 115:

Table 115—`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.

Box 227
Comment: this condition is unclear. If the 2nd condition is true, is the 1st condition always true? If so, the 2nd operation may occur, mayn't it?

2 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 116:

Table 116—`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`	*`seekhigh`* minus the beginning pointer (*`seekhigh`* - *`xbeg`*)
`If (`*`newoff`*` + `*`off`*`) < (`*`seeklow`*` - `*`xbeg`*`),` or `(`*`seekhigh`*` - `*`xbeg`*`) < (`*`newoff`*` + `*`off`*`)`	the positioning operation fails

3 Otherwise, the function assigns *`xbeg`* `+` *`newoff`* `+` *`off`* to the next pointer *`xnext`*.

4 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.

Box 228

Note: Need `posT` object which stores an invalid stream position. Comment: Not clear if the constructed object cannot represent the resultant stream position

27.4.1.1.14 `basic_strstreambuf::seekpos` [lib.strstreambuf::seekpos]

```
//  virtual pos_type seekpos(pos_type sp,
//      basic_ios<charT,baggage>::openmode which
//          = basic_ios<charT,baggage>::in | basic_ios<charT,baggage>::out);
```
inherited

Box 229

Check vs. 27.4.2.1.12

1 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.

2 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`*.

3 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.

4

27.4.1.1.15 `basic_strstreambuf::setbuf` [lib.strstreambuf::setbuf]

```
//  virtual basic_streambuf<char_type,baggage>*
//      setbuf(char_type* s, streamsize n);     inherited
```

1 Performs an operation that is defined separately for each class derived from `basic_strstreambuf`.

Box 230
Should be pure virtual.

2 Default behavior: the same as for `basic_streambuf::setbuf(char_type*, streamsize)`.

27.4.1.1.16 `basic_strstreambuf::sync` [lib.strstreambuf::sync]

```
//  virtual int sync();     inherited
```

1 Behaves the same as `basic_streambuf::sync()`.

27.4.1.2 Template class `basic_istrstream` [lib.istrstream]

```
template<class charT, class baggage = int_baggage<charT> >
    class basic_istrstream : public basic_istream<charT,baggage> {
    public:
        typedef charT char_type;
        typedef baggage::char_bag::int_type int_type;
        typedef baggage::pos_bag::pos_type  pos_type;
        typedef baggage::pos_bag::off_type  off_type;
        int_type  eof()     { return baggage::char_bag::eof(); }
        char_type newline() { return baggage::char_bag::newline(); }

    public:
        basic_istrstream(const char_type* s);
        basic_istrstream(const char_type* s, streamsize n);
        basic_istrstream(char_type* s);
        basic_istrstream(char_type* s, streamsize n);
        virtual ~basic_istrstream();
        basic_strstreambuf<char_type,baggage>* rdbuf() const;
        char *str();
    private:
//      basic_strstreambuf<char_type,baggage> sb;        exposition only
//      typedef basic_strstreambuf<char_type,baggage> sb_type;     exposition only
};
```

1 The class `basic_istrstream` is a derivative of `basic_istream` that assists in the reading of objects of class `basic_strstreambuf`. It supplies a `basic_strstreambuf` object to control the associated array object.

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

— *sb*, the `basic_strstreambuf` object.

27.4.1.2.1 `basic_istrstream` constructors [lib.basic.istrstream.cons]

```
    basic_istrstream(const char_type* s);
```

1 Constructs an object of class `basic_istrstream`, initializing the base class with `basic_istream(&sb)`, and initializing *sb* with `sb_type(s, 0)`. *s* shall designate the first element of an NTBS.

Box 231
Needs work, NTBS.

```
basic_istrstream(const char_type* s, streamsize n);
```

2 Constructs an object of class `basic_istrstream`, initializing the base class with `basic_istream(&sb)`, and initializing *sb* with `sb_type(s, n)`. *s* shall designate the first element of an array whose length is *n* elements, and *n* shall be greater than zero.

```
basic_istrstream(char_type* s);
```

3 Constructs an object of class `basic_istrstream`, initializing the base class with `basic_istream(&sb)`, and initializing *sb* with `sb_type((const char_type*)s,0)`. *s* shall designate the first element of an NTBS.

Box 232
Needs work, NTBS.

```
basic_istrstream(char_type* s, streamsize n);
```

4 Constructs an object of class `basic_istrstream`, initializing the base class with `basic_istream(&sb)`, and initializing *sb* with `sb_type((const char_type*)s,n)`. *s* shall designate the first element of an array whose length is *n* elements, and *n* shall be greater than zero.

27.4.1.2.2 basic_istrstream destructor [lib.basic.istrstream.des]

```
virtual ~basic_istrstream();
```

1 Destroys an object of class `basic_istrstream`.

27.4.1.2.3 basic_istrstream::rdbuf [lib.istrstream::rdbuf]

```
basic_strstreambuf* rdbuf() const;
```

1 Returns `(basic_strstreambuf*)&sb`.

27.4.1.2.4 basic_istrstream::str [lib.istrstream::str]

```
char_type* str();
```

1 Returns `sb.str()`.

27.4.1.3 Template class basic_ostrstream [lib.ostrstream]

```
template <class charT, class baggage = int_baggage<charT> >
class basic_ostrstream : public ostream<charT,baggage> {
public:
    typedef charT char_type;
    typedef baggage::char_bag::int_type int_type;
    typedef baggage::pos_bag::pos_type  pos_type;
    typedef baggage::pos_bag::off_type  off_type;
    int_type  eof()     { return baggage::char_bag::eof(); }
    char_type newline() { return baggage::char_bag::newline(); }
    public:
        basic_ostrstream();
        basic_ostrstream(char_type* s, int n,
                         basic_ios<charT,baggage>::openmode mode
                             = basic_ios<charT,baggage>::out);
        virtual ~basic_ostrstream();
        basic_strstreambuf<char_type,baggage>* rdbuf() const;
        void freeze(int freezefl = 1);
        char_type* str();
        int pcount() const;
    private:
//      basic_strstreambuf<char_type,baggage> sb;           exposition only
//      typedef basic_strstreambuf<char_type,baggage> sb_type;     exposition only
};
```

1 The class `basic_ostrstream` is a derivative of `basic_ostream` that assists in the writing of objects of class `basic_strstreambuf`. It supplies a `basic_strstreambuf` object to control the associated array object.

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

— *sb*, the `basic_strstreambuf` object.

27.4.1.3.1 basic_ostrstream constructors [lib.basic.ostrstream.cons]

```
    basic_ostrstream();
```

1 Constructs an object of class `basic_ostrstream`, initializing the base class with `basic_ostream(&sb)`, and initializing *sb* with `sb_type()`.

```
    basic_ostrstream(char_type* s, int n,
        basic_ios<charT,baggage>::openmode mode = basic_ios<charT,baggage>::out);
```

2 Constructs an object of class `basic_ostrstream`, initializing the base class with `basic_ostream(&sb)`, and initializing *sb* with one of two constructors:

— If *mode* `& app` is zero, then *s* shall designate the first element of an array of *n* elements. The constructor is `sb_type(s, n, s)`.

— If *mode* `& app` is nonzero, then *s* shall designate the first element of an array of *n* elements that contains an NTBS whose first element is designated by *s*.

Box 233
Needs work, NTBS and `length()`.

The constructor is `sb_type(s, n, s + ::strlen(s))`. [166)]

27.4.1.3.2 basic_ostrstream destructor [lib.basic.ostrstream.des]

```
    virtual ~basic_ostrstream();
```

1 Destroys an object of class `basic_ostrstream`.

166) The function signature `strlen(const char_type*)` is declared in `<cstring>` (27.5).

27.4.1.3.3 `basic_ostrstream::rdbuf` [lib.ostrstream::rdbuf]

```
    basic_strstreambuf* rdbuf() const;
```

1 Returns `(basic_strstreambuf*)&`*`sb`*.

27.4.1.3.4 `basic_ostrstream::freeze` [lib.ostrstream::freeze]

```
    void freeze(int freezefl = 1);
```

1 Calls *`sb`*`.freeze(`*`freezefl`*`)`.

27.4.1.3.5 `basic_ostrstream::str` [lib.ostrstream::str]

```
    char_type* str();
```

1 Returns *`sb`*`.str()`.

27.4.1.3.6 `basic_ostrstream::pcount` [lib.ostrstream::pcount]

```
    int pcount() const;
```

1 Returns *`sb`*`.pcount()`.

27.4.2 `string` streams [lib.string.strstreams]

1 The header `<sstream>` defines three types that associate stream buffers with objects of class `string`, as described in clause 21.1.2.

27.4.2.1 Template class `basic_stringbuf` [lib.stringbuf]

Box 234
When strings are rationalized with STL, that is when they become containers with iterators, this class ought to be replaced by a method that uses a general STL container.

```
template <class charT, class baggage = int_charT_baggage<charT> >
class basic_stringbuf : public basic_streambuf<charT,baggage> {
public:
    typedef charT char_type;
    typedef baggage::char_bag::int_type int_type;
    typedef baggage::pos_bag::pos_type  pos_type;
    typedef baggage::pos_bag::off_type  off_type;
    int_type  eof()     { return baggage::char_bag::eof(); }
    char_type newline() { return baggage::char_bag::newline(); }

public:
    basic_stringbuf(basic_ios<charT,baggage>::openmode which
                        = basic_ios<charT,baggage>::in
                        | basic_ios<charT,baggage>::out);
    basic_stringbuf(const basic_string<char_type>& str,
                    basic_ios<charT,baggage>::openmode which
                        = basic_ios<charT,baggage>::in
                        | basic_ios<charT,baggage>::out);
    virtual ~basic_stringbuf();
    basic_string<char_type> str() const;
    void str(const basic_string<char_type>& str_arg);
```

```
protected:
//  virtual int_type   overflow (int_type c = eof());   inherited
//  virtual int_type   pbackfail(int_type c = eof());   inherited
```

Box 235

The following was not part of the proposal. Should we keep it?

```
//  virtual int        showmany();       inherited
//  virtual int_type   underflow();      inherited
//  virtual int_type   uflow();          inherited
//  virtual streamsize xsgetn(char_type* s, streamsize n);      inherited
//  virtual streamsize xsputn(const char_type* s, streamsize n);        inherited
//  virtual pos_type   seekoff(off_type off,
//                             basic_ios<charT,baggage>::seekdir way,
//                             basic_ios<charT,baggage>::openmode which
//                               = basic_ios<charT,baggage>::in
//                               | basic_ios<charT,baggage>::out);      inherited
//  virtual pos_type   seekpos(pos_type sp,
//                             basic_ios<charT,baggage>::openmode which
//                               = basic_ios<charT,baggage>::in
//                               | basic_ios<charT,baggage>::out);      inherited
//  virtual basic_streambuf<char_type,baggage>*
//                     setbuf(char_type* s, streamsize n);      inherited
//  virtual int sync();          inherited
private:
//  basic_ios<charT,baggage>::openmode mode;      exposition only
};
```

Box 236

The descriptions of the virtuals in this class need to be brought into agreement with the new descriptions of the generic protocols. As part of that we have

1 Jerry Schwarz proposal:

```
//
//  string             data ;  exposition only
```

```
    class stringbuf : public basic_stringbuf<char> {};

    class wstringbuf : public basic_stringbuf<wchar_t> {};
```

Box 237

Note: same charT type string can be fed.

2 The class basic_stringbuf is derived from basic_streambuf to associate possibly the input sequence and possibly the output sequence with a sequence of arbitrary *characters*. The sequence can be initialized from, or made available as, an object of class basic_string.

Box 238

[To Be Filled]

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

— `basic_ios<charT,baggage>::openmode` *`mode`*, has `in` set if the input sequence can be read, and `out` set if the output sequence can be written.

4 For the sake of exposition, the stored character sequence is described here as an array object.

27.4.2.1.1 `basic_stringbuf` constructors [lib.basic.stringbuf.cons]

```
basic_stringbuf(basic_ios<charT,baggage>::openmode which
                    = basic_ios<charT,baggage>::in
                    | basic_ios<charT,baggage>::out);
```

1 Constructs an object of class `basic_stringbuf`, initializing the base class with `basic_streambuf()`, and initializing *`mode`* with *`which`*. The function allocates no array object.

```
basic_stringbuf(const basic_string<char_type>& str,
                basic_ios<charT,baggage>::openmode which
                    = basic_ios<charT,baggage>::in
                    | basic_ios<charT,baggage>::out);
```

2 Constructs an object of class `basic_stringbuf`, initializing the base class with `basic_streambuf()`, and initializing *`mode`* with *`which`*.

3 If *`str`*`.length()` is nonzero, the function allocates an array object *`x`* whose length *`n`* is *`str`*`.length()` and whose elements *`x`*`[`*`I`*`]` are initialized to *`str`*`[`*`I`*`]`. If *`which`* `& basic_ios::in` is nonzero, the function executes:

```
setg(x, x, x + n);
```

4 If *`which`* `& basic_ios::out` is nonzero, the function executes:

```
setp(x, x + n);
```

27.4.2.1.2 `basic_stringbuf` destructor [lib.basic.stringbuf.des]

```
virtual ~basic_stringbuf();
```

1 Destroys an object of class `basic_stringbuf`.

27.4.2.1.3 `basic_stringbuf::str` [lib.stringbuf::str]

```
basic_string<char_type> str() const;
```

The return value of this function are indicated in Table 117:

Table 117—`str` return values

Condition	Return Value
`(`*`mode`* `& basic_ios::in) != 0` and `(`*`gnext`* `!= 0)`	`basic_string<char_type>(`*`gbeg`*`,`*`gend`* `-` *`gbeg`*`)`
`(`*`mode`* `& basic_ios::out) != 0` and `(`*`pnext`* `!= 0`	`basic_string<char_type>(`*`pbeg`*`,`*`pptr`* `-` *`pbeg`*`)`
Otherwise	`basic_string<char_type>()`

```
void str(const basic_string<char_type>& str_arg);
```

1 If *str_arg*.length() is zero, executes:

```
setg(0, 0, 0);
setp(0, 0);
```

2 and frees storage for any associated array object. Otherwise, the function allocates an array object *x* whose length *n* is *str_arg*.length() and whose elements *x*[*I*] are initialized to *str_arg*[*I*]. If *which* & basic_ios<charT,baggage>::in is nonzero, the function executes:

```
setg(x, x, x + n);
```

3 If *which* & basic_ios<charT,baggage>::out is nonzero, the function executes:

```
setp(x, x + n);
```

27.4.2.1.4 **basic_stringbuf::overflow** [lib.stringbuf::overflow]

```
//  virtual int_type overflow(int_type c = eof());
```
inherited

1 Appends the character designated by *c* to the output sequence, if possible, in one of two ways:

— If *c* != eof() and if either the output sequence has a write position available or the function makes a write position available (as described below), the function assigns *c* to **pnext*++. The function signals success by returning (char_type)*c*.

— If *c* == eof(), there is no character to append. The function signals success by returning a value other than eof().

2 The function can alter the number of write positions available as a result of any call.

3 Returns eof() to indicate failure.

4 The function can make a write position available only if (*mode* & basic_ios<charT,baggage>::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* & basic_ios<charT,baggage>::in) != 0, the function alters the read end pointer *gend* to point just past the new write position (as does the write end pointer *pend*).

27.4.2.1.5 **basic_stringbuf::pbackfail** [lib.stringbuf::pbackfail]

```
//  virtual int_type pbackfail(int_type c = eof());
```
inherited

Box 239

Check vs. 27.2.1.2.27 and 27.5.2.1.7

1 Puts back the character designated by *c* to the input sequence, if possible, in one of three ways:

— If *c* != eof(), if the input sequence has a putback position available, and if (char_type)*c* == (char_type)*gnext*[-1], assigns *gnext* - 1 to *gnext*.

