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.

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.

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; 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.

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*).

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

— *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). 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.[2)]

[2)] 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.

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 (17.1.2) 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.[3)]

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 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.

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

> **Box 2**
> 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

<table>
<tr><th>trigraph</th><th>replacement</th><th>trigraph</th><th>replacement</th><th>trigraph</th><th>replacement</th></tr>
<tr><td><code>??=</code></td><td><code>#</code></td><td><code>??(</code></td><td><code>[</code></td><td><code>??<</code></td><td><code>{</code></td></tr>
<tr><td><code>??/</code></td><td><code>\</code></td><td><code>??)</code></td><td><code>]</code></td><td><code>??></code></td><td><code>}</code></td></tr>
<tr><td><code>??'</code></td><td><code>^</code></td><td><code>??!</code></td><td><code>|</code></td><td><code>??-</code></td><td><code>~</code></td></tr>
</table>

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 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

<table>
<tr><th>alternate</th><th>primary</th><th>alternate</th><th>primary</th><th>alternate</th><th>primary</th></tr>
<tr><td><code><%</code></td><td><code>{</code></td><td><code>and</code></td><td><code>&&</code></td><td><code>and_eq</code></td><td><code>&=</code></td></tr>
<tr><td><code>%></code></td><td><code>}</code></td><td><code>bitor</code></td><td><code>|</code></td><td><code>or_eq</code></td><td><code>|=</code></td></tr>
<tr><td><code><:</code></td><td><code>[</code></td><td><code>or</code></td><td><code>||</code></td><td><code>xor_eq</code></td><td><code>^=</code></td></tr>
<tr><td><code>:></code></td><td><code>]</code></td><td><code>xor</code></td><td><code>^</code></td><td><code>not</code></td><td><code>!</code></td></tr>
<tr><td><code>%:</code></td><td><code>#</code></td><td><code>compl</code></td><td><code>~</code></td><td><code>not_eq</code></td><td><code>!=</code></td></tr>
<tr><td><code>bitand</code></td><td><code>&</code></td><td></td><td></td><td></td><td></td></tr>
</table>

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 character, the next token is taken to be the longest string of characters 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 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 (_ _) 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>
<tr><td>!</td><td>%</td><td>^</td><td>&</td><td>*</td><td>(</td><td>)</td><td>-</td><td>+</td><td>=</td><td>{</td><td>}</td><td>|</td><td>~</td></tr>
<tr><td>[</td><td>]</td><td>\</td><td>;</td><td>'</td><td>:</td><td>"</td><td><</td><td>></td><td>?</td><td>,</td><td>.</td><td>/</td><td></td></tr>
</table>

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

Table 6—character combinations used as operators

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

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-digit octal-digit*
  `\` *octal-digit octal-digit 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.

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, two, or three 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 a sequence of hexadecimal digits that are taken to specify the value of the desired character. There is no limit to the number of hexadecimal digits in the 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`.

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 `char`" and *static* storage duration (3.6), 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 `wchar_t`." Concatenation of ordinary and wide-character string literals is undefined.

Box 3
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 `\`.

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, 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 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.

9 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 4
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 (17.5.1.1)
};

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() { }
};
```

3.2 One definition rule [basic.def.odr]

Box 5
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.

Box 6
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 7
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 type.

Box 8
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 9
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 values of the enumerators are the same.

Box 10
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 11
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 (3.3.10) 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 in a declaration in the *condition* part of an `if`, `while`, `for`, `do`, or `switch` statement are local to the controlled statement and shall not be redeclared in the outermost block of that statement.

3.3.2 Function prototype scope [basic.scope.proto]

1 In a function declaration, 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 File scope [basic.file.scope]

1 A name declared outside all named namespaces (_namespace_), blocks (6.3) and classes (9) has *file scope*. The potential scope of such a name begins at its point of declaration (3.3.10) and ends at the end of the translation unit that is its declarative region. Names declared with file scope are said to be *global*.

2 File scope can be treated as a special case of namespace scope (3.3.5) by viewing an entire translation unit as an unnamed namespace called the *global namespace*.

3.3.5 Namespace scope [basic.scope.namespace]

1 A name declared in a namespace (_namespace_) has namespace scope. Its potential scope includes its namespace from the name's point of declaration (3.3.10) onwards, as well as the potential scope of any *using directive* (7.3.4) that nominates its namespace. A namespace member can be also be used after the `::` scope resolution operator (5.1) applied to the name of its namespace.

2 A function may be defined only in namespace or class scope.

3.3.6 Class scope [basic.scope.class]

1 The name of a class member is local to its class and can be used only in a member of that class (9.4) or a class derived from that class, after the `.` operator applied to an expression of the type of its class (5.2.4) or a class derived from (10) its class, after the `->` operator applied to a pointer to an object of its class (5.2.4) or a class derived from (10) 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 directive* (7.3.4).

Box 12
What does: "can be used only in a member of that class" mean? It should be phrased to include: body of member functions, ctor-init-list, static member initializers.

2 The scope of names introduced by friend declarations is described in 7.3.1.

3 A function may be defined only in namespace or class scope.

4 The scope rules for classes are summarized in 9.3.

3.3.7 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) may be hidden by the name of an object, function, or enumerator declared in the same scope. If a class and an object, function, or enumerator are declared in the same scope (in any order) with the same name the class name is hidden.

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".

3.3.8 Explicit qualification [basic.scope.exqual]

Box 14
The information in this section is very similar to the one provided in 7.3.5. The information in these two sections (3.3.8 and 7.3.5) should be consolidated in one place.

1 A hidden name 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).

3.3.9 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).

3.3.10 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. The process of linking together translation units. 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 is said to be ''of namespace scope'' if its immediate scope is the file scope or the scope of a named or unnamed namespace.

> **Box 16**
> The definition of ''of namespace scope'' should probably appear elsewhere.

A name of namespace scope has internal linkage if it is the name of

— a variable that is explicitly declared `static` or is explicitly declared `const` and not explicitly declared `extern`; or

— a function that is explicitly declared `static` or is explicitly declared `inline` and not explicitly declared `extern`. In addition, the name of a data member of an anonymous union declared at namespace scope has internal linkage.

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 that has any static data members (9.5), any member functions that are not defined within the class definition and are not explicitly declared `inline` (9.4.2), or any member types with external linkage; or

— a template (14). Moreover, the name of a class (9) or enumeration (7.2) has external linkage if it is used to declare a function, variable, or type with external linkage, to declare a template, or to specify a template argument. Using a class object in a *throw-expression* does not affect the linkage of the class.

> **Box 17**
> This was voted in San Diego but was probably a mistake. There can, after all, be no issue of C compatibility where exceptions are involved. Moreover, this treatment creates a bad pitfall:

```
// file a.h
struct A { };

// file main.c
#include "a.h"
extern void f();

main()
{
    try {
        f();
    } catch (A) {
    }
}

// file f.c
void f() { throw A(); }
```

> 5 It is reasonable to expect that the throw and the catch refer to the same type, but according to the San Diego resolutions they don't.

The names of class members and enumerators has external linkage if the class or enumeration to which they belong has external linkage.

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

7 Names not covered by these rules have no linkage. Moreover, except as noted, a name declared in a local scope (3.3.1) has no linkage and shall not be used in a way that also requires it to have external linkage. For example:

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

Here, there are conflicting constraints on `A`: its use as the type of an object with external linkage requires it to have external linkage, but because it is declared in a local scope, it has no linkage.

8 Two names are the same 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.

Box 18

A definition of name-sameness should probably appear elsewhere, since it is also assumed in [basic.scope.hiding].

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

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

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

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

9 Inline class member functions must have exactly one definition in a program. *

Box 19

To be reworked when the ODR is clarified.

10 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 20

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

11 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 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 as zero-terminated strings in `argv[0]` through `argv[argc-1]` and `argv[0]` shall be 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`.

4 Calling the function

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

declared in <cstdlib> (17.2.4.5) 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 21
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 dynamic (that is, run-time) initialization. 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()` (_exccept.terminate_).

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()` (17.2.4.5). 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 _stmt.decl_.

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

Box 22
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` (17.3.3.4) and `operator new[]` (17.3.3.5), and the global *deallocation functions* `operator delete` (17.3.3.2) and `operator delete[]` (17.3.3.3).

2 These functions are always implicitly declared. The library provides default definitions for them (17.3.3). 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>` (17.3).

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 is requested is zero, the value returned shall be nonzero and disjoint from any other currently allocated storage. The results of dereferencing a pointer returned as a request for zero size are undefined.[4)]

3 If an allocation function is unable to obtain an appropriate block of storage, it may invoke the currently installed `new_handler`[5)] and/or throw an exception (15) of class `alloc` (17.3.2.9) 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. 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) 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.

5) A program-supplied allocation function may obtain the address of the currently installed `new_handler` using the `set_new_handler()` function (17.3.3.1).

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.[6)]

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 Except in the case of a reference declaration initialised by an rvalue (8.5.3), a reference may be used to name an existing object denoted by an lvalue.

2 The reference has static storage duration if it is declared non-locally, automatic storage duration if declared locally including as a function parameter, and inherited storage duration 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.

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

Box 23
Can references be declared auto or static? This section probably does not belong here.

3.7 Types [basic.types]

Box 24
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

[6)] On some architectures, it causes a system-generated runtime fault.

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. 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` consume the same amount of space.

3 An *enumeration* comprises a set of named integer constant 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.[7] 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] See 7.1.5.2 regarding the correspondence between types and the sequences of *type-specifier*s that designate them.

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 (17.5.9.1). 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`.[8] 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 (4.1, 5.2.3). Although values of type `bool` generally behave as signed integers, for example by promoting (4.1) 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 standard header `<cfloat>`.

10 Types `bool`, `char`, 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 (4.1) to signed or unsigned `int`. *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*[9] *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:[10] 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.[11] A compound type is not

---

[8] 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`.

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

[10] See 8.3.4 and 8.3.5 regarding cv-qualified array and function types.

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

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.

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.

3.7.3 CV-qualifiers [basic.type.qualifier]

Box 25

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-qualifier*s, `const` and `volatile`. When applied to an object, `const` means the program may not change the object, and `volatile` has an implementation-defined meaning.[12)] 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*
}
```