Returns (char_type)*c*.

— If *c* != eof(), if the input sequence has a putback position available, and if *mode* & *basic_ios<charT,baggage>::out* is nonzero, assigns *c* to *--*gnext*.

Returns (char_type)*c*.

— If *c* == eof() and if the input sequence has a putback position available, assigns *gnext* - 1 to *gnext*.

Returns `(char_type)`*c*.

2 Returns `eof()` to indicate failure.

3 Notes:

4 If the function can succeed in more than one of these ways, it is unspecified which way is chosen.

Box 240
The following was not part of the proposal. Should we keep it?

27.4.2.1.6 `basic_stringbuf::showmany` [lib.stringbuf::showmany]

```
//  virtual int showmany();     inherited
```

1 Behaves the same as `basic_streambuf<charT,baggage>::showmany()`.

27.4.2.1.7 `basic_stringbuf::underflow` [lib.stringbuf::underflow]

```
//  virtual int_type underflow();     inherited
```

1 If the input sequence has a read position available, returns `(char_type)*`*gnext*. Otherwise, returns `eof()`.

27.4.2.1.8 `basic_stringbuf::uflow` [lib.stringbuf::uflow]

```
//  virtual int_type uflow();     inherited
```

1 Behaves the same as `basic_streambuf<charT,baggage>::uflow()`.

27.4.2.1.9 `basic_stringbuf::xsgetn` [lib.stringbuf::xsgetn]

```
//  virtual streamsize xsgetn(char_type* s, streamsize n);     inherited
```

1 Behaves the same as `basic_streambuf<charT,baggage>::xsgetn(`*s,n*`)`.

27.4.2.1.10 `basic_stringbuf::xsputn` [lib.stringbuf::xsputn]

```
//  virtual streamsize xsputn(const char_type* s, streamsize n);     inherited
```

1 Behaves the same as `basic_streambuf<charT,baggage>::xsputn(`*s,n*`)`.

27.4.2.1.11 `basic_stringbuf::seekoff` [lib.stringbuf::seekoff]

```
//  virtual pos_type seekoff(off_type off, basic_ios<charT,baggage>::seekdir way,
//                           basic_ios<charT,baggage>::openmode which
//                               = basic_ios<charT,baggage>::in | basic_ios<charT,baggage>::out);
```

Box 241
Check vs. 27.4.1.1.14

1 Alters the stream position within one of the controlled sequences, if possible, as indicated in Table 118:

Table 118—`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 is a null pointer, the positioning operation fails. Otherwise, the function determines *`newoff`* as indicated in Table 119:

Table 119—`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`*`)` `T}`	the positioning operation fails

3 Otherwise, the function assigns *`xbeg`* `+` *`newoff`* `+` *`off`* to the next pointer *`xnext`*.

4 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.

27.4.2.1.12 `basic_stringbuf::seekpos` [lib.stringbuf::seekpos]

```
//  virtual pos_type seekpos(pos_type sp,
//                           basic_ios<charT,baggage>::openmode which
//                             = basic_ios<charT,baggage>::in
//                             | basic_ios<charT,baggage>::out);     inherited
```

Box 242
Check vs. lib.strstreambuf::seekpos

1 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.

2 For a sequence to be positioned, if its next pointer is a null pointer, the positioning operation fails. Otherwise, the function determines *newoff* from `sp.offset()`:

— If *newoff* is an invalid stream position, has a negative value, or has a value greater than (*xend* - *xbeg*), the positioning operation fails.

— Otherwise, the function adds *newoff* to the beginning pointer *xbeg* and stores the result in the next pointer *xnext*.

3 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.

27.4.2.1.13 basic_stringbuf::setbuf [lib.stringbuf::setbuf]

```
//  virtual basic_streambuf<charT,baggage>*
//      setbuf(char_type* s, streamsize n);     inherited
```

1 Performs an operation that is defined separately for each class derived from `basic_stringbuf<charT,baggage>`.

Box 243
Should be pure virtual.

2 Default behavior: the same as for `basic_streambuf<charT,baggage>::setbuf(char_type*, streamsize)`.

27.4.2.1.14 basic_stringbuf::sync [lib.stringbuf::sync]

```
//  virtual int sync();     inherited
```

1 Behaves the same as `basic_streambuf<charT,baggage>::sync()`.

27.4.2.2 Template class basic_istringstream [lib.istringstream]

```
template <class charT, class baggage = ios_baggage<charT> >
class basic_istringstream : public basic_istream<charT,baggage> {
public:
    typedef charT char_type;
    typedef baggage::char_bag::int_type int_type;
    typedef baggage::pos_bag::pos_type  pos_type;
    typedef baggage::pos_bag::off_type  off_type;
    int_type  eof()     { return baggage::char_bag::eof(); }
    char_type newline() { return baggage::char_bag::newline(); }
public:
    basic_istringstream(basic_ios<charT,baggage>::openmode which
                            = basic_ios<charT,baggage>::in);
    basic_istringstream(const basic_string<charT>& str,
                        basic_ios<charT,baggage>::openmode which
                            = basic_ios<charT,baggage>::in);
    virtual ~basic_istringstream();
    basic_stringbuf<charT,baggage>* rdbuf() const;
    basic_string<charT> str() const;
    void str(const basic_string<charT>& str);
private:
//  basic_stringbuf<charT,baggage> sb;    exposition only
};

  class istringstream : public basic_istringstream<char> {};

  class wistringstream : public basic_istringstream<char> {};
```

1 The class `basic_istringstream<charT,baggage>` is a derivative of `basic_istream<charT,baggage>` that assists in the reading of objects of class `basic_stringbuf<charT,baggage>`. It supplies a `basic_stringbuf` object to control the associated array object.

Box 244
[To Be Filled]

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

— *sb*, the `basic_stringbuf` object.

27.4.2.2.1 `basic_istringstream` constructors [lib.basic.istringstream.m.cons]

```
    basic_istringstream(basic_ios<charT,baggage>::openmode which
                            = basic_ios<charT,baggage>::in);
```

1 Constructs an object of class `basic_istringstream<charT,baggage>`, initializing the base class with `basic_istream<charT,baggage>(&sb)`, and initializing *sb* with `sb(which)`.

```
    basic_istringstream(const basic_string<charT>& str,
        basic_ios<charT,baggage>::openmode which = basic_ios<charT,baggage>::in);
```

2 Constructs an object of class `basic_istringstream<charT,baggage>`, initializing the base class with `basic_istream<charT,baggage>(&sb)`, and initializing *sb* with `sb(str, which)`.

27.4.2.2.2 `basic_istringstream` destructor [lib.basic.istringstream.des]

```
    virtual ~basic_istringstream();
```

1 Destroys an object of class `basic_istringstream`.

27.4.2.2.3 `basic_istringstream::rdbuf` [lib.istringstream::rdbuf]

```
    basic_stringbuf<charT,baggage>* rdbuf() const;
```

1 Returns `(basic_stringbuf*)&sb`.

27.4.2.2.4 `basic_istringstream::str` [lib.istringstream::str]

```
    basic_string<charT> str() const;
```

1 Returns *sb*`.str()`.

```
    void str(const basic_string<charT>& str_arg);
```

2 Calls *sb*`.str(`*str_arg*`)`.

27.4.2.3 Class `basic_ostringstream` [lib.ostringstream]

```
template <class charT, class baggage = ios_baggage<charT> >
class basic_ostringstream : public basic_ostream<charT,baggage> {
public:
    typedef charT char_type;
    typedef baggage::char_bag::int_type int_type;
    typedef baggage::pos_bag::pos_type  pos_type;
    typedef baggage::pos_bag::off_type  off_type;
    int_type  eof()     { return baggage::char_bag::eof(); }
    char_type newline() { return baggage::char_bag::newline(); }

public:
    basic_ostringstream(basic_ios<charT,baggage>::openmode which
                            = basic_ios<charT,baggage>::out);
    basic_ostringstream(const basic_string<charT>& str,
                        basic_ios<charT,baggage>::openmode which
                            = basic_ios<charT,baggage>::out);
    virtual ~basic_ostringstream();
    basic_stringbuf<charT,baggage>* rdbuf() const;
    basic_string<charT> str() const;
    void str(const basic_string<charT>& str);
private:
//  basic_stringbuf<charT,baggage> sb;    exposition only
};

    class ostringstream : public basic_ostringstream<char> {};

    class wostringstream : public basic_ostringstream<wchar_t> {};
```

1 The class `basic_ostringstream<charT,baggage>` is a derivative of `basic_ostream<charT,baggage>` that assists in the writing of objects of class `basic_stringbuf<charT,baggage>`. It supplies a `basic_stringbuf` object to control the associated array object.

Box 245
[To Be Filled]

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

— *sb*, the `basic_stringbuf<charT,baggage>` object.

27.4.2.3.1 `basic_ostringstream` constructors [lib.basic.ostringstream.cons]

```
    basic_ostringstream(basic_ios<charT,baggage>::openmode which
                            = basic_ios<charT,baggage>::out);
```

1 Constructs an object of class `basic_ostringstream`, initializing the base class with `basic_ostream(&`*sb*`)`, and initializing *sb* with *sb*`(`*which*`)`.

```
basic_ostringstream(const basic_string<charT>& str,
    basic_ios<charT,baggage>::openmode which = basic_ios<charT,baggage>::out);
```

2 Constructs an object of class `basic_ostringstream<charT,baggage>`, initializing the base class with `basic_ostream<charT,baggage>(&`*`sb`*`)`, and initializing *`sb`* with *`sb`*`(`*`str`*`, `*`which`*`)`.

27.4.2.3.2 `basic_ostringstream` destructor [lib.basic.ostringstream.des]

```
virtual ~basic_ostringstream();
```

1 Destroys an object of class `basic_ostringstream`.

27.4.2.3.3 `basic_ostringstream::rdbuf` [lib.ostringstream::rdbuf]

```
basic_stringbuf<charT,baggage>* rdbuf() const;
```

1 Returns `(basic_stringbuf<charT,baggage>*)&`*`sb`*.

27.4.2.3.4 `basic_ostringstream::str` [lib.ostringstream::str]

```
basic_string<charT> str() const;
```

1 Returns *`sb`*`.str()`.

```
void str(const basic_string<charT>& str_arg);
```

2 Calls *`sb`*`.str(`*`str_arg`*`)`.

27.5 File-based streams [lib.file.streams]

1 Headers:

— `<cstream>`

— `<fstream>`

— `<stdiostream>`

— `<cstdio>`

— `<cwchar>`

2 Table 120:

3 Table 121:

Table 121—Header `<cstream>` synopsis

Type	Name(s)	
Template class:	`basic_convbuf`	
Template structs:	`conv_baggage`	`ios_conv_baggage`
Structs:	`conv_baggage<wchar_t>`	`ios4baggage<wstreampos>`

Table 121—Header `<fstream>` synopsis

Type	Name(s)		
Template classes:			
`basic_filebuf`	`basic_ifstream`	`basic_ofstream`	`basic_stdiobuf`
Classes:			
`basic_filebuf<char>`		`filebuf`	`wfilebuf`
`basic_filebuf<wchar_t>`		`ifstream`	`wifstream`
`basic_ifstream<char>`		`istdiostream`	`wofstream`
`basic_ifstream<wchar_t>`		`ofstream`	
`basic_ofstream<char>`		`ostdiostream`	
`basic_ofstream<wchar_t>`		`stdiobuf`	

4 Table 122:

Table 122—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`	

5 Table 123:

Table 123—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`	

SEE ALSO: `ISO C subclause 7.9, Amendment 1 subclause 4.6.2.`

27.5.1 Multi-byte conversions [lib.conv.fstreams]

Box 246

While this may be a useful class, its primary purpose is to allow implementation of `wfilebuf`. In the interest of simplicity I think it is better to specify `wfilebuf` directly and leave the details of implementations upto the system vendors.

1 Jerry Schwarz proposal: Delete this section.

2 The header `<cstream>` defines one type, `basic_convbuf` that associates its internal stream buffers holding a sequence of *characters* with the external source/sink stream representing a sequence of another type of characters.

— **Conversion** There is a bidirectional conversion between the external source/sink stream and *character* sequences held in the `basic_convbuf` class object.

— **uchar_type: underlaid character container type** The external source/sink stream can be regarded as a sequence of a *character container type*, which may be different to `charT`. The character container type on the external source/sink stream is called the *underlaid character container type*, or *uchar_type*.

— **Encoding rule** Performing the conversion from a *uchar_type* character sequences to the corresponding *character* sequence, the `basic_convbuf` may parse the uchar_type sequence to extract the corresponding character represented on the uchar_type sequence. The rule how a certain character is to be represented on a sequence of uchar_type characters is called the *encoding rule*. The `basic_convbuf` can be regarded as having a (virtual) conversion machine which obeys the encoding rule to parse uchar_type sequences.

— **Conversion state** The conversion machine has its internal state corresponding to a certain position of the external source/sink stream. If the `basic_convbuf` stops the conversion to resume later, it has to save its internal state, so a class or a type is prepared for saving the state onto it so that an user of the `basic_convbuf` class object can handle it. The class (or the type) is called the *conversion state*.

—

Box 247

Multibyte character I/O support The `basic_convbuf` can support multibyte character file I/O operations. Provided that the code conversion functions between multibyte characters and wide characters are prepared, we may specify the conversion state, and the conversion functions in the `ios4baggage` so that the `basic_convbuf` can handle this multibyte characters.

27.5.1.1 Template class `conv_baggage` [lib.conv.baggage]

```
template <class charT> struct conv_baggage {}

struct conv_baggage<wchar_t> {
    typedef ios_char_baggage<wchar_t>   char_bag;
    typedef ios_pos_baggage<wstreampos> pos_bag;
    typedef ios4baggage<state_t>   conv_bag;
};
```

1 The template struct `conv_baggage<charT>` is a *baggage class* which maintains the definitions of the types and functions necessary to implement the template class `basic_convbuf`. The template parameter `charT` reprsents the character container type and each specialized version of `conv_baggage` provides the default definitions corresponding to the specialized character container type.

2 The `conv_baggage<charT>` declaration has three members, `ios_char_baggage`, `ios_pos_baggage`, and `ios4baggage`.

3 Although the former two of them are same as in the `ios_baggage<charT>`, the last baggage, `ios4baggage<charT>` provides all the definitions about types and functions necessary to perform conversion.

27.5.1.2 Template class `ios4baggage` [lib.ios.conv.baggage]

```
    template <class stateT> struct ios4baggage {};
```

1 As is the other baggage structs, there is no definition in the declaration of the template struct, `ios4baggage`. All of the definitions related to the conversion and necessary to implement `basic_convbuf` class shall be provided in a template version, `ios4baggage<STATE_T>` specialized for a conversion state `STATE_T`.

27.5.1.2.1 Struct `ios4baggage<STATE_T>` [lib.ios.conv.baggage.state.t]

```
struct ios4baggage<STATE_T> {
    typedef CHAR_T  char_type;
    typedef STATE_T conv_state;
    typedef UPOS_T  conv_upos;
    typedef UCHAR_T uchar_type;

    typedef POS_T   pos_type;
    typedef OFF_T   off_type;

    typedef locale::codecnv<uchar_type,char_type,conv_state> codecvt_in;
    typedef locale::codecnv<char_type,uchar_type,conv_state> codecvt_out;

    locale::result convin(codecvt_in*, conv_state&,
            const uchar_type*, const uchar_type*, const uchar_type*&,
            char_type*, char_typeT*, char_type*&);
    locale::result convout(codecvt_out*, conv_state&,
            const char_type*, const char_type*, const char_type*&,
            uchar_type*, uchar_type*, uchar_type*&);
    pos_type   get_pos(conv_state&, conv_upos&);
    off_type   get_off(conv_state&, conv_upos&);

    conv_state get_posstate pos_type&);
    conv_state get_offstate(off_type&);
    conv_upos  get_posupos (pos_type&);
    conv_upos  get_offupos (off_type&);
};
```

1 A specialized version of the struct `ios4baggage<STATE_T>` is necessary to use the `basic_convbuf` class. In order to construct a specialized version of the struct, the following set of types and functions shall be provided.

— `CHAR_T`: is the *character container type* which shall be same as the template parameter of the class, `conv_baggage`.

— `STATE_T`: is the *conversion state type* which also the template parameter type of this baggage. It is the parsing/tracing state which the function, convin and convout are taken.

— Every type of conversion states have a state, called *initial state*, which corresponds to the state beginning to parse a new sequence. The constructor, `STATE_T::STATE_T()` or `STATE_T::STATE_T(0)` constructs the initial state.

— `UCHAR_T`: is the *underlaid character container type* for the external source/sink stream handled by the `basic_convbuf`.

— `UPOS_T`: is the *repositional information type* for the external source/sink stream. It is used to implement the `seekpos`/`seekoff` member functions in the `basic_convbuf`.

— `POS_T`: is the *repositional information* which the `basic_convbuf` supports for the client of the object. It shall be the same as the template parameter type of the struct `ios_pos_baggage` in the same `conv_baggage`.

— `OFF_T`: is an integral type which is used as the repositional information which the `basic_convbuf` supports.

27.5.1.2.1.1 `convin` [lib.ios.conv.baggage.state.t::convin]

```
    locale::result
        convin(codecvt_in* ccvt, conv_state& stat,
               const uchar_type* from, const uchar_type* from_end,
               const uchar_type*& from_next,
               char_type* to, char_typeT* to_limit, char_type*& to_next);
```

1 The conversion state designated by *`stat`* represents the parsing state on the underlaid source sequence. According to the conversion state designated by *`stat`*, it begins to parse the source character sequence whose begining and end position designated by *`from`* and *`from_end`* to convert them. It stores the result sequence (if any) into the successive location of an array pointed to by *`to`* and *`to_limit`*. The conversion stops when either of the following occurs:

1) parses all of the underlaid character sequences and stores the last *character* into the destination array.

2) fills all of the destination array and no more room to store.

3) encounters an invalid underlaid character in the source sequence.

2 In case (1), it returns ok. In case (2), it returns partial. In case (3), it returns error.

3 In all cases it leaves the *`from_next`* and *`to_next`* pointers pointing one beyond the last character successfully converted and leaves the conversion state onto the *`stat`* `for the next time conversion. If case (3) occurs, the conversion state on the` *`stat`* `becomes undefined value. No more conversion with the value is available.`

4 If the locale-dependent conversion functions are needed, the nconversion function which the `locale::codecnv` facet object, *ccnv* provides is available as the following invokation:

```
ccnv.convert(stat, from, from_end, from_next, to, to_limit, to_next);
```

5 The default conversion function depends on the `locale_codecvt` facets. Users can customize the encoding scheme by specifying their own conversion functions in a new `conv_baggage` class.

27.5.1.2.1.2 convout [lib.ios.conv.baggage.state.t::convout]

```
locale::result convout(codecvt_out* ccvt, conv_state&,
            const char_type from*, const char_type* from_end,
            const char_type*& from_next.,
            uchar_type*, uchar_type*, uchar_type*&);
```

1 The conversion state designated by *stat* represents the parsing state on the underlaid destination sequence. It begins to convert the source character sequence whose begining and end position designated by *from* and *from_end*. According to the conversion state designated by *stat*, it stores the result destination underlaid chracter sequence (if any) into the successive location of an array pointed to by *to* and *to_limit*. The conversion stops when either of the following occurs:

1) consumes all of the source character sequences and stores the last underlaid *character* into the destination array.

2) fills all of the destination array and no more room to store.

3) encounters an invalid character in the source sequence.

2 In case (1), it returns ok. In case (2), it returns partial. In case (3), it returns error.

3 In all cases it leaves the *5from_next* and *to_next* pointers pointing one beyond the last character successfully converted and leaves the conversion state onto the *stat* for the next time conversion. If case (3) occurs, the conversion state on the *stat* becomes undefined value. No more conversion with the value is available.

4 If the locale-dependent conversion functions are needed, the conversion function which the `locale::codecnv` facet object, *ccnv* provides is available as the following invocation;

```
ccnv.convert(stat, from, from_end, from_next, to, to_limit, to_next);
```

5 The default conversion function is depends on the `locale_codecvt` facets. Users can customize the encoding scheme by specifying their own conversion functions in a new `conv_baggage` class.

27.5.1.2.1.3 get_pos and get_off [lib.ios.conv.baggage.state.t::get.pos]

```
pos_type get_pos(conv_state& stat, conv_upos& upos);
off_type get_off(conv_state&, conv_upos&);
```

1 Constructs the `pos_type` object or `off_type` object from the conversion state, *stat* and the repositional information for the underlaid stream, *upos* and returns it. These are used to make the return value of the `basic_convbuf::seekpos/seekoff` protected member function.

27.5.1.2.1.4 get_posstate and get_offstate [lib.ios.conv.baggage.state.t::get.pos.state]

```
conv_state get_posstate(pos_type& pos);
conv_state get_offstate(off_type& off);
```

1 Extract the conversion state held by the repositional information, *pos* or *off*. These are used to reset the conversion state maintained in the `basic_convbuf` class object when the `basic_convbuf::seetpos/seekoff` functions are performed.

27.5.1.2.1.5 ios4baggage<STATE_T>::get_posupos and get_offupos [lib.ios.conv.baggage.state.t::get.pos.upos]

```
    conv_upos  get_posupos(pos_type&);
    conv_upos  get_offupos(off_type&);
```

1 Extract the repositioning information for the underlaid stream from the repositioning information on the basic_convbuf stream. These are used in the basic_convbuf::seekpos/seekoff protected member functions to reposition the external source/sink stream.

27.5.1.2.2 Struct ios4baggage<wstreampos> [lib.ios.conv.baggage.wstreampos]

```
struct ios4baggage<wstreampos> {
    typedef wchar_t char_type;
    typedef STATE_T conv_state;
    typedef UPOS_T  conv_upos;
    typedef char    uchar_type;
    typedef wstreampos pos_type;
    typedef wstreamoff off_type;

    typedef locale::codecnv<char,wchar_t,STATE_T> codecnv_in;
    typedef locale::codecnv<wchar_t,char,STATE_T> codecnv_out;

    locale::result convin(codecvt_in* ccvt, conv_state& stat,
           const char *from, const char* from_end, const char*& from_next,
           wchar_t* to, wchar_t* to_limit, wchar_t*& to_next) {
            return ccvt->convert (stat, from, from_end, from_next,
                 to, to_limit, to_next);
        }
    locale::result convout(codecvt_out* ccvt, conv_state& stat,
           const wchar_t* from, const wchar_t* from_end, const wchar_t*& from_next,
           char* to, char* to_limit, char* to_next) {
            return ccvt->convert (stat, from, from_end, from_next,
                 to, to_limit, to_next);
        }
    pos_type   get_pos(conv_state&, conv_upos&);
    off_type   get_off(conv_state&, conv_upos&);

    conv_state get_posstate(pos_type&);
    conv_state get_offstate(off_type&);
    conv_upos  get_posupos (pos_type&);
    conv_upos  get_offupos (off_type&);
};
```

1 The specialized version of the struct, ios4baggage<wstreampos> supports the wide-oriented basic_convbuf class and the conversion between wide characters and multibyte characters as the underlaid character sequence. The types, STATE_T and UPOS_T are implementation-defined types. The behavior of the conversion functions depend on those of the locale object.

2 The encoding rule of the multibyte characters is implementation-defined.

27.5.1.3 Template class basic_convbuf [lib.basic.convbuf]

```
template <class charT, class baggage = conv_baggage<charT> >
class basic_convbuf : public basic_streambuf<charT,baggage> {
public:
    typedef charT char_type;
    typedef baggage::char_bag::int_type int_type;
    typedef baggage::pos_bag::pos_type  pos_type;
    typedef baggage::pos_bag::off_type  off_type;
    int_type  eof()     { return baggage::char_bag::eof(); }
    char_type newline() { return baggage::char_bag::newline(); }
private:
    typedef baggage::conv_bag::conv_state state_type;
    typedef baggage::conv_bag::conv_upos upos_type ;
public:
    basic_convbuf();

    virtual ~basic_convbuf();

protected:
//  virtual int_type overflow (int_type c = eof());      inherited
//  virtual int_type pbackfail(int_type c = eof());      inherited
//  virtual int_type underflow();                                       inherited
//  virtual int_type uflow();                                           inherited
//  virtual streamsize xsgetn(charT* s, streamsize n);           inherited
//  virtual streamsize xsputn(const charT* s, streamsize n);     inherited
//  virtual pos_type seekoff(off_type, basic_ios<charT,baggage>::seekdir way,
//                           basic_ios<charT,baggage>::openmode which
//                             = basic_ios<charT,baggage>::in
//                             | basic_ios<charT,baggage>::out);        inherited
//  virtual pos_type seekpos(pos_type  sp, basic_ios<charT,baggage>::openmode which
//                             = basic_ios<charT,baggage>::in
//                             | basic_ios<charT,baggage>::out);        inherited
//  virtual basic_streambuf<charT,baggage>*
//                   setbuf(charT* s, streamsize n);     inherited
//  virtual int sync();          inherited
private:
//  state_type state;     exposition only
};
```

1 The template class `basic_convbuf<charT,baggage>` is derived from `basic_streambuf<charT,baggage>` to associate the *character* sequences and the underlaid character sequences. It performs the conversion between these two types of chracter sequences.

2 For the sake of exposition, the `basic_convbuf` maintains the conversion state to restart the conversion to read/write the character sequence from/to the underlaid stream.

27.5.1.3.1 `basic_convbuf` constructor [lib.basic.convbuf.cons]

```
basic_convbuf();
```

1 Constructs an object of template class `basic_convbuf<charT,baggage>`, initializing the base class with `basic_streambuf<charT,baggage>()`, and initializing;

— *`state`* with `state_type(0)` as to specify initial state.

27.5.1.3.2 `basic_convbuf` destructor [lib.basic.convbuf.des]

```
virtual ~basic_convbuf();
```

1 Destroys an object of class `basic_convbuf<charT,baggage>`.

27.5.1.3.3 `basic_convbuf::overflow` [lib.basic.convbuf.overflow]

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

1 Behaves the same as `basic_streambuf<charT,baggage>::overflow(c)`, except that the behavior of ‘‘consuming a character’’ is as follows:

— (1) Converting the characters to be consumed into the underlaid character sequence with the function `baggage::conv_bag::convout()`.

— (2) The result underlaid character sequence is appended to the associated output stream.

27.5.1.3.4 `basic_convbuf::pbackfail` [lib.basic.convbuf::pbackfail]

```
virtual int_type pbackfail(int_type c);
```

1 Puts back the character designated by `c` to the input sequence, if possible, in one of three ways:

Box 248
Because the parsing on the underlaid character sequence generally can only go advance or most of parsing machined cannot go back, some of the `basic_convbuf` implementations cannot ‘‘put back characters directly to the associated input sequence.’’ So the behaviors related to putting back the associated input stream are removed.

— If *`c`* `!= eof()`, if either the input sequence has a putback position available or the function makes a putback position available, and if `(charT)`*`c`* `== (charT)`*`gnext`*`[-1]`, assigns *`gnext`* `- 1` to *`gnext`*.

Returns `(charT)`*`c`*.

— If *`c`* `!= eof()`, if either the input sequence has a putback position available or the function makes a putback position available, and if the function is permitted to assign to the putback position, assigns *`c`* to `*--`*`gnext`*.

Returns `(charT)`*`c`*.

— If *`c`* `== eof()` and if either the input sequence has a putback position available or the function makes a putback posotion available, assigns *`gnext`* `- 1` to *`gnext`*.

Returns `(charT)`*`c`*.

2 Returns `eof()` to signal the failure.

3 Notes:

4 The function does not put back a character directly to the input sequence.

5 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.

6 Default behavior: returns `baggage::char_bag::eof()`.

Box 249
Should be pure virtual.

27.5.1.3.5 `basic_convbuf::underflow` [lib.basic.convbuf::underflow]

`//  virtual int_type underflow();` ***inherited***

1 Returns the first *character* of the *pending sequence*, if possible, without moving the stream position past it. If the pending sequence is null then the function fails.

2 The *pending sequence* of characters is defined as the concatenation of:

a) If *`gnext`* is non-`NULL`, then the *`gend - gnext`* characters starting at *`gnext`*, otherwise the empty sequence.

b) Some sequence (possibly empty) of characters read from the input stream.

— These underlaid character sequence shall be converted by the function `baggage::conv_bag::convin()` before concatinating to construct the pending sequence. In case that there remains some underlaid characters which cannot be converted to any more *characters*, we treat it empty as the subsequence (b).

3 The *result character* is the first character of the *pending sequence*, if any. The *backup sequence* is defined as the concatenation of:

a) If *`gbeg`* is non-`NULL` then empty, otherwise the *`gnext - gbeg`* characters beginning at *`gbeg`*.

b) the result character.

4 The function sets up *`gnext`* and *`gend`* satisfying:

a) In case the pending sequence has more than one character the *`gend - gnext`* characters starting at *`gnext`* are the characters in the pending sequence after the result character.

b) If the pending sequence has exactly one character, then *`gnext`* and *`gend`* may be `NULL` or may both be set to the same non-`NULL` pointer.

5 If *`gbeg`* and *`gnext`* are non-`NULL` then the function is not constrained as to their contents, but the ‘‘unusual backup condition’’ is that either:

a) If the backup sequence contains at least *`gnext - gbeg`* characters then the *`gnext - gbeg`* characters starting at *`gbeg`* agree with the last *`gnext - gbeg`* characters of the backup sequence.

b) Or the n characters starting a *`gnext - n`* agree with the backup sequence (where *`n`* is the length of the backup sequence)

6 Returns `eof()` to indicate failure.

7 Default behavior: returns `baggage::char_bag::eof()`.

Box 250
Should be pure virtual.

27.5.1.3.6 `basic_convbuf::uflow` [lib.basic.convbuf::uflow]

`//  virtual int_type uflow();` ***inherited***

1 Behaves the same as `basic_streambuf<charT,baggage>::uflow()`.

27.5.1.3.7 `basic_convbuf::xsgetn` [lib.convbuf::xsgetn]

`//  virtual streamsize xsgetn(char_type* s, streamsize n);` ***inherited***

1 Behaves the same as `basic_streambuf<charT,baggage>::xsgetn(char_type*, streamsize)`.

27.5.1.3.8 basic_convbuf::xsputn [lib.convbuf::xsputn]

```
//  virtual streamsize xsputn(const char_type* s, streamsize n);     inherited
```

1 Behaves the same as `basic_streambuf<charT,baggage>::xsputn(char_type*, streamsize)`.

27.5.1.3.9 basic_convbuf::seekoff [lib.convbuf::seekoff]

```
//  virtual pos_type seekoff(off_type off, basic_ios<charT,baggage>::seekdir way,
//      basic_ios<charT,baggage>::openmode which
//        = basic_ios<charT,baggage>::in | basic_ios<charT,baggage>::out);    inherited
```

1 Behaves the same as `basic_streambuf<charT,baggage>::seekoff(`*off,way,which*`)`, except that the behavior ‘‘alters the stream position’’ means that:

— (a) altering the underlaid character stream position which they get with the return value of the `baggage::conv_bag::get_posupos(off_type&)`; and,

— (b) restoring the current conversion state by resetting the member, *state* with the return value of the `baggage::conv_bag::get_offstate(off_type&)`.