12) 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).

3.7.4 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).

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.[13)]

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 Rvalues may be qualified types, however the unqualified type is used unless the rvalue is of class type and a member function is called on the rvalue. |

8 Whenever an lvalue that refers to a non-array[14)] non-class object appears in a context where an lvalue is not expected, the value contained in the referenced object is used. When this occurs, the value has the unqualified type of the lvalue. For example:

```
const int* cip;
int i = *cip   // "*cip" has type int
```

If this type is incomplete, the program is ill-formed. *


```
const int* cip;         int i = *cip   // "*cip" has type int
```


When an lvalue is used as the operand of `sizeof` the value contained in the referenced object is *not* accessed, since that operator does not evaluate its operand. |

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

Box 26
Provide example and cross-reference.

|

13) 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. |

14) An lvalue that refers to an array object is usually converted to a (rvalue) pointer to the initial element of the array (4.6). |

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.

4 Standard conversions [conv]

1 Some operators may, depending on their operands, cause conversion of the value of an operand from one type to another. This section summarizes the conversions demanded by most ordinary operators and explains the result to be expected from such conversions; it will be supplemented as required by the discussion of each operator. These conversions are also used in initialization (8.5, 8.5.3, 12.8, 12.1). 12.3 and 13.2 describe user-defined conversions and their interaction with standard conversions. The result of a conversion is an lvalue only if the result is a reference (8.3.2).

4.1 Integral promotions [conv.prom]

1 A `char`, `wchar_t`, `bool`, `short int`, enumerator, object of enumeration type (7.2), or an `int` bit-field (9.7) (in both their signed and unsigned varieties) may be used wherever an integer rvalue may be used. In contexts where a constant integer is required, the `bool`, `char`, `wchar_t`, `short int`, object of enumeration type (7.2), or bit-field must be constant. (Enumerators are always constant).

2 Except for enumerators, objects of enumeration type, and type `wchar_t`, if an `int` can represent all the values of the original type, the value is converted to `int`; otherwise it is converted to `unsigned int`.

3 For enumerators, objects of enumeration type, and type `wchar_t`, if an `int` can represent all the values of the underlying type, the value is converted to an `int`; otherwise if an `unsigned int` can represent all the values, the value is converted to an `unsigned int`; otherwise, if a `long` can represent all the values, the value is converted to a `long`; otherwise it is converted to `unsigned long`.

4 A Boolean value may be converted to `int`, taking `false` to zero and `true` to one.

5 This process is called *integral promotion*.

4.2 Integral conversions [conv.integral]

1 An integer rvalue may be converted to any integral type. If the target 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.

2 When an integer is converted to a signed type, the value is unchanged if it can be represented in the new type; otherwise the value is implementation dependent.

3 When an integer is converted to `bool`, see 4.9.

4.3 Float and double [conv.double]

1 Single-precision floating point arithmetic may be used for `float` expressions. When a less precise floating value is converted to an equally or more precise floating type, the value is unchanged. When a more precise floating value is converted to a less precise floating type and the value is within representable range, the result may be either the next higher or the next lower representable value. If the result is out of range, the behavior is undefined.

4.4 Floating and integral [conv.float]

1 Conversion of a floating value to an integral type truncates; that is, the fractional part is discarded. Such conversions are machine dependent; for example, the direction of truncation of negative numbers varies from machine to machine. The result is undefined if the value cannot be represented in the integral type.