27.5.1.3.10 basic_convbuf::seekpos [lib.convbuf::seekpos]

```
//  virtual pos_type seekpos(pos_type sp,
//      basic_ios<charT,baggage>::openmode which
//        = basic_ios<charT,baggage>::in | basic_ios<charT,baggage>::out);    inherited
```

1 At first do `sync()` to clear up the get buffer, and alters the stream position, as follows:

— (a) altering the underlaid character stream position by which the function, `baggage::conv_bag::get_posupos(`*pos*`)`, returns. and,

— (b) restoring the member, *state* with the return value of the `baggage::conv_bag::get_posstate(`*pos*`)`.

2 Returns `pos_type`, *newpos*, constructed from the resultant *upos* and *state* if both (a) and (b) are successfully terminated.

3 If either or both (a) or/and (b) fail(s), or if the constructed object cannot represent the resultant stream position, the object stores an invalid stream position.

4 Whichever value is specified in the *which* parameters, the `basic_convbuf` handles only one external source/sink stream.

5 If (a) succeeds, returns a newly constructed `streampos` object returned by `baggage::conv_bag::get_pos(`*state, upos*`)` where *upos* is a `UPOS_T` type object which represent the current position of the underlaid stream.

6 Otherwise, the object stores an invalid stream position.

27.5.1.3.11 basic_convbuf::setbuf [lib.convbuf::setbuf]

```
//  virtual basic_streambuf<charT,baggage>*
//      setbuf(char_type* s, streamsize n);     inherited
```

1 Makes the array of *n* (`charT` type) characters, whose first element is designated by *s*, available for use as a buffer area for the controlled sequences, if possible.

Box 251
The `basic_convbuf` does not fix the buffer management strategy. It remains alternatives for the derived class designers.

27.5.1.3.12 `basic_convbuff::sync` [lib.convbuf::sync]

`//  virtual int sync();` ***inherited***

1 Reflects the pending sequence to the external sink sequence and reset the get/put buffer pointers. It means that if there are some unread sequence on the get buffer and the external source sequense is not seekable, the unread sequence is perfectly lost.

2 The detailed behavior is as follows:

a) Consumes all of the pending sequence of characters, (as *pending sequence* and ‘‘consumes the sequence,’’ see the description in the `basic_streambuf<charT,baggage>::overflow()` _lib.streambuf::overflow_)

   In case that consuming means appending characters to the associated output stream, the *character* sequense shall be converted to the corresponding underlaid character sequence by the `baggage::conv_bag::convout()`.

b) Clears all of the following pointers: *`pbeg`*, *`pnext`*, *`pend`*, *`gbeg`*, *`gnext`*, and *`gend`*.

3 Returns `-1` if (a) fails, otherwise 0.

27.5.1.4 Examples of trait specialization [lib.examples.traits]

```
class fstate_t { ... };       // Implementation-defined conversion state
                              // object which is for file I/O.
class wfstate_t { ... };

template <class charT> struct file_baggage {};

struct file_baggage<char> {
    typedef ios_char_baggage<char> char_bag;
    typedef ios_pos_baggage<streampos> char_pos;
    typedef ios4baggage<fstate_t> conv_pos;
};

struct file_baggage<wchar_t>{
    typedef ios_char_baggage<char> char_bag;
    typedef ios_pos_baggage<wstreampos> char_pos;
    typedef ios4baggage<wfstate_t> conv_pos;
};

template <class stateT> struct ios_file_baggage {};
//
// Specialized for the single-byte filebuf
//

struct ios4baggage<fstate_t> {
    typedef char       char_type;       // char/wchar_t...
    typedef fstate_t   conv_state;      // key parameter(mainly depend on uchar_type)
    typedef streamoff  conv_upos;       // Physical file offset
    typedef char       uchar_type;      // Physical file I/O
    typedef streampos  pos_type;        // Enough to large
    typedef streamoff  off_type;        // Enough to large
    typedef locale::codecnv<char,wchar_t,fstate_t> codecnv_in;
    typedef locale::codecnv<wchar_t,char,fstate_t> codecnv_out;

    locale::result convin(codecvt_in* ccvt, conv_state& stat,
                          const char *from, const char* from_end,
                          const char*& from_next,
                          char* to, char* to_limit, char*& to_next) {
        return ccvt->convert (stat, from, from_end, from_next,
                              to, to_limit, to_next);
    }
    locale::result convout(codecvt_out* ccvt, conv_state& stat,
                           const char* from, const char* from_end,
                           const char*& from_next,
                           char* to, char* to_limit, char* to_next) {
        return ccvt->convert (stat, from, from_end, from_next,
                              to, to_limit, to_next);
    }
    pos_type get_pos(conv_state&, conv_upos&);
    off_type get_off(conv_state&, conv_upos&);

    conv_state get_posstate(pos_type&);
    conv_state get_offstate(off_type&);
    conv_upos  get_posupos (pos_type&);
    conv_upos  get_offupos (off_type&);
};

//
// Specialized for the wfilebuf
//

struct ios4baggage<wfstate_t> {
    typedef wchar_t    char_type;       // char/wchar_t...
    typedef wfstate_t   conv_state;     // key parameter(mainly depend on uchar_type)
    typedef streamoff  conv_upos;       // Physical file offset
    typedef char       uchar_type;      // Physical file I/O
    typedef wstreampos pos_type;        // Enough to large
    typedef wstreamoff off_type;        // Enough to large

    typedef locale::codecnv<char,wchar_t,fstate_t> codecnv_in;
    typedef locale::codecnv<wchar_t,char,fstate_t> codecnv_out;

    locale::result convin (codecvt_in* ccvt, conv_state& stat,
              const char *from, const char* from_end, const char*& from_next,
              wchar_t* to, wchar_t* to_limit, wchar_t*& to_next) {
        return ccvt->convert (stat, from, from_end, from_next,
                              to, to_limit, to_next);
    }
    locale::result convout (codecvt_out* ccvt, conv_state& stat,
              const wchar_t* from, const wchar_t* from_end,
              const wchar_t*& from_next,
              char* to, char* to_limit, char* to_next) {
        return ccvt->convert (stat, from, from_end, from_next,
                              to, to_limit, to_next);
    }
    pos_type get_pos (conv_state&, conv_upos&);
        // wstreampos get_pos (wfstate_t&, streampos);
    off_type get_off (conv_state&, conv_upos&);
        // wstreamoff get_off (wfstate_t&, streampos);
    conv_state get_posstate (pos_type&);
        // wfstate_t get_posstate (wstreampos&);
    conv_state get_offstate (off_type&);
        // wfstate_t get_offstate (wstreamoff&);
    conv_upos  get_posupos (pos_type&);
        //
    conv_upos  get_offupos (off_type&);
};
```

Box 252

Need more description about these definitions

27.5.2 File streams [lib.fstreams]

1 The header `<fstream>` defines six types that associate stream buffers with files and assist reading and writing files.

Box 253

Note: Although `filebuf` object is templated, it accepts only type `char` as `charT` parameter and it continues to provide narrow-oriented file I/O operations. So this subclause remains unchanged. -- mjv

Box 254

Jerry Schwarz proposal: `basic_filebuf<charT,baggage>` should be specified so that it treats a file as a sequence of `charT`. Except for `filebuf` and `wfilebuf` that implies it treats the file as binary.

2 In this subclause, the type name *`FILE`* is a synonym for the type `FILE`.[167)]

— **File** A File provides an external source/sink stream whose *underlaid character type* is `char` (byte).

— **Multibyte characters and Files** Class `basic_filebuf` is derived from `basic_convbuf` to support multibyte character I/O. A File is a sequence of multibyte characters. In order to provide the contents as a wide character sequence, `wfilebuf` should convert between wide character sequences and multibyte character sequences.

— **basic_filebuf** Derived from `basic_convbuf`. Even the single byte character version because single byte code conversion may be necessary... A `basic_filebuf` provide... file I/O, `char/wchar_t` parity. `fstate_t` ... template parameter for `ios_conv_baggage`

— **Customize Mechanism** Change name (`conv_bag` -> `file_bag`), customize `ios4baggage<fstate_t>`. Modify the definition of `ios4baggage`.

— **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.

167) `FILE` is defined in `<cstdio>` (27.5).

27.5.2.1 Template class `basic_filebuf` [lib.filebuf]

```
template <class charT, class baggage = conv_baggage<charT> >
class basic_filebuf : public basic_convbuf<charT,baggage> {
public:
    typedef charT char_type;
    typedef baggage::char_bag::int_type int_type;
    typedef baggage::pos_bag::pos_type  pos_type;
    typedef baggage::pos_bag::off_type  off_type;
    int_type  eof()     { return baggage::char_bag::eof(); }
    char_type newline() { return baggage::char_bag::newline(); }

public:
    basic_filebuf();
    virtual ~basic_filebuf();
    bool is_open() const;
    basic_filebuf<charT,baggage>*
         open(const char* s, basic_ios<charT,baggage>::openmode mode);
//  basic_filebuf<charT,baggage>* open(const char* s, ios::open_mode mode);
    basic_filebuf<charT,baggage>* close();
protected:
//  virtual int_type overflow (int_type c = eof());      inherited
//  virtual int_type pbackfail(int_type c = eof());      inherited
```

Box 255
The following was not part of the proposal. Should we keep it?

```
//  virtual int      showmany();                inherited
//  virtual int_type underflow();               inherited
//  virtual int_type uflow();                   inherited
//  virtual streamsize xsgetn(char_type* s  streamsize n);      inherited
//  virtual streamsize xsputn(const char_type* s, streamsize n);        inherited
//  virtual pos_type seekoff(off_type off,
//                           basic_ios<charT,baggage>::seekdir way,
//                           basic_ios<charT,baggage>::openmode which
//                             = basic_ios<charT,baggage>::in
//                             | basic_ios<charT,baggage>::out);        inherited
//  virtual pos_type seekpos(pos_type sp,
//                           basic_ios<charT,baggage>::openmode which
//                             = basic_ios<charT,baggage>::in
//                             | basic_ios<charT,baggage>::out);        inherited
//  virtual basic_streambuf<charT,baggage>*
//                   setbuf(char_type* s, streamsize n);        inherited
//  virtual int sync();                         inherited
private:
//  FILE* file; exposition only
};

    class filebuf : public basic_filebuf<char> {};

    class wfilebuf : public basic_filebuf<wchar_t> {};
```

1 The class `basic_filebuf<charT,bagggage>` is derived from `basic_streambuf<charT,bagggage>` to associate both the input sequence and the output sequence with an object of type `FILE`.

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

— *`FILE *file`*, points to the `FILE` associated with the object of class `basic_filebuf<charT,bagggage>`.

3 The restrictions on reading and writing a sequence controlled by an object of class `basic_filebuf<charT,bagggage>` are the same as for reading and writing its associated file. In particular:

— If the file is not open for reading or for update, the input sequence cannot be read.

— If the file is not open for writing or for update, the output sequence cannot be written.

— A joint file position is maintained for both the input sequence and the output sequence.

4 In order to support file I/O and multibyte/wide character conversion, the following arrangement is applied to the baggage structures:

— Specify `char` as the underlaid character type, *uchar_type* defined in the `ios4baggage<>`.

— Define `state_t`, the template parameter of the `ios_conv_baggage<>`, so that it can apply to the conversion function. In case the conversion function provided by the `locale::codecnv` facet uses, we adopt a specialized template parameter `stateT` for the locale-oriented conversion function suitable for the multibyte/wide character conversion. Now we assume the name of the conversion state object is `fstate_t`.

— Define `streamoff` (or `streampos`) as the repositional information for the underlaid byte sequence.

— Define `streamoff`, `wstreampos` for the wide-oriented `streambuf`s (or `filebuf`s) so that some composite function in the `ios4baggage()`, for example:

```
wstreampos get_pos(fstate_t fs, streampos s);
wstreamoff get_off(fstate_t fs, streampos s);
```

27.5.2.1.1 `basic_filebuf` constructor [lib.basic.filebuf.cons]

```
basic_filebuf();
```

1 Constructs an object of class `basic_filebuf<charT,baggage>`, initializing the base class with `basic_streambuf<charT,baggage>()`, and initializing *`file`* to a null pointer.

27.5.2.1.2 `basic_filebuf` destructor [lib.basic.filebuf.des]

```
virtual ~basic_filebuf();
```

1 Destroys an object of class `basic_filebuf<charT,baggage>`. Calls `close()`.

27.5.2.1.3 `basic_filebuf::is_open` [lib.filebuf::is.open]

```
bool is_open() const;
```

1 Returns `true` if *`file`* is not a null pointer.

27.5.2.1.4 `basic_filebuf::open` [lib.filebuf::open]

```
basic_filebuf<charT,baggage>* open(const char* s,
                                   basic_ios<charT,baggage>::openmode mode);
```

1 If *`file`* is not a null pointer, returns a null pointer. Otherwise, calls `basic_streambuf<charT,baggage>::basic_streambuf()`. It then opens a file, if possible, whose name is the NTBS *`s`*, ‘‘as if’’ by calling `fopen(`*`s`*`,`*`modstr`*`)` and assigning the return value to *`file`*.

2 The NTBS *`modstr`* is determined from *`mode`* `& ~basic_ios<charT,baggage>::ate` as indicated in Table 124:

Table 124—File open modes

`basic_ios<charT,baggage>` 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"`

3 If the resulting *`file`* is not a null pointer and *`mode & basic_ios<charT,baggage>::ate`* is nonzero, calls `fseek(`*`file`*`, 0, SEEK_END)`.[168)]

4 If `fseek` returns a null pointer, calls `close()` and returns a null pointer. Otherwise, returns `this`.

```
basic_filebuf<charT,baggage>* open(const char* s, ios::open_mode mode);
```

5 Returns `open(`*`s`*`, basic_ios<charT,baggage>::openmode(`*`mode`*`))`.

27.5.2.1.5 `basic_filebuf::close` [lib.filebuf::close]

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

1 If *`file`* is a null pointer, returns a null pointer. Otherwise, if the call `fclose(`*`file`*`)` returns zero, the function stores a null pointer in *`file`* and returns `this`.[169)] Otherwise, returns a null pointer.

27.5.2.1.6 `basic_filebuf::overflow` [lib.filebuf::overflow]

```
//  virtual int_type overflow(int_type c = eof());     inherited
```

1 Behaves the same as `basic_streambuf<charT,baggage>::overflow(`*`c`*`)`, except that tThe behavior of ‘‘consuming a character’’ is as follows:

— (1) Converting the characters to be consumed into the underlaid character sequence with the function `baggage::conv_bag::convout()`. During the conversion, `convout` maintains the conversion state in the member *`state`*. At the end of execution of the conversion, the conversion state is saved on it.

— (2) The result underlaid character sequence is written to the file specified by *`file`*.

At the consumption of the pending characters, none of them are discarded. All of the characters are converted to be writted to the file.

168) 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>` (27.5).
169) The function signature `fclose(FILE*)` is declared, in `<cstdio>` (27.5).

2 Returns `eof()` to indicate failure. If *`file`* is a null pointer, the function always fails.

27.5.2.1.7 `basic_filebuf::pbackfail` [lib.filebuf::pbackfail]

`//  virtual int_type pbackfail(int_type c = eof());` ***inherited***

Box 256
Check vs. 27.2.1.2.27 and 27.4.1.1.7 and 27.5.1.3.4

1 Puts back the character designated by *`c`* to the input sequence, if possible, in one of four ways:

Box 257
Because the parsing on the underlaid character sequence generally can only go advance or most of parsing machined cannot go back, some of the `basic_convbuf` implementations cannot ‘‘put back characters directly to the associated input sequence.’’ So the behaviors related to putting back the associated input stream are removed.

— If *`c`* `!= eof()` and if the function makes a putback position available and if `(char_type)`*`c`* `== (char_type)`*`gnext`*`[-1]`, assigns *`gnext`* `- 1` to *`gnext`*.

Returns `(char_type)`*`c`*.

— If *`c`* `!= eof()` and if the function makes a putback position available and if the function is permitted to assign to the putback position, assigns *`c`* to `*--`*`gnext`*.

Returns `(char_type)`*`c`*.

— If *`c`* `== eof()` and if either the input sequence has a putback position available or the function makes a putback position available, assigns *`gnext`* `- 1` to *`gnext`*.

Returns `(char_type)`*`c`*.

2 Returns `eof()` to indicate failure.

3 Notes:

4 If *`file`* is a null pointer, the function always fails.

5 The function does not put back a character directly to the input sequence.

6 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.

7 Default behavior: returns `baggage::char_bag::eof()`.

Box 258
Should be pure virtual.

Box 259
Shall we impose Library uses onto performing `sync()`... make gbuffer/pbuffer empty every time they try to write after read or vice versa, as in the current MSE. -- mjv

Box 260

No. On systems where this is necessary the `basic_filebuf` virtuals should keep track of the last operation performed on the `FILE` and do whatever is neccessary (equivalent of an `fseek`?) in order to reverse the direction. -- jss

Box 261

The following was not part of the proposal. Should we keep it?

27.5.2.1.8 `basic_filebuf::showmany` [lib.filebuf::showmany]

```
//  virtual int showmany();     inherited
```

1 Behaves the same as `basic_streambuf::showmany()`.[170)]

27.5.2.1.9 `basic_filebuf::underflow` [lib.filebuf::underflow]

```
//  virtual int_type underflow();     inherited
```

1 Behaves the same as `basic_convbuf<charT,baggage>::underflow()`, except that the underlaid input character sequence is the byte sequence in the file specified by *`file`*.

Box 262

Describing the behavior about maintaining the conversion state is needed.

27.5.2.1.10 `basic_filebuf::uflow` [lib.filebuf::uflow]

```
//  virtual int_type uflow();     inherited
```

1 Behaves the same as `basic_convbuf<charT,baggage>::uflow()`.

27.5.2.1.11 `basic_filebuf::xsgetn` [lib.filebuf::xsgetn]

```
//  virtual streamsize xsgetn(char_type* s, streamsize n);     inherited
```

1 Behaves the same as `basic_convbuf<charT,baggage>::xsgetn(`*`s,n`*`)`.

27.5.2.1.12 `basic_filebuf::xsputn` [lib.filebuf::xsputn]

```
//  virtual streamsize xsputn(const char_type* s, streamsize n);     inherited
```

1 Behaves the same as `basic_convbuf<charT,baggage>::xsputn(`*`s,n`*`)`.

27.5.2.1.13 `basic_filebuf::seekoff` [lib.filebuf::seekoff]

```
//  virtual pos_type seekoff(off_type off, basic_ios<charT,baggage>::seekdir way,
//                           basic_ios<charT,baggage>::openmode which
//                             = basic_ios<charT,baggage>::in
//                             | basic_ios<charT,baggage>::out);     inherited
```

170) 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.

1 Alters the stream position within the controlled sequences, if possible, as described below.

2 Returns a newly constructed `pos_type` object that stores the resultant stream position, if possible. If the positioning operation fails, or if the object cannot represent the resultant stream position, the object stores an invalid stream position.

3 If *`file`* is a null pointer, the positioning operation fails. Otherwise, the function determines one of three values for the argument *`whence`*, of type `int`, as indicated in Table 125:

Table 125—`seekoff` effects

way Value	**`stdio`** Equivalent
`basic_ios::beg`	`SEEK_SET`
`basic_ios::cur`	`SEEK_CUR`
`basic_ios::end`	`SEEK_END`

4 The function then calls `fseek(`*`file, off, whence`*`)` and, if that function returns nonzero, the positioning operation fails.[171)]

5 The function extracts the conversion state from *`off`* by means of `get_offstate()` to reset the *`state`* member.

27.5.2.1.14 `basic_filebuf::seekpos` [lib.filebuf::seekpos]

```
//  virtual pos_type seekpos(pos_type sp,
//                           basic_ios<charT,baggage>::openmode which
//                             = basic_ios<charT,baggage>::in
//                             | basic_ios<charT,baggage>::out);
```
inherited

Box 263
[To Be Filled]

27.5.2.1.15 `basic_filebuf::setbuf` [lib.filebuf::setbuf]

```
//  virtual basic_streambuf* setbuf(char_type* s, int n);
```
inherited

Box 264
[To Be Filled]

27.5.2.1.16 `basic_filebuf::sync` [lib.filebuf::sync]

```
//  virtual int sync();
```
inherited

Box 265
[To Be Filled]

171) The macros `SEEK_SET`, `SEEK_CUR`, and `SEEK_END` are defined, and the function signature `fseek(FILE*, long, int)` is declared, in `<cstdio>` (27.5).

27.5.2.2 Template class `basic_ifstream` [lib.ifstream]

```
template <class charT, class baggage = file_baggage<charT> >
class basic_ifstream : public basic_istream<charT,baggage> {
public:
    typedef charT char_type;
    typedef baggage::char_bag::int_type int_type;
    typedef baggage::pos_bag::pos_type  pos_type;
    typedef baggage::pos_bag::off_type  off_type;
    int_type  eof()     { return baggage::char_bag::eof(); }
    char_type newline() { return baggage::char_bag::newline(); }

public:
    basic_ifstream();
    basic_ifstream(const char* s, openmode mode = in);
    virtual ~basic_ifstream();
    basic_filebuf<charT,baggage>* rdbuf() const;
    bool is_open();
    void open(const char* s, openmode mode = in);
//  void open(const char* s, open_mode mode = in);     optional
    void close();
private:
//  basic_filebuf<charT,baggage> fb;      exposition only
};

    class ifstream : public basic_ifstream<char> {};

    class wifstream : public basic_ifstream<wchar_t> {};
```

1 The class `basic_ifstream<charT,baggage>` is a derivative of `basic_istream<charT,baggage>` that assists in the reading of named files. It supplies a `basic_filebuf<charT,baggage>` object to control the associated sequence.

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

— `basic_filebuf<charT,baggage>` *`fb`*, the `basic_filebuf` object.

27.5.2.2.1 `basic_ifstream` constructors [lib.basic.ifstream.cons]

```
    basic_ifstream();
```

1 Constructs an object of class `basic_ifstream<charT,baggage>`, initializing the base class with `basic_istream<charT,baggage>(&`*`fb`*`)`.

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

2 Constructs an object of class `basic_ifstream`, initializing the base class with `basic_istream<charT,baggage>(&`*`fb`*`)`, then calls `open(`*`s`*`, `*`mode`*`)`.

27.5.2.2.2 `basic_ifstream` destructor [lib.basic.ifstream.des]

```
    virtual ~basic_ifstream();
```

1 Destroys an object of class `basic_ifstream<charT,baggage>`.

27.5.2.2.3 `basic_ifstream::rdbuf` [lib.ifstream::rdbuf]

```
    basic_filebuf<charT,baggage>* rdbuf() const;
```

1 Returns *fb*.

27.5.2.2.4 basic_ifstream::is_open [lib.ifstream::is.open]

```
    bool is_open();
```

1 Returns *fb*.is_open().

27.5.2.2.5 basic_ifstream::open [lib.ifstream::open]

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

1 Calls *fb*.open(*s*, *mode*). If is_open() returns zero, calls setstate(failbit).

27.5.2.2.6 basic_ifstream::close [lib.ifstream::close]

```
    void close();
```

1 Calls *fb*.close() and, if that function returns zero, calls setstate(failbit).

27.5.2.3 Template class basic_ofstream [lib.ofstream]

```
template <class charT, class baggage = file_baggage<charT> >
class basic_ofstream : public basic_ostream<charT,baggage> {
public:
    typedef charT char_type;
    typedef baggage::char_bag::int_type int_type;
    typedef baggage::pos_bag::pos_type  pos_type;
    typedef baggage::pos_bag::off_type  off_type;
    int_type  eof()     { return baggage::char_bag::eof(); }
    char_type newline() { return baggage::char_bag::newline(); }

public:
    basic_ofstream();
    basic_ofstream(const char* s, openmode mode = out);
    virtual ~basic_ofstream();
    basic_filebuf<charT,baggage>* rdbuf() const;
    bool is_open();
    void open(const char* s, openmode mode = out | trunc);
//  void open(const char* s, open_mode mode = out | trunc);     optional
    void close();

private:
//  basic_filebuf<charT,baggage> fb;      exposition only
};

    class ofstream : public basic_ofstream<char> {};

    class wofstream : public basic_ofstream<wchar_t> {};
```

1 The class basic_ofstream<charT,baggage> is a derivative of basic_ostream<charT,baggage> that assists in the writing of named files. It supplies a basic_filebuf<charT,baggage> object to control the associated sequence.

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

— basic_filebuf<charT,baggage> *fb*, the basic_filebuf object.

27.5.2.3.1 `basic_ofstream` constructors [lib.basic.ofstream.cons]

```
basic_ofstream();
```

1 Constructs an object of class `basic_ofstream<charT,baggage>`, initializing the base class with `basic_ostream<charT,baggage>(&`*`fb`*`)`.

```
basic_ofstream(const char* s, openmode mode = out);
```

2 Constructs an object of class `basic_ofstream<charT,baggage>`, initializing the base class with `basic_ostream<charT,baggage>(&`*`fb`*`)`, then calls `open(`*`s`*`, `*`mode`*`)`.

27.5.2.3.2 `basic_ofstream` destructor [lib.basic.ofstream.des]

```
virtual ~basic_ofstream();
```

1 Destroys an object of class `basic_ofstream<charT,baggage>`.

27.5.2.3.3 `basic_ofstream::rdbuf` [lib.ofstream::rdbuf]

```
basic_filebuf<charT,baggage>* rdbuf() const;
```

1 Returns `(basic_filebuf<charT,baggage>*)&`*`fb`*.

27.5.2.3.4 `basic_ofstream::is_open` [lib.ofstream::is.open]

```
bool is_open();
```

1 Returns *`fb`*`.is_open()`.

27.5.2.3.5 `basic_ofstream::open` [lib.ofstream::open]

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

1 Calls *`fb`*`.open(`*`s`*`, `*`mode`*`)`. If `is_open()` is then false, calls `setstate(failbit)`.

27.5.2.3.6 `basic_ofstream::close` [lib.ofstream::close]

```
void close();
```

1 Calls *`fb`*`.close()` and, if that function returns zero, calls `setstate(failbit)`.

27.5.3 `stdio` streams [lib.stdio.fstreams]

27.5.3.1 Template class `basic_stdiobuf` [lib.basic.stdiobuf]

Box 266

`basic_stdiobuf` may be useful, and it is based on (untemplatized) current practice, but there is nothing fundamental in it (it can be defined by a user in a fully portable and efficient way) no other part of the workiing paper depends on it and it is used relatively little. So in the interests of keeping the working paper small

1 Jerry Schwarz proposal: Delete this section.

```
    template <class charT, class baggage = stdio_baggage<charT> >
    class basic_stdiobuf : public streambuf<charT,baggage> {
    public:
        typedef charT char_type;
        typedef baggage::char_bag::int_type int_type;
        typedef baggage::pos_bag::pos_type  pos_type;
        typedef baggage::pos_bag::off_type  off_type;
        int_type  eof()     { return baggage::char_bag::eof(); }
        char_type newline() { return baggage::char_bag::newline(); }

    public:
    basic_stdiobuf(FILE* file_arg = 0);
    virtual ~basic_stdiobuf();
    bool buffered() const;
    void buffered(bool buf_fl);

    protected:
//  virtual int_type overflow (int_type c = eof());      inherited
//  virtual int_type pbackfail(int_type c = eof());      inherited
```

Box 267
The following was not part of the proposal. Should we keep it?

```
//  virtual int      showmany();        inherited
//  virtual int_type underflow();       inherited
//  virtual int_type uflow();   inherited
//  virtual streamsize xsgetn(char_type* s, streamsize n);      inherited
//  virtual streamsize xsputn(const char_type* s, streamsize n);        inherited
//  virtual pos_type seekoff(off_type off, basic_ios<charT,baggage>::seekdir way,
//                           basic_ios<charT,baggage>::openmode which
//                             = basic_ios<charT,baggage>::in
//                             | basic_ios<charT,baggage>::out);        inherited
//  virtual pos_type seekpos(pos_type sp,
//                           basic_ios<charT,baggage>::openmode which
//                             = basic_ios<charT,baggage>::in
//                             | basic_ios<charT,baggage>::out);        inherited
//  virtual int sync(); inherited
private:
//  FILE* file; exposition only
//  bool is_buffered;                               exposition only
    };
```

Box 268
[To Be Filled]

2 The class `basic_stdiobuf<charT,baggage>` is derived from `basic_streambuf<charT,baggage>` to associate both the input sequence and the output sequence with an externally supplied object of type `FILE`.[172)]

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

— *`FILE *file`*, points to the `FILE` associated with the stream buffer;

— bool *`is_buffered`*, nonzero if the `basic_stdiobuf` object is *buffered*, and hence need not be kept synchronized with the associated file (as described below).

172) Type `FILE` is defined in `<cstdio>` (27.5).

4 The restrictions on reading and writing a sequence controlled by an object of class `basic_stdiobuf` are the same as for an object of class `basic_filebuf`.

5 If an `basic_stdiobuf` object is not buffered and *`file`* is not a null pointer, it is kept synchronized with the associated file, as follows: as indicated in Table 126:[173)]

Table 126—`stdiobuf` synchronization

`stdiobuf`	**`stdio` Equivalent**
`sputc(`*`c`*`)`	`fputc(`*`c,file`*`)`
`sputbackc(`*`c`*`)`	`ungetc(`*`c,file`*`)`
`sbumpc()`	`fgetc(`*`file`*`)`

27.5.3.1.1 `basic_stdiobuf::basic_stdiobuf` constructor [lib.basic.stdiobuf.cons]

```
basic_stdiobuf(FILE* file_arg = 0);
```

1 Constructs an object of class `basic_stdiobuf<charT,baggage>`, initializing the base class with `basic_streambuf<charT,baggage>()`, and initializing *`file`* to *`file_arg`* and *`is_buffered`* to zero.

27.5.3.1.2 `basic_stdiobuf` destructor [lib.basic.stdiobuf.des]

```
virtual ~basic_stdiobuf();
```

1 Destroys an object of class `basic_stdiobuf<charT,baggage>`.

27.5.3.1.3 `basic_stdiobuf::buffered` [lib.basic.stdiobuf::buffered]

```
bool buffered() const;
```

1 Returns `true` if *`is_buffered`* is nonzero.

```
void buffered(bool buf_fl);
```

2 Assigns *`buf_fl`* to *`is_buffered`*.

27.5.3.1.4 `basic_stdiobuf::overflow` [lib.basic.stdiobuf::overflow]

```
//  virtual int_type overflow(int_type c = eof());      inherited
```

1 Behaves the same as `basic_filebuf<charT,baggage>::overflow(int)`, subject to the buffering requirements specified by *`is_buffered`*.