2 Conversions of integral values to floating type are as mathematically correct as the hardware allows. Loss of precision occurs if an integral value cannot be represented exactly as a value of the floating type.

4.5 Arithmetic conversions [conv.arith]

1 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."

2

- 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 (4.1) 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`.

4.6 Pointer conversions [conv.ptr]

1 The following conversions may be performed wherever pointers (8.3.1) are assigned, initialized, compared, or otherwise used:

- A constant expression (5.19) that evaluates to zero (the null pointer constant) when assigned to, compared with, alternated with (5.16), or used as an initializer of an operand of pointer type is converted to a pointer of that type. It is guaranteed that this value will produce a pointer distinguishable from a pointer to any object or function.
- A pointer to a cv-qualified or unqualified object type may be converted to a pointer to the same type with greater cv-qualifications (3.7.3). That is, for any unqualified type `T`, a `T*` may be converted to a `const T*`, a `volatile T*`, or a `const volatile T*`; a `const T*` may be converted to a `const volatile T*`; or a `volatile T*` may be converted to a `const volatile T*`.
- A pointer to any object type may be converted to a `void*` with the greater or equal cv-qualifications (3.7.3). That is, for any unqualified type `T`, a `T*` may be converted to a `void*`, a `const void*`, a `volatile void*`, or a `const volatile void*`; a `const T*` may be converted to a `const void*` or a `const volatile void*`; a `volatile T*` may be converted to a `volatile void*` or a `const volatile void*`; and a `const volatile T*` may be converted to a `const volatile void*`.
- Two pointer types T1 and T2 are *similar* if there exists a type T and integer $N > 0$ such that:

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

  and

$T2$ 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 $T1$ may 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 $0 < k < j$.

— A pointer to function may be converted to a `void*` provided a `void*` has sufficient bits to hold it.

— A pointer to a cv-qualified or unqualified class type may be converted to a pointer to an accessible[15] base class type (10) with greater or equal cv-qualifications (3.7.3) provided the conversion is unambiguous (_class.ambig_); a base class is accessible if its public members are accessible (11.1). If `D` is a derived class type and `B` one of its unambiguous base classes, a `D*` may be converted to a `B*`, a `const B*`, a `volatile B*`, or a `const volatile B*`; a `const D*` may be converted to a `const B*`, or a `const volatile B*`; a `volatile D*` may be converted to a `volatile B*`, or a `const volatile B*`; or a `const volatile D*` may be converted to a `const volatile B*`. The result of the conversion is a pointer to the base class sub-object of the derived class object. The null pointer (`0`) is converted into itself.

— An expression with type "array of `T`" is converted to a pointer to the initial element of the array (5) except when used as the operand of the address-of operator `&` or the `sizeof` operator.

— An expression with type "function returning `T`" is converted to "pointer to function returning `T`" except when used as the operand of the address-of operator `&` or the function call operator `()` or the `sizeof` operator, or when the expression refers to a non-static member function.

— A pointer may be converted to type `bool`, see 4.9.

4.7 Reference conversions [conv.ref]

1 The following conversion may be performed wherever references (8.3.2) are initialized (including argument passing (5.2.2) and function value return (6.6.3)):

— An lvalue of a cv-qualified or unqualified object type may be converted to a reference to the same type with increased cv-qualifications.

— An lvalue of cv-qualified or unqualified class type may be converted to a reference to an accessible base class type (10) with greater or equal cv-qualifications (3.7.3) provided the conversion is unambiguous (_class.ambig_); base class is accessible if its public members are accessible (11.1). If `D` is a derived class type and `B` one of its unambiguous base classes, an lvalue of type `D` may be converted to a `B&`, a `const B&`, a `volatile B&`, or a `const volatile B&`; an lvalue of type `const D` may be converted to a `const B&`, or a `const volatile B&`; an lvalue of type `volatile D` may be converted to a `volatile B&`, or a `const volatile B&`; or an lvalue of type `const volatile D` may be converted to a `const volatile B&`. The result of the conversion is a reference to the base class sub-object of the derived class object.

4.8 Pointers to members [conv.mem]

1 The following conversion may be performed wherever pointers to members (8.3.3) are initialized, assigned, compared, or otherwise used:

— A constant expression (5.19) that evaluates to zero is converted to a pointer to member. It is guaranteed that this value will produce a pointer to member distinguishable from any other pointer to

[15] A pointer to a class may be explicitly converted to a pointer to a base class, regardless of accessibility, using a cast (5.2.3 or 5.4).

member.

— A pointer to member of type `T1 B::*` can be converted to a pointer to member of type `T2 D::*` provided the class `B` is an accessible base class of class `D` (11.1), provided the (inverse) conversion from a pointer to `D` to a pointer to base class `B` can be done unambiguously (_class.ambig_), and provided that `T1` and `T2` are the same type or differ only in that `T2` has greater cv-qualifications than `T1` (3.7.3). The result of this conversion refers to the same member as the pointer to member before the conversion took place, but refers to the member as a member of the derived class `D`. That is, the result of this conversion refers to the member in `D`'s instance of `B`.

2 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) (4.6, 10). This inversion is necessary to ensure type safety.

3 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*`.

4 A pointer to member may be converted to type `bool`, see 4.9.

4.9 Boolean conversions [conv.bool]

1 Conversion to `bool` is required in several contexts, such as initializing a `bool` variable, or in the *condition* of an `if` or `while` statement or the first operand of the `?:` operator.

2 In all such cases, the expression to be converted must be of arithmetic, pointer, or pointer to member type or of a class type for which only one unambiguous conversion exists to arithmetic, pointer, pointer to member, or `bool`. Otherwise, the program is ill-formed.

3 A zero value (or a pointer that would compare equal to zero) becomes `false`; any other value becomes `true`.

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).[16)]

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 `const T* volatile` can be used where a `T*` is required. Such uses do not count as standard conversions when considering overloading resolution (13.2).

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, as an initializer (8.5), as the controlling expression in a selection (6.4) or iteration (6.5) statement, as a function return value (6.6.3), or as a function argument (5.2.2).

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

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 27
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 28
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 type of the result is “`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[17] (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).

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

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 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`, enumeration, or a bit-field type for which there is no parameter are converted to `int` or `unsigned` by integral promotion (4.1). An object of a class for which no parameter is declared is passed as a data structure.

Box 29
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 Recursive calls are permitted.

10 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;[18] the result of that evaluation,

[18] 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.

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)[19].

3 If the *id-expression* is a *qualified-id*, the class specified by the the *nested-name-specifier* of the *qualified-id* 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. If the *nested-name-specifier* contains a *template-class-id* (_temp.class_), its *template-argument*s are evaluated in 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.

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 "cv-qualifier 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 may be used only as the left-hand operand of a member function call (9.4) or as the operand of the parenthesis operator (13.4.4). 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`".

[19] Note that if `E1` has the type "pointer to class `X`", then `(*(E1))` is an lvalue.

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 the value of the operand. The operand must be a modifiable lvalue. The type of the operand must be an arithmetic type or a pointer to object type. After the result is noted, the object is incremented by `1`, 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 must be a pointer or a reference to a complete class type or “pointer to *cv* `void`”. *The type of* `v` *must be a complete pointer type if* `T` *is a pointer, or a complete reference type if* `T` *is a reference.*

2 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[20] `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 `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 umambiguous 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*.

5 The value of a failed cast to pointer type is the null pointer. A failed cast to reference type throws `bad_cast` (17.3). For example,

[20] The complete object pointed or refereed 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&` (17.3). 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 (17.3). 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`. Any type conversion not mentioned below and not explicitly defined by the user (12.3) is ill-formed.

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

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

4 A pointer to a complete class `B` may be explicitly converted to a pointer to a complete class `D` that has `B` as a direct or indirect base class if an unambiguous conversion from `D` to `B` exists (4.6, _class.ambig_) and if `B` is not a virtual base class (10.1). Such a cast from a base to a derived class is valid only if the pointer points to an object of the base class that is actually a sub-object of an object of the derived class; the resulting pointer points to the enclosing object of the derived class. Otherwise (the object of the base class is not a sub-object of an object of the derived class) the result of the cast is undefined.

Box 30
The two proposals differed in the preceding behavior. We believe this is the intended behavior;

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

5 Additional conversions that may be performed explicitly using `static_cast` are listed below. No other conversions may be performed explicitly using `static_cast.`

6 A value of integral type may be explicitly converted to an enumeration type. The result of the conversion will compare equal to the integral value provided that the value is within the range of the enumeration's underlying type (7.2). Otherwise, the result is undefined.

7 A "pointer to member of `class A` of type `T1`" may be explicitly converted to a "pointer to member of `class B` of type `T2`" when `class A` and `class B` are either the same class or one is is unambiguously derived from the other (4.6), and the types `T1` and `T2` are the same.

Box 31
The proposal implied the above without direct statement. Check this.

The effect of calling a member function through a pointer to member function type that differs from the type used in the definition of the member function is undefined.

8 The effect of calling a member function through a pointer to member function type that differs from the type used in the definition of the member function is undefined.