27.5.3.1.5 `basic_stdiobuf::pbackfail` [lib.basic.stdiobuf::pbackfail]

```
//  virtual int_type pbackfail(int_type c = eof());      inherited
```

1 Behaves the same as `basic_filebuf<charT,baggage>::pbackfail(int)`, subject to the buffering requirements specified by *`is_buffered`*.

[173)] The functions `fgetc(FILE*)`, `fputc(int, FILE*)`, and `ungetc(int, FILE*)` are declared in `<cstdio>` (27.5).

Box 269
The following was not part of the proposal. Should we keep it?

27.5.3.1.6 `basic_stdiobuf::showmany` [lib.basic.stdiobuf::showmany]

```
//  virtual int showmany();
```
inherited

1 Behaves the same as `basic_streambuf<charT,baggage>::showmany()`.[174)]

27.5.3.1.7 `basic_stdiobuf::underflow` [lib.basic.stdiobuf::underflow]

```
//  virtual int_type underflow();
```
inherited

1 Behaves the same as `basic_filebuf<charT,baggage>::underflow()`, subject to the buffering requirements specified by *`is_buffered`*.

27.5.3.1.8 `basic_stdiobuf::uflow` [lib.basic.stdiobuf::uflow]

```
//  virtual int_type uflow();
```
inherited

1 Behaves the same as `basic_filebuf<charT,baggage>::uflow()`, subject to the buffering requirements specified by *`is_buffered`*.

27.5.3.1.9 `basic_stdiobuf::xsgetn` [lib.basic.stdiobuf::xsgetn]

```
//  virtual streamsize xsgetn(char_type* s, streamsize n);
```
inherited

1 Behaves the same as `basic_streambuf<charT,baggage>::xsgetn(`*`s,n`*`)`.

27.5.3.1.10 `basic_stdiobuf::xsputn` [lib.basic.stdiobuf::xsputn]

```
//  virtual streamsize xsputn(const char_type* s, streamsize n);
```
inherited

1 Behaves the same as `basic_streambuf<charT,baggage>::xsputn(`*`s,n`*`)`.

27.5.3.1.11 `basic_stdiobuf::seekoff` [lib.basic.stdiobuf::seekoff]

```
//  virtual pos_type seekoff(off_type off, basic_ios<charT,baggage>::seekdir way,
//                           basic_ios<charT,baggage>::openmode which
//                             = basic_ios<charT,baggage>::in
//                             | basic_ios<charT,baggage>::out);
```
inherited

1 Behaves the same as `basic_filebuf<charT,baggage>::seekoff(`*`off,way,which`*`)`

27.5.3.1.12 `basic_stdiobuf::seekpos` [lib.basic.stdiobuf::seekpos]

```
//  virtual pos_type seekpos(pos_type sp,
//                           basic_ios<charT,baggage>::openmode which
//                             = basic_ios<charT,baggage>::in
//                             | basic_ios<charT,baggage>::out);
```
inherited

1 Behaves the same as `basic_filebuf<charT,baggage>::seekpos(`*`sp,which`*`)`

174) An implementation might well provide an overriding definition for this function signature if it can determine that more characters can be read from the input sequence.

27.5.3.1.13 `basic_stdiobuf::setbuf` [lib.basic.stdiobuf::setbuf]

```
//  virtual basic_streambuf<charT,baggage>* setbuf(char_type* s, int n);      inherited
```

1 Behaves the same as `basic_filebuf<charT,baggage>::setbuf(`*`s,n`*`)`

27.5.3.1.14 `basic_stdiobuf::sync` [lib.basic.stdiobuf::sync]

```
//  virtual int sync();     inherited
```

1 Behaves the same as `basic_filebuf`*`s,n`*`::sync()`

27.5.3.2 Class `istdiostream` [lib.istdiostream]

```
    class istdiostream : public basic_istream<char> {
    public:
        istdiostream(FILE* file_arg = 0);
        virtual ~istdiostream();
        basic_stdiobuf<char>* rdbuf() const;
        bool buffered() const;
        void buffered(bool buf_fl);
    private:
//      basic_stdiobuf<char> fb;            exposition only
    };
```

1 The class `istdiostream` is a derivative of `basic_istream<charT,baggage>` that assists in the reading of files controlled by objects of type `FILE`. It supplies a `basic_stdiobuf<charT,baggage>` object to control the associated sequence.

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

— `basic_stdiobuf` *`fb`*, the `basic_stdiobuf` object.

27.5.3.2.1 `istdiostream::istdiostream` constructor [lib.istdiostream.cons]

```
    istdiostream(FILE* file_arg = 0);
```

1 Constructs an object of class `istdiostream`, initializing the base class with `basic_istream(&`*`fb`*`)` and initializing *`fb`* with `basic_stdiobuf(`*`file_arg`*`)`.

27.5.3.2.2 `istdiostream` destructor [lib.istdiostream.des]

```
    virtual ~istdiostream();
```

1 Destroys an object of class `istdiostream`.

27.5.3.2.3 `istdiostream::rdbuf` [lib.istdiostream::rdbuf]

```
    basic_stdiobuf<char>* rdbuf() const;
```

1 Returns *`fb`*.

27.5.3.2.4 `istdiostream::buffered` [lib.istdiostream::buffered]

```
    bool buffered() const;
```

1 Returns `true` if *`is_buffered`* is nonzero.

27.5.3.2.5 istdiostream::buffered [lib.istdiostream::buffered.b]

```
    void buffered(bool buf_fl);
```

1 Assigns *buf_fl* to *is_buffered*.

27.5.3.3 Class ostdiostream [lib.ostdiostream]

```
    class ostdiostream : public basic_ostream<char> {
    public:
        ostdiostream(FILE* file_arg = 0);
        virtual ~ostdiostream();
        basic_stdiobuf<char>* rdbuf() const;
        bool buffered() const;
        void buffered(bool buf_fl);
    private:
//      basic_stdiobuf<char> fb;          exposition only
    };
```

1 The class `ostdiostream` is a derivative of `basic_ostream` that assists in the writing of files controlled by objects of type `FILE`. It supplies a `basic_stdiobuf` object to control the associated sequence.

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

— `basic_stdiobuf<char>` *fb*, the `basic_stdiobuf` object.

27.5.3.3.1 ostdiostream constructor [lib.ostdiostream.cons]

```
    ostdiostream(FILE* file_arg = 0);
```

1 Constructs an object of class `ostdiostream`, initializing the base class with `basic_ostream(&`*fb*`)` and initializing *fb* with `basic_stdiobuf(`*file_arg*`)`.

27.5.3.3.2 ostdiostream destructor [lib.ostdiostream.des]

```
    virtual ~ostdiostream();
```

1 Destroys an object of class `ostdiostream`.

27.5.3.3.3 ostdiostream::rdbuf [lib.ostdiostream::rdbuf]

```
    basic_stdiobuf<char>* rdbuf() const;
```

1 Returns *fb*.

27.5.3.3.4 ostdiostream::buffered [lib.ostdiostream::buffered]

```
    bool buffered() const;
```

1 Returns `true` if *is_buffered* is nonzero.

27.5.3.3.5 ostdiostream::buffered [lib.ostdiostream::buffered.b]

```
    void buffered(bool buf_fl);
```

1 Assigns *buf_fl* to *is_buffered*.

Annex A (informative) Grammar summary [gram]

1 This summary of C++ syntax is intended to be an aid to comprehension. It is not an exact statement of the language. In particular, the grammar described here accepts a superset of valid C++ constructs. Disambiguation rules (6.8, 7.1, _class.ambig_) must be applied to distinguish expressions from declarations. Further, access control, ambiguity, and type rules must be used to weed out syntactically valid but meaningless constructs.

A.1 Keywords [gram.key]

1 New context-dependent keywords are introduced into a program by `typedef` (7.1.3), namespace (7.3.1), class (9), enumeration (7.2), and `template` (14) declarations.

typedef-name:
    identifier

namespace-name:
    original-namespace-name
    namespace-alias

original-namespace-name:
    identifier

namespace-alias:
    identifier

class-name:
    identifier
    template-class-id

enum-name:
    identifier

template-name:
    identifier

Note that a *typedef-name* naming a class is also a *class-name* (9.1). *

A.2 Lexical conventions [gram.lex]

preprocessing-token:
header-name
identifier
pp-number
character-constant
string-literal
operator
digraph
punctuator
each non-white-space character that cannot be one of the above

digraph:
`<%`
`%>`
`<:`
`:>`
`%:`

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
```

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-constant:
fractional-constant exponent-part$_{opt}$ *floating-suffix*$_{opt}$
digit-sequence exponent-part floating-suffix$_{opt}$

fractional-constant:
digit-sequence$_{opt}$ `.` *digit-sequence*
digit-sequence `.`

exponent-part:
`e` *sign*$_{opt}$ *digit-sequence*
`E` *sign*$_{opt}$ *digit-sequence*

sign: one of
`+ -`

digit-sequence:
digit
digit-sequence digit

floating-suffix: one of
`f l F L`

string-literal:
`"`*s-char-sequence*$_{opt}$`"`
`L"`*s-char-sequence*$_{opt}$`"`

s-char-sequence:
s-char
s-char-sequence s-char

s-char:
any member of the source character set except
the double-quote `"`, backslash `\`, or new-line character
escape-sequence

boolean-literal:
`false`
`true`

A.3 Basic concepts [gram.basic]

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

A.4 Expressions [gram.expr]

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

id-expression:
 unqualified-id
 qualified-id

unqualified-id:
 identifier
 operator-function-id
 conversion-function-id
 ~ *class-name*

qualified-id:
 nested-name-specifier unqualified-id

postfix-expression:
 primary-expression
 postfix-expression `[` *expression* `]`
 postfix-expression `(` $expression\text{-}list_{opt}$ `)`
 simple-type-specifier `(` $expression\text{-}list_{opt}$ `)`
 postfix-expression `.` *id-expression*
 postfix-expression `->` *id-expression*
 postfix-expression `++`
 postfix-expression `--`
 `dynamic_cast <` *type-id* `> (` *expression* `)`
 `static_cast <` *type-id* `> (` *expression* `)`
 `reinterpret_cast <` *type-id* `> (` *expression* `)`
 `const_cast <` *type-id* `> (` *expression* `)`
 `typeid (` *expression* `)`
 `typeid (` *type-id* `)`

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

unary-expression:
 postfix-expression
 `++` *unary-expression*
 `--` *unary-expression*
 unary-operator cast-expression
 `sizeof` *unary-expression*
 `sizeof (` *type-id* `)`
 new-expression
 delete-expression

unary-operator: one of
 `* & + - ! ~`

new-expression:
 $::_{opt}$ `new` $new\text{-}placement_{opt}$ *new-type-id* $new\text{-}initializer_{opt}$
 $::_{opt}$ `new` $new\text{-}placement_{opt}$ `(` *type-id* `)` $new\text{-}initializer_{opt}$

new-placement:
 `(` *expression-list* `)`

new-type-id:
 type-specifier-seq $new\text{-}declarator_{opt}$

new-declarator:
    `*` *cv-qualifier-seq*$_{opt}$ *new-declarator*$_{opt}$
    `::`$_{opt}$ *nested-name-specifier* `*` *cv-qualifier-seq*$_{opt}$ *new-declarator*$_{opt}$
    direct-new-declarator

direct-new-declarator:
    `[` *expression* `]`
    direct-new-declarator `[` *constant-expression* `]`

new-initializer:
    `(` *expression-list*$_{opt}$ `)`

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

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

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

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

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

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

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

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

and-expression:
    equality-expression
    and-expression `&` *equality-expression*

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

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

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

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

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

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

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

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

constant-expression:
  conditional-expression

A.5 Statements [gram.stmt.stmt]

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

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

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

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

statement-seq:
 statement
 statement-seq statement

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

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

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

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

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

declaration-statement:
 declaration

A.6 Declarations [gram.dcl.dcl]

declaration:
 decl-specifier-seq$_{opt}$ *init-declarator-list*$_{opt}$ `;`
 function-definition
 template-declaration
 asm-definition
 linkage-specification
 namespace-definition
 namespace-alias-definition
 using-declaration
 using-directive

decl-specifier-seq$_{opt}$ *init-declarator-list*$_{opt}$ `;`

decl-specifier:
 storage-class-specifier
 type-specifier
 function-specifier
 `friend`
 `typedef`

decl-specifier-seq:
 decl-specifier-seq$_{opt}$ *decl-specifier*

storage-class-specifier:
    `auto`
    `register`
    `static`
    `extern`
    `mutable`

function-specifier:
    `inline`
    `virtual`

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:
  original-namespace-definition
  extension-namespace-definition
  unnamed-namespace-definition

original-namespace-definition:
  `namespace` *identifier* `{` *namespace-body* `}` |

extension-namespace-definition:
  `namespace` *original-namespace-name* `{` *namespace-body* `}` |

unnamed-namespace-definition:
  `namespace` `{` *namespace-body* `}` |

namespace-body:
  declaration-seq$_{opt}$

id-expression: ||
  unqualified-id ||
  qualified-id ||

nested-name-specifier: ||
  class-or-namespace-name `::` *nested-name-specifier*$_{opt}$ ||

class-or-namespace-name: ||
  class-name ||
  namespace-name ||

namespace-name: ||
  original-namespace-name ||
  namespace-alias ||

namespace-alias:
  identifier

namespace-alias-definition:
  `namespace` *identifier* `=` *qualified-namespace-specifier ;*

qualified-namespace-specifier:
  ::$_{opt}$ *nested-name-specifier*$_{opt}$ *class-or-namespace-name*

using-declaration:
  `using ::`$_{opt}$ *nested-name-specifier unqualified-id* `;`
  `using ::` *unqualified-id* `;`

using-directive:
  `using namespace ::`$_{opt}$ *nested-name-specifier*$_{opt}$ *namespace-name ;*

asm-definition: *
  `asm (` *string-literal* `) ;`

linkage-specification:
  `extern` *string-literal* `{` *declaration-seq*$_{opt}$ `}`
  `extern` *string-literal* *declaration*

declaration-seq:
  declaration
  declaration-seq *declaration*

A.7 Declarators [gram.dcl.decl]

init-declarator-list:
  init-declarator
  init-declarator-list `,` *init-declarator*

init-declarator:
  declarator *initializer*$_{opt}$

declarator:
  direct-declarator
  ptr-operator declarator

direct-declarator:
  declarator-id
  direct-declarator `(` *parameter-declaration-clause* `)` *cv-qualifier-seq*$_{opt}$ *exception-specification*$_{opt}$
  direct-declarator `[` *constant-expression*$_{opt}$ `]`
  `(` *declarator* `)`

ptr-operator:
  `*` *cv-qualifier-seq*$_{opt}$
  `&`
  `::`$_{opt}$ *nested-name-specifier* `*` *cv-qualifier-seq*$_{opt}$

cv-qualifier-seq:
  cv-qualifier *cv-qualifier-seq*$_{opt}$

cv-qualifier:
  `const`
  `volatile`

declarator-id:
  id-expression
  nested-name-specifier$_{opt}$ *type-name*

type-id:
  type-specifier-seq *abstract-declarator*$_{opt}$

type-specifier-seq:
  type-specifier type-specifier-seq$_{opt}$

abstract-declarator:
  ptr-operator abstract-declarator$_{opt}$
  direct-abstract-declarator

direct-abstract-declarator:
  direct-abstract-declarator$_{opt}$ `(` *parameter-declaration-clause* `)` *cv-qualifier-seq*$_{opt}$ *exception-specification*$_{opt}$
  direct-abstract-declarator$_{opt}$ `[` *constant-expression*$_{opt}$ `]`
  `(` *abstract-declarator* `)`

parameter-declaration-clause:
    parameter-declaration-list$_{opt}$ `...`$_{opt}$
    parameter-declaration-list `,` `...`

parameter-declaration-list:
    parameter-declaration
    parameter-declaration-list `,` *parameter-declaration*

parameter-declaration:
    decl-specifier-seq declarator
    decl-specifier-seq declarator `=` *assignment-expression*
    decl-specifier-seq abstract-declarator$_{opt}$
    decl-specifier-seq abstract-declarator$_{opt}$ `=` *assignment-expression*

function-definition:
    decl-specifier-seq$_{opt}$ *declarator ctor-initializer*$_{opt}$ *function-body*

function-body:
    compound-statement

initializer:
    `=` *initializer-clause*
    `(` *expression-list* `)`

initializer-clause:
    assignment-expression
    `{` *initializer-list* `,`$_{opt}$ `}`
    `{ }`

initializer-list:
    initializer-clause
    initializer-list `,` *initializer-clause*

A.8 Classes [gram.class]

class-name:
    identifier
    template-id

class-specifier:
    class-head `{` *member-specification*$_{opt}$ `}`

class-head:
    class-key identifier$_{opt}$ *base-clause*$_{opt}$
    class-key nested-name-specifier identifier base-clause$_{opt}$

class-key:
    `class`
    `struct`
    `union`

member-specification:
    member-declaration member-specification$_{opt}$
    access-specifier `:` *member-specification*$_{opt}$

member-declaration:
decl-specifier-seq$_{opt}$ *member-declarator-list*$_{opt}$ `;`
function-definition `;`$_{opt}$
qualified-id `;`
using-declaration

member-declarator-list:
member-declarator
member-declarator-list `,` *member-declarator*

member-declarator:
declarator pure-specifier$_{opt}$
declarator constant-initializer$_{opt}$
identifier$_{opt}$ `:` *constant-expression*

pure-specifier:
`= 0`

constant-initializer:
`=` *constant-expression*

A.9 Derived classes [gram.class.derived]

base-clause:
`:` *base-specifier-list*

base-specifier-list:
base-specifier
base-specifier-list `,` *base-specifier*

base-specifier:
::$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name*
virtual access-specifier$_{opt}$ *::*$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name*
access-specifier virtual$_{opt}$ *::*$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name*

access-specifier:
`private`
`protected`
`public`

A.10 Special member functions [gram.special]

class-name `(` *expression-list*$_{opt}$ `)`

conversion-function-id:
`operator` *conversion-type-id*

conversion-type-id:
type-specifier-seq conversion-declarator$_{opt}$

conversion-declarator:
ptr-operator conversion-declarator$_{opt}$

ctor-initializer:
`:` *mem-initializer-list*

mem-initializer-list:
    mem-initializer
    mem-initializer `,` *mem-initializer-list*

mem-initializer:
    `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name* `(` *expression-list*$_{opt}$ `)`
    identifier `(` *expression-list*$_{opt}$ `)`

A.11 Overloading [gram.over]

operator-function-id: *
    `operator` *operator*

operator: one of