9 An lvalue expression of type "`T`" may be explicitly converted to the type "reference to `X`" if an expression of type "pointer to `T`" may be explicitly converted to the type "pointer to `X`" with a `static_cast`. The implementation shall not copy a sub-object to bind a reference; for example,

```
struct B {};
struct D : public B {};
const B &r = D();  // copying only B sub-object not allowed
```

Box 32
Issue (core#1, editorial): 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`".

Constructors or conversion functions are not called as the result of a cast to a reference. Conversion of a reference to a base class to a reference to a derived class is exactly analogous to the conversion of a pointer to a base class to a pointer to a derived class, with respect to restrictions and semantics.

10 The result of a cast to a reference type is an lvalue; the results of other casts are not. Operations performed on the result of a pointer or reference cast refer to the same object as the original (uncast) expression.

11 An expression may be converted to a class type (only) if an appropriate constructor or conversion operator has been declared; see12.3.

12 If a null pointer value is converted to a type "pointer to `T`", the resulting pointer value is a null pointer value.

13 In the description of types, the notation *cv* represents a set of cv-qualifiers (one of {`const`}, {`volatile`}, {`const, volatile`}, or the empty set).

Box 33
This probably should be moved to the discussion of types.

14 Any expression may be explicitly converted to type "*cv* `void`."

Box 34
We believe this was the intent; check this.

15 The following rules define casting away constness. In these rules `Tn` and `Xn` represent types. For two pointer types:

```
X1 = T1 cv11 * cv12 * ... cv1N *   where T1 is not a pointer type and
X2 = T2 cv21 * cv22 * ... cv2M *   where T2 is not a pointer type and
K is the minimum of N and M,
```

Box 35
Editor: re-format this into subscripts, etc.

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:

```
T cv1(N-K+1) * cv1(N-K+2) * ... cv1N *   to
T cv2(N-K+1) * cv2(M-K+2) * ... cv2M *
```

16 Casting from a type "reference to `T1`" to "reference to `T2`" casts away constness if a cast from "pointer to `T1`" to "pointer to `T2`" casts away constness.

17 Casting from "pointer to `C1` member of type `T1`" to "pointer to `C2` member of type `T2`" casts away constness if a cast from "pointer to `T1`" to "pointer to `T2`" casts away constness.

18 For `static_cast` or `const_cast`, `N` and `M` must be equal, otherwise a `reinterpret_cast` is required. Note that these rules are not intended to protect constness in all cases -- in particular, conversions between pointers to functions are not covered because such conversions lead to values whose use causes undefined behavior.

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

2 Conversions that can be performed explicitly using `reinterpret_cast` are listed below. The mapping performed by `reinterpret_cast` is implementation-defined; it may, or may not, produce a representation different from the original value. *

3 The `reinterpret_cast` operator cannot cast away constness; see `static_cast` (_expr.static.cast_). |

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 An incomplete class can be used in a pointer cast. If there is any inheritance relationship between the source and target classes, the behavior is undefined. |

7 A pointer to function may be explicitly converted to a pointer to an object type provided the object pointer type has enough bits to hold the function pointer. A pointer to an object type may be explicitly converted to a pointer to function provided the function pointer type has enough bits to hold the object pointer. In both cases, use of the resulting pointer may cause addressing exceptions if the subject pointer does not refer to suitable storage.

8 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 4.6.

9 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 a pointer into a pointer to a smaller object and back to its original type will yield the original pointer.

10 A "pointer to member of `class A` of type `T1`" may be explicitly converted to a "pointer to member of `class B` of type `T2`" when `class A` and `class B` are either the same class or one is is unambiguously derived from the other (4.6), and the types `T1` and `T2` differ. (The case when `T1` and `T2` are the same type is covered by `static_cast`, (5.2.8).

11 The effect of calling a member function through a pointer to member function type that differs from the type used in the definition of the member function is undefined.

12 If a null pointer value is converted to a type "pointer to `T`", the resulting pointer value is a null pointer value.

13 An lvalue expression of type " `T`" may be explicitly converted to the type "reference to `X`" if an expression of type "pointer to `T`" may be explicitly converted to the type "pointer to `X`" using `reinterpret_cast`. Constructors or conversion functions are not called as the result of a cast to a reference. Conversion of a reference to a base class to a reference to a derived class is exactly analogous to the conversion of a pointer to a base class to a pointer to a derived class, with respect to restrictions and semantics.

14 The result of a cast to a reference type is an lvalue; the results of other casts are not. Operations performed on the result of a pointer or reference cast refer to the same object as the original (uncast) expression.

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

2 A pointer or reference to any object type, or a pointer to data member may be explicitly converted to a type that is identical except for `const` and `volatile` qualifiers. For pointers and references, the result will refer to the original object. For pointers to data members, the result will refer to the same member as the original (uncast) pointer to data member. Depending on the type of the referenced object, a write operation through the resulting pointer, reference or pointer to data member may produce undefined behavior (7.1.5).

3 If a null pointer value is converted to a type "pointer to `T`", the resulting pointer value is a null pointer value.

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`."

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).

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

6 The operand of the unary `-` operator must have arithmetic type and the result is the negation of its operand. Integral promotion is performed on integral 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`(4.9); 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 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 incremented by `1`, or set to `true` if it is `bool` (this use is deprecated). The operand must be a modifiable lvalue. The type of the operand must 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 name. 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)` is implementation-defined. [21)]

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 Types may not be defined in a `sizeof` expression.

5 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

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.

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

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 a constant integral 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 (17.3.3.4) and `operator new[]()` for arrays (17.3.3.5). A C++ program may provide alternative definitions of these functions (17.1.5.4), 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).[22] If the class does not have a default

[22] This means that `struct s{}; s x; s y(x);` is allowed on the grounds that `class s` has an implicitly declared copy constructor, to which the argument `x` is being provided.

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.[23] 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, 17.3.2.9). 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 36

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. *

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 37

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 and the class of the complete object has a destructor (12.4), 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.

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

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 (17.3.3.2) and `operator delete[]()` for arrays (17.3.3.3). A C++ program may provide alternative definitions of these functions (17.1.5.4), 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`, `reinterpret_cast`, `const_cast`, 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 (_class.ambig_). 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 an `.*` expression or a `->*` expression is an lvalue only if its first operand is an lvalue and its second operand refers to an lvalue.

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 (4.5) 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 (4.5) 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 (17.3). 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.[24)]

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.

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

7 When viewed in this way, an implementation need only provide one extra byte (which 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.

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 a 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 dependent.

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 (4.8) are performed. A pointer to member may be compared to a constant expression that evaluates to zero.

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`(4.9). 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`(4.9). 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`(4.9). 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 a constant expression that evaluates to zero, pointer conversions (4.6) 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 a constant expression that evaluates to zero, pointer to member conversions (4.8) are performed to bring them to a common type[25] 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 (4.7). Otherwise, if both the second and the third expressions have type “*cv* `void`”, the common type is “*cv* `void`.” Otherwise, if both the second and the third expressions are of the same class `T`, the common type is `T`. 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.

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

5.17 Assignment operators [expr.ass]

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

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

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

2 In simple assignment (=), the value of the expression replaces that of the object referred to by the left operand. If both operands have arithmetic type, the right operand is converted to the type of the left preparatory to the assignment. There is no implicit conversion to an enumeration (7.2), so if the left operand is of an enumeration type the right operand must be of the same type. If the left operand is of pointer type, the right operand must be the null pointer (4.6) or of a type that can be converted to the type of the left operand, which conversion takes place before the assignment.

3 An expression of type "pointer to *cv1* `T`" can be assigned to a pointer of type "pointer to *cv2* `T`" if the set of cv-qualifiers *cv1* is a subset of *cv2* (7.1.5 see also 8.5).

4 If the left operand is of pointer to member type, the right operand must be of pointer to member type or a constant expression that evaluates to zero; the right operand is converted to the type of the left before the assignment.

5 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 (4.6) can be assigned to an `X`.

6 A pointer to a member of class `B` may be assigned to a pointer to a member of class `D` of the same type provided `D` is derived from `B` (directly or indirectly) by unambiguous public derivation (_class.ambig_).

7 Assignment to an object of type "reference to `T`" assigns to the object of type `T` denoted by the reference.

8 If `E1` is not of type `bool`, 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. In += and −=, the left operand may be a pointer to completely defined object type, in which case the (integral) right operand is converted as explained in 5.7; all right operands and all nonpointer left operands must have arithmetic type.

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

10 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

A *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, and references cannot be used. The comma operator and *assignment-operator*s may not be used in a constant expression. *

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 may 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 may 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)
    for (int i;;) {
        // ...
    }
```

may be equivalently rewritten as

```
if (x) {
    for (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* may 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)
    for (int i;;) {
        // ...
    }
```

may be equivalently rewritten as

```
while (x) {
    for (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`(4.9).

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`(4.9).

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`(4.9).

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 A local object with static storage duration (3.6.1) 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. Where a variable with static storage duration is initialized with an expression that is not a *constant-expression*, default initialization to zero of the appropriate type (8.5) happens before its block is first entered.

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). 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-specifer,*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 (3.3.6) has internal linkage. For a nonmember function, an `inline` specifier is equivalent to a `static` specifier for linkage purposes (3.4).