```
new  delete    new[]     delete[]
+    -    *    /    %    ^    &    |    ~
!    =    <    >    +=   -=   *=   /=   %=
^=   &=   |=   <<   >>   >>=  <<=  ==   !=
<=   >=   &&   ||   ++   --   ,    ->*  ->
()   []
```

A.12 Templates [gram.temp]

template-declaration:
    `template` `<` *template-parameter-list* `>` *declaration*

template-parameter-list:
    template-parameter
    template-parameter-list `,` *template-parameter*

template-id:
    template-name `<` *template-argument-list* `>`

template-name:
    identifier

template-argument-list:
    template-argument
    template-argument-list `,` *template-argument*

template-argument:
    assignment-expression
    type-id
    template-name

type-name-declaration:
    `typedef` *qualified-name* `;`

explicit-instantiation:
    `template` *inst* `;`

inst:
    class-key template-id
    type-specifier-seq template-id `(` *parameter-declaration-clause* `)`

specialization:
    declaration

template-parameter:
  type-parameter
  parameter-declaration

type-parameter:
  `class` *identifier*$_{opt}$
  `class` *identifier*$_{opt}$ `=` *type-id*
  `typedef` *identifier*$_{opt}$
  `typedef` *identifier*$_{opt}$ `=` *type-name*
  `template <` *template-parameter-list* `> class` *identifier*$_{opt}$
  `template <` *template-parameter-list* `> class` *identifier*$_{opt}$ `=` *template-name*

A.13 Exception handling [gram.except]

try-block:
  `try` *compound-statement handler-seq*

handler-seq:
  handler handler-seq$_{opt}$

handler:
  `catch (` *exception-declaration* `)` *compound-statement*

exception-declaration:
  type-specifier-seq declarator
  type-specifier-seq abstract-declarator
  type-specifier-seq
  `...`

throw-expression:
  `throw` *assignment-expression*$_{opt}$

exception-specification:
  `throw (` *type-id-list*$_{opt}$ `)`

type-id-list:
  type-id
  type-id-list `,` *type-id*

Annex B (informative) Implementation quantities [limits]

1 Because computers are finite, C++ implementations are inevitably limited in the size of the programs they can successfully process. Every implementation shall

Box 270

This clause is non-normative, which means that this sentence must be restated in elsewhere as a normative requirement on implementations.

document those limitations where known. This documentation may cite fixed limits where they exist, say how to compute variable limits as a function of available resources, or say that fixed limits do not exist or are unknown.

2 The limits may constrain quantities that include those described below or others. The bracketed number following each quantity is recommended as the minimum for that quantity. However, these quantities are only guidelines and do not determine compliance.

— Nesting levels of compound statements, iteration control structures, and selection control structures [256].

— Nesting levels of conditional inclusion [256].

— Pointer, array, and function declarators (in any combination) modifying an arithmetic, structure, union, or incomplete type in a declaration [256].

— Nesting levels of parenthesized expressions within a full expression [256].

— Number of initial characters in an internal identifier or macro name [1 024]. *

— Number of initial characters in an external identifier [1 024].

— External identifiers in one translation unit [65 536].

— Identifiers with block scope declared in one block [1 024].

— Macro identifiers simultaneously defined in one transation unit [65 536].

— Parameters in one function definition [256].

— Arguments in one function call [256].

— Parameters in one macro definition [256].

— Arguments in one macro invocation [256].

— Characters in one logical source line [65 536].

— Characters in a character string literal or wide string literal (after concatenation) [65 536].

— Size of an object [262 144].

Box 271
This is trivial for some implementations to meet and very hard for others.

— Nesting levels for `#include` files [256].

— Case labels for a `switch` statement (excluding those for any nested `switch` statements) [16 384].

— Data members in a single class, structure, or union [16 384].

— Enumeration constants in a single enumeration [4 096].

— Levels of nested class, structure, or union definitions in a single *struct-declaration-list* [256].

— Functions registered by `atexit()`[32].

— Direct and indirect base classes [16 384].

— Direct base classes for a single class [1 024].

— Members declared in a single class [4 096].

— Final overriding virtual functions in a class, accessible or not [16 384].

Box 272
I'm not quite sure what this means, but it was passed in Munich in this form.

— Direct and indirect virtual bases of a class [1 024].

— Static members of a class [1 024].

— Friend declarations in a class [4 096].

— Access control declarations in a class [4 096].

— Member initializers in a constructor definition [6 144].

— Scope qualifications of one identifier [256].

— Nested external specifications [1 024].

— Template arguyments in a template declaration [1 024].

— Handlers per `try` block [256].

— Throw specifications on a single function declaration [256].

Annex C (informative) Compatibility [diff]

1 This Annex summarizes the evolution of C++ since the first edition of *The C++ Programming Language* and explains in detail the differences between C++ and C. Because the C language as described by this International Standard differs from the dialects of Classic C used up till now, we discuss the differences between C++ and ISO C as well as the differences between C++ and Classic C.

2 C++ is based on C (K&R78) and adopts most of the changes specified by the ISO C standard. Converting programs among C++, K&R C, and ISO C may be subject to vicissitudes of expression evaluation. All differences between C++ and ISO C can be diagnosed by a compiler. With the exceptions listed in this Annex, programs that are both C++ and ISO C have the same meaning in both languages.

C.1 Extensions [diff.c]

1 This subclause summarizes the major extensions to C provided by C++.

C.1.1 C++ features available in 1985 [diff.early]

1 This subclause summarizes the extensions to C provided by C++ in the 1985 version of its manual:

2 The types of function parameters can be specified (8.3.5) and will be checked (5.2.2). Type conversions will be performed (5.2.2). This is also in ISO C.

3 Single-precision floating point arithmetic may be used for `float` expressions; 3.7.1 and 4.8. This is also in ISO C.

4 Function names can be overloaded; 13.

5 Operators can be overloaded; 13.4.

6 Functions can be inline substituted; 7.1.2.

7 Data objects can be `const`; 7.1.5. This is also in ISO C.

8 Objects of reference type can be declared; 8.3.2 and 8.5.3.

9 A free store is provided by the `new` and `delete` operators; 5.3.4, 5.3.5.

10 Classes can provide data hiding (11), guaranteed initialization (12.1), user-defined conversions (12.3), and dynamic typing through use of virtual functions (10.3).

11 The name of a class or enumeration is a type name; 9.

12 A pointer to any non-`const` and non-`volatile` object type can be assigned to a `void*`; 4.10. This is also in ISO C.

13 A pointer to function can be assigned to a `void*`; 4.10.

14 A declaration within a block is a statement; 6.7.

15 Anonymous unions can be declared; 9.6.

C.1.2 C++ features added since 1985 [diff.c++]

1 This subclause summarizes the major extensions of C++ since the 1985 version of this manual:

2 A class can have more than one direct base class (multiple inheritance); 10.1.

3 Class members can be `protected`; 11 .

4 Pointers to class members can be declared and used; 8.3.3, 5.5.

5 Operators `new` and `delete` can be overloaded and declared for a class; 5.3.4, 5.3.5, 12.5. This allows the "assignment to `this`" technique for class specific storage management to be removed to the anachronism subclause; C.3.3.

6 Objects can be explicitly destroyed; 12.4.

7 Assignment and initialization are defined as memberwise assignment and initialization; 12.8.

8 The `overload` keyword was made redundant and moved to the anachronism subclause; C.3.

9 General expressions are allowed as initializers for static objects; 8.5.

10 Data objects can be `volatile`; 7.1.5. Also in ISO C.

11 Initializers are allowed for `static` class members; 9.5.

12 Member functions can be `static`; 9.5.

13 Member functions can be `const` and `volatile`; 9.4.1.

14 Linkage to non-C++ program fragments can be explicitly declared; 7.5.

15 Operators `->`, `->*`, and `,` can be overloaded; 13.4.

16 Classes can be abstract; 10.4.

17 Prefix and postfix application of `++` and `--` on a user-defined type can be distinguished.

18 Templates; 14.

19 Exception handling; 15.

20 The `bool` type (3.7.1).

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 3.4 [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 3.5

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 3.7

5 **Change:** C allows "compatible types" in several places, C++ does not
For example, otherwise-identical `struct` types with different tag names are "compatible" in C but are distinctly different types in C++.
Rationale: Stricter type checking is essential for C++.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation The "typesafe linkage" mechanism will find many, but not all, of such problems. Those problems not found by typesafe linkage will continue to function properly, according to the "layout compatibility rules" of this International Standard.
How widely used: Common.

Subclause 4.10

6 **Change:** Converting `void*` to a pointer-to-object type requires casting

```
char a[10];
void *b=a;
void foo() {
char *c=b;
}
```

ISO C will accept this usage of pointer to void being assigned to a pointer to object type. C++ will not.
Rationale: C++ tries harder than C to enforce compile-time type safety.
Effect on original feature: Deletion of semantically well-defined feature.

Difficulty of converting: Could be automated. Violations will be diagnosed by the C++ translator. The fix is to add a cast. For example:

```
char *c = (char *) b;
```

How widely used: This is fairly widely used but it is good programming practice to add the cast when assigning pointer-to-void to pointer-to-object. Some ISO C translators will give a warning if the cast is not used.

Subclause 4.10

7 **Change:** Only pointers to non-const and non-volatile objects may be implicitly converted to `void*`
Rationale: This improves type safety.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Could be automated. A C program containing such an implicit conversion from (e.g.) pointer-to-const-object to void* will receive a diagnostic message. The correction is to add an explicit cast.
How widely used: Seldom.

C.2.3 Clause 5: expressions [diff.expr]

Subclause 5.2.2

1 **Change:** Implicit declaration of functions is not allowed
Rationale: The type-safe nature of C++.
Effect on original feature: Deletion of semantically well-defined feature. Note: the original feature was labeled as "obsolescent" in ISO C.
Difficulty of converting: Syntactic transformation. Facilities for producing explicit function declarations are fairly widespread commercially.
How widely used: Common.

Subclause 5.3.3, 5.4

2 **Change:** Types must be declared in declarations, not in expressions
In C, a sizeof expression or cast expression may create a new type. For example,

```
p = (void*)(struct x {int i;} *)0;
```

declares a new type, struct x .
Rationale: This prohibition helps to clarify the location of declarations in the source code.
Effect on original feature: Deletion of a semantically well-defined feature.
Difficulty of converting: Syntactic transformation.
How widely used: Seldom.

C.2.4 Clause 6: statements [diff.stat]

Subclause 6.4.2, 6.6.4 (`switch` and `goto` statements)

1 **Change:** It is now invalid to jump past a declaration with explicit or implicit initializer (except across entire block not entered)
Rationale: Constructors used in initializers may allocate resources which need to be de-allocated upon leaving the block. Allowing jump past initializers would require complicated run-time determination of allocation. Furthermore, any use of the uninitialized object could be a disaster. With this simple compile-time rule, C++ assures that if an initialized variable is in scope, then it has assuredly been initialized.
Effect on original feature: Deletion of semantically well-defined feature.

Difficulty of converting: Semantic transformation.
How widely used: Seldom.

Subclause 6.6.3

2 **Change:** It is now invalid to return (explicitly or implicitly) from a function which is declared to return a value without actually returning a value
Rationale: The caller and callee may assume fairly elaborate return-value mechanisms for the return of class objects. If some flow paths execute a return without specifying any value, the compiler must embody many more complications. Besides, promising to return a value of a given type, and then not returning such a value, has always been recognized to be a questionable practice, tolerated only because very-old C had no distinction between void functions and int functions.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation. Add an appropriate return value to the source code, e.g. zero.
How widely used: Seldom. For several years, many existing C compilers have produced warnings in this case.

C.2.5 Clause 7: declarations [diff.dcl]

Subclause 7.1.1

1 **Change:** In C++, the `static` or `extern` specifiers can only be applied to names of objects or functions Using these specifiers with type declarations is illegal in C++. In C, these specifiers are ignored when used on type declarations. Example:

```
static struct S {      // valid C, invalid in C++
int i;
// ...
};
```

Rationale: Storage class specifiers don't have any meaning when associated with a type. In C++, class members can be defined with the `static` storage class specifier. Allowing storage class specifiers on type declarations could render the code confusing for users.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Syntactic transformation.
How widely used: Seldom.

Subclause 7.1.3

2 **Change:** A C++ typedef name must be different from any class type name declared in the same scope (except if the typedef is a synonym of the class name with the same name). In C, a typedef name and a struct tag name declared in the same scope can have the same name (because they have different name spaces)
Example:

```
typedef struct name1 { /*...*/ } name1; // valid C and C++
struct name { /*...*/ };
typedef int name;                 // valid C, invalid C++
```

Rationale: For ease of use, C++ doesn't require that a type name be prefixed with the keywords `class`, `struct` or `union` when used in object declarations or type casts. Example:

```
class name { /*...*/ };
name i;                  // i has type 'class name'
```

Effect on original feature: Deletion of semantically well-defined feature.

Difficulty of converting: Semantic transformation. One of the 2 types has to be renamed.
How widely used: Seldom.

Subclause 7.1.5 [see also 3.4]

3 **Change:** const objects must be initialized in C++ but can be left uninitialized in C
Rationale: A const object cannot be assigned to so it must be initialized to hold a useful value.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation.
How widely used: Seldom.

Subclause 7.2

4 **Change:** C++ objects of enumeration type can only be assigned values of the same enumeration type. In C, objects of enumeration type can be assigned values of any integral type
Example:

```
enum color { red, blue, green };
color c = 1;   // valid C, invalid C++
```

Rationale: The type-safe nature of C++.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Syntactic transformation. (The type error produced by the assignment can be automatically corrected by applying an explicit cast.)
How widely used: Common.

Subclause 7.2

5 **Change:** In C++, the type of an enumerator is its enumeration. In C, the type of an enumerator is `int`.
Example:

```
enum e { A };
sizeof(A) == sizeof(int)  // in C
sizeof(A) == sizeof(e)    // in C++
/* and sizeof(int) is not necessary equal to sizeof(e) */
```

Rationale: In C++, an enumeration is a distinct type.
Effect on original feature: Change to semantics of well-defined feature.
Difficulty of converting: Semantic transformation.
How widely used: Seldom. The only time this affects existing C code is when the size of an enumerator is taken. Taking the size of an enumerator is not a common C coding practice.

C.2.6 Clause 8: declarators [diff.decl]

Subclause 8.3.5

1 **Change:** In C++, a function declared with an empty parameter list takes no arguments.
In C, an empty parameter list means that the number and type of the function arguments are unknown"
Example:

```
int f();  // means   int f(void)     in C++
              //         int f(unknown)  in C
```

Rationale: This is to avoid erroneous function calls (i.e. function calls with the wrong number or type of arguments).
Effect on original feature: Change to semantics of well-defined feature. This feature was marked as

"obsolescent" in C.
Difficulty of converting: Syntactic transformation. The function declarations using C incomplete declaration style must be completed to become full prototype declarations. A program may need to be updated further if different calls to the same (non-prototype) function have different numbers of arguments or if the type of corresponding arguments differed.
How widely used: Common.

Subclause 8.3.5 [see 5.3.3]

2 **Change:** In C++, types may not be defined in return or parameter types. In C, these type definitions are allowed
Example:

```
        void f( struct S { int a; } arg ) {}     // valid C, invalid C++
        enum E { A, B, C } f() {}                // valid C, invalid C++
```

Rationale: When comparing types in different compilation units, C++ relies on name equivalence when C relies on structural equivalence. Regarding parameter types: since the type defined in an parameter list would be in the scope of the function, the only legal calls in C++ would be from within the function itself.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation. The type definitions must be moved to file scope, or in header files.
How widely used: Seldom. This style of type definitions is seen as poor coding style.

Subclause 8.4

3 **Change:** In C++, the syntax for function definition excludes the "old-style" C function. In C, "old-style" syntax is allowed, but deprecated as "obsolescent."
Rationale: Prototypes are essential to type safety.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Syntactic transformation.
How widely used: Common in old programs, but already known to be obsolescent.

Subclause 8.5.2

4 **Change:** In C++, when initializing an array of character with a string, the number of characters in the string (including the terminating `'\0'`) must not exceed the number of elements in the array. In C, an array can be initialized with a string even if the array is not large enough to contain the string terminating `'\0'`
Example:

```
        char array[4] = "abcd";    // valid C, invalid C++
```

Rationale: When these non-terminated arrays are manipulated by standard string routines, there is potential for major catastrophe.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation. The arrays must be declared one element bigger to contain the string terminating `'\0'`.
How widely used: Seldom. This style of array initialization is seen as poor coding style.

C.2.7 Clause 9: classes [diff.class]

Subclause 9.1 [see also 7.1.3]

1 **Change:** In C++, a class declaration introduces the class name into the scope where it is declared and hides any object, function or other declaration of that name in an enclosing scope. In C, an inner scope declaration of a struct tag name never hides the name of an object or function in an outer scope
Example:

```
int x[99];
void f()
{
        struct x { int a; };
        sizeof(x);  /* size of the array in C    */
        /* size of the struct in C++ */
}
```

Rationale: This is one of the few incompatibilities between C and C++ that can be attributed to the new C++ name space definition where a name can be declared as a type and as a nontype in a single scope causing the nontype name to hide the type name and requiring that the keywords `class`, `struct`, `union` or `enum` be used to refer to the type name. This new name space definition provides important notational conveniences to C++ programmers and helps making the use of the user-defined types as similar as possible to the use of built-in types. The advantages of the new name space definition were judged to outweigh by far the incompatibility with C described above.
Effect on original feature: Change to semantics of well-defined feature.
Difficulty of converting: Semantic transformation. If the hidden name that needs to be accessed is at global scope, the `::` C++ operator can be used. If the hidden name is at block scope, either the type or the struct tag has to be renamed.
How widely used: Seldom.

Subclause 9.8

2 **Change:** In C++, the name of a nested class is local to its enclosing class. In C the name of the nested class belongs to the same scope as the name of the outermost enclosing class
Example:

```
struct X {
        struct Y { /* ... */ } y;
};
struct Y yy;     // valid C, invalid C++
```

Rationale: C++ classes have member functions which require that classes establish scopes. The C rule would leave classes as an incomplete scope mechanism which would prevent C++ programmers from maintaining locality within a class. A coherent set of scope rules for C++ based on the C rule would be very complicated and C++ programmers would be unable to predict reliably the meanings of nontrivial examples involving nested or local functions.
Effect on original feature: Change of semantics of well-defined feature.
Difficulty of converting: Semantic transformation. To make the struct type name visible in the scope of the enclosing struct, the struct tag could be declared in the scope of the enclosing struct, before the enclosing struct is defined. Example:

```
struct Y;  // struct Y and struct X are at the same scope
struct X {
        struct Y { /* ... */ } y;
};
```

All the definitions of C struct types enclosed in other struct definitions and accessed outside the scope of the enclosing struct could be exported to the scope of the enclosing struct. Note: this is a consequence of the difference in scope rules, which is documented at subclause 3.3 above.
How widely used: Seldom.

Subclause 9.10

3 **Change:** In C++, a typedef name may not be redefined in a class declaration after being used in the declaration
Example:

```
typedef int I;
struct S {
        I i;
        int I;       // valid C, invalid C++
};
```

Rationale: When classes become complicated, allowing such a redefinition after the type has been used can create confusion for C++ programmers as to what the meaning of 'I' really is.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation. Either the type or the struct member has to be renamed.
How widely used: Seldom.

C.2.8 Clause 16: preprocessing directives [diff.cpp]

Subclause 16.8 (predefined names)

1 **Change:** Whether `__STDC__` is defined and if so, what its value is, are implementation-defined
Rationale: C++ is not identical to ISO C. Mandating that `__STDC__` be defined would require that translators make an incorrect claim. Each implementation must choose the behavior that will be most useful to its marketplace.
Effect on original feature: Change to semantics of well-defined feature.
Difficulty of converting: Semantic transformation.
How widely used: Programs and headers that reference `__STDC__` are quite common.

C.3 Anachronisms [diff.anac]

1 The extensions presented here may be provided by an implementation to ease the use of C programs as C++ programs or to provide continuity from earlier C++ implementations. Note that each of these features has undesirable aspects. An implementation providing them should also provide a way for the user to ensure that they do not occur in a source file. A C++ implementation is not obliged to provide these features.

2 The word `overload` may be used as a *decl-specifier* (7) in a function declaration or a function definition. When used as a *decl-specifier*, `overload` is a reserved word and cannot also be used as an identifier.

3 The definition of a static data member of a class for which initialization by default to all zeros applies (8.5, 9.5) may be omitted.

4 An old style (that is, pre-ISO C) C preprocessor may be used.

5 An `int` may be assigned to an object of enumeration type.

6 The number of elements in an array may be specified when deleting an array of a type for which there is no destructor; 5.3.5.

7 A single function `operator++()` may be used to overload both prefix and postfix `++` and a single function `operator--()` may be used to overload both prefix and postfix `--`; 13.4.6.

8

C.3.1 Old style function definitions [diff.fct.def]

1 The C function definition syntax

old-function-definition:
  decl-specifiers$_{opt}$ *old-function-declarator* *declaration-seq*$_{opt}$ *function-body*

old-function-declarator:
    declarator (*parameter-list*$_{opt}$)

parameter-list:
    identifier
    parameter-list , *identifier*

For example,

```
max(a,b) int b; { return (a<b) ? b : a; }
```

may be used. If a function defined like this has not been previously declared its parameter type will be taken to be `(...)`, that is, unchecked. If it has been declared its type must agree with that of the declaration.

2 Class member functions may not be defined with this syntax.

C.3.2 Old style base class initializer [diff.base.init]

1 In a *mem-initializer*(12.6.2), the *class-name* naming a base class may be left out provided there is exactly one immediate base class. For example,

```
class B {
    // ...
public:
    B (int);
};

class D : public B {
    // ...
    D(int i) : (i) { /* ... */ }
};
```

causes the `B` constructor to be called with the argument `i`.

C.3.3 Assignment to `this` [diff.this]

1 Memory management for objects of a specific class can be controlled by the user by suitable assignments to the `this` pointer. By assigning to the `this` pointer before any use of a member, a constructor can implement its own storage allocation. By assigning the null pointer to `this`, a destructor can avoid the standard deallocation operation for objects of its class. Assigning the null pointer to `this` in a destructor also suppressed the implicit calls of destructors for bases and members. For example,

```
class Z {
    int z[10];
    Z()  { this = my_allocator( sizeof(Z) ); }
    ~Z() { my_deallocator( this ); this = 0; }
};
```

2 On entry into a constructor, `this` is nonnull if allocation has already taken place (as it will have for `auto`, `static`, and member objects) and null otherwise.

3 Calls to constructors for a base class and for member objects will take place (only) after an assignment to `this`. If a base class's constructor assigns to `this`, the new value will also be used by the derived class's constructor (if any).

4 Note that if this anachronism exists either the type of the `this` pointer cannot be a `*const` or the enforcement of the rules for assignment to a constant pointer must be subverted for the `this` pointer.

C.3.4 Cast of bound pointer [diff.bound]

1 A pointer to member function for a particular object may be cast into a pointer to function, for example, `(int(*)())p->f`. The result is a pointer to the function that would have been called using that member function for that particular object. Any use of the resulting pointer is – as ever – undefined.

C.3.5 Nonnested classes [diff.class.nonnested]

1 Where a class is declared within another class and no other class of that name is declared in the program that class can be used as if it was declared outside its enclosing class (exactly as a C `struct`). For example,

```
struct S {
    struct T {
        int a;
    };
    int b;
};

struct T x;     // meaning ‘S::T x;’
```

C.4 Standard C library [diff.library]

1 This subclause summarizes the explicit changes in definitions, declarations, or behavior within the Standard C library when it is part of the Standard C++ library. (Subclause _lib.introduction_ imposes some *implicit* changes in the behavior of the Standard C library.)

C.4.1 Modifications to headers [diff.mods.to.headers]

1 For compatibility with the Standard C library, the Standard C++ library provides the 18 *C headers*, as shown in Table 127:

Table 127—C Headers

`<assert.h>`	`<iso646.h>`	`<setjmp.h>`	`<stdio.h>`	`<wchar.h>`
`<ctype.h>`	`<limits.h>`	`<signal.h>`	`<stdlib.h>`	`<wctype.h>`
`<errno.h>`	`<locale.h>`	`<stdarg.h>`	`<string.h>`	
`<float.h>`	`<math.h>`	`<stddef.h>`	`<time.h>`	

2 Each C header, whose name has the form *`name`*`.h`, includes its corresponding C++ header `c`*`name`*, followed by an explicit using-declaration (7.3.3) for each name placed in the standard library namespace by the header (_lib.namespace_).[175)]

C.4.2 Modifications to definitions [diff.mods.to.definitions]

1

175) For example, the header `<cstdlib>` provides its declarations and definitions within the namespace `std`. The header `<stdlib.h>`, makes these available in the global name space, much as in the C Standard.

C.4.2.1 Type wchar_t [diff.wchar.t]

1 wchar_t is a keyword in this International Standard (_lex.key_). It does not appear as a type name defined in any of <cstddef>, <stddef.h>, <cstdlib>, <stdlib.h>, <cwchar>, or <wchar.h>.

C.4.2.2 Header <iso646.h> [diff.header.iso646.h]

1 The tokens and, and_eq, bitand, bitor, compl, not_eq, not, or, or_eq, xor, and xor_eq are keywords in this International Standard (2.8). They do not appear as macro names defined in <iso646.h>.

C.4.2.3 Macro NULL [diff.null]

1 The macro NULL, defined in any of <clocale>, <locale.h>, <cstddef>, <stddef.h>, <cstdio>, <stdio.h>, <cstdlib>, <stdlib.h>, <cstring>, <string.h>, <ctime>, <time.h>, <cwchar>, or <wchar.h>, is an implementation-defined C++ null-pointer constant in this International Standard.[176)]

C.4.3 Modifications to declarations [diff.mods.to.declarations]

1 Header <string.h>: The following functions have different declarations:

— strchr

— strpbrk

— strrchr

— strstr

— memchr

Subclause (21.2) describes the changes.

C.4.4 Modifications to behavior [diff.mods.to.behavior]

1 Header <stdlib.h>: The following functions have different behavior: atexit

— exit

Subclause (18.3) describes the changes.

2 Header <setjmp.h>: The following functions have different behavior: longjmp

Subclause (18.7) describes the changes.

3

C.4.4.1 Macro offsetof(*type*, *member-designator*) <stddef.h> [diff.offsetof]

1 The macro offsetof, defined in <stddef.h>, accepts a restricted set of *type* arguments in this International Standard. *type* shall be a POD structure or a POD union.

C.4.5 Names with external linkage [diff.extern.c.names]

— Each name declared with external linkage in a C header is reserved to the implementation for use as a name with extern "C" linkage.

— Each function signature declared with external linkage in a C header is reserved to the implementation for use as a function signature with both extern "C" and extern "C++" linkage.[177)]

176) Possible definitions include 0 and 0L, but not (void*)0.
177) The function signatures declared in <cwchar> and <cwctype> are always reserved, notwithstanding the restrictions imposed

1 It is unspecified whether a name declared with external linkage in a C header has either `extern "C"` or `extern "C++"` linkage.[178)]

in subclause 4.5.1 of Amendment 1 to the C Standard for these headers.

[178)] The only reliable way to declare an object or function signature from the Standard C library is by including the header that declares it, notwithstanding the latitude granted in subclause 7.1.7 of the C Standard.

Annex D (informative)
Future directions [future.directions]