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 at file scope with the `extern` specifier has external linkage. An object or function declared at block scope with the `extern` specifier has external linkage unless the declaration matches a previous file scope declaration 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 file scope declaration.[26]

6 A name declared at file 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
    { /* ... */ }

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
```

[26] Here, ''previously'' includes enclosing scopes. This implies that a name specified `static` and then specified `extern` in an inner scope still has internal 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. For a nonmember function, the `inline` specifier also gives the function internal linkage (3.4). A function (5.2.2, 8.3.5) defined within the declaration of a class is inline by default.

3 An inline member function must have exactly the same definition in every compilation in which it appears.

4 A class member function need not be explicitly declared with the `inline` specifier in the class declaration to be inline. When no `inline` specifier is used, linkage will be external unless a definition with the `inline` specifer appears before the first call.

```
class X {
public:
    int f();
    inline int g();  // X::g() has internal linkage
    int h();
};

void k(X* p)
{
    int i = p->f();  // now X::f() has external linkage
    int j = p->g();
    // ...
}
```

```
inline int X::f()     // error: called before defined
                      // as inline
{
    // ...
}

inline int X::g()
{
    // ...
}

inline int X::h()    // now X::h() has internal linkage
{
    // ...
}
```

5 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. 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.[27] Regarding the prohibition of the default `int` specifier in `typedef`

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

declarations, see _typedef_; 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 38

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 a constant expression may be used in 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.

Box 39

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-specifer*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-specifer* 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 refering 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 file scope
        struct Data* Data;      // ok: Declares type Data
                                // at file scope and member Data
};

struct Data {
        struct Node* Node;      // ok: Refers to Node at file 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 file 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`,[28)] 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 (4.1). 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,

```
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 40
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

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

(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 a kind of declarative region that effectively attaches an additional identifier to any names declared inside it. Unlike other declarative regions, the definition of a namespace may be split over several parts of a single translation unit.

2 The declarations in file scope of a translation unit behave as if they appeared in a namespace called the *global namespace*.

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 either at file scope or immediately within another *namespace-definition*.

5 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.[29] 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 f() { 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
}
```

6 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; }
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
}
```

7 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 may be nested. For example:

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

8 The use of the `static` keyword is deprecated when declaring objects in a namespace scope (see _future.directions_); the *unnamed-namespace* provides a superior alternative. |

9 Members of a namespace may be defined within that namespace. For example:

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

10 Members of a named namespace may also be defined outside that namespace by explicit qualification (7.3.5) 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();
  }
}
```

11 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;

main() {
  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). |

12 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 non-class namespace. For example:

```
namespace X {
     void p() {
       q();                  // error: q not yet declared
       extern void q();      // q is a member of namespace X
     }
     void q() { }            // definition of q
}
void q() { }                 // some other, unrelated q
```

13

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;    // 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 it 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 may be used in a class definition. For example:

```
struct Base {
        void f(char);
        void g(char);
};
struct Derived: Base
{
  using Base::f;
  void f(int) { f('c'); } // calls Base::f(char)
  void g(int) { g('c'); } // recursively calls Derived::g(int)
};
```

4 An entity with the name of the *unqualified-id* shall be known to the nominated class or namespace at the point that the *using-declaration* appears. Additional definitions added to the namespace after the *using-declaration* are not considered when a use of the name is made.

Box 42
Please check this example carefully.

For example:

```
namespace A {
    void f(int);
}

using A::f;              // f is a synonym for A::f
namespace A {
        void f(char);
}

void foo() {
    f('a');              // calls f(int),
}                        // even though f(char) exists

void bar() {
    using A::f;
    f('a');              // calls f(char)
}
```

5 The names thus defined are aliases for their original declarations so that the *using-declaration* does not affect the type, linkage or other attributes of the members refered to.

6 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
}
```

Box 43

This reflects paper 93-0105 but does not reflect the original namespace paper. According to the original paper, the previous example should read:

```
void g()
{
    int i;
    using B::i;     // error: i declared twice
    void f(char);
    using B::f;     // error: f declared twice
}
```

7 During overload resolution, a locally declared function is prefered over an injected one when both have the same signature. If the signature with the best match refers to more than one function, an ambiguity exists and the program is ill-formed.

Box 44

This treatment is a mistake, but it was voted in San Jose.

Editorial proposal: if a local declaration conflicts with one introduced by a *using-declaration*, the program is ill-formed. Thus, in the example below, the declaration of `f(int)` in function `h` should render the example 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;
    f(1);       // ambiguity B::f(int) or C::f(int)
    void f(int);
    f(1);       // calls local f(int)
    f('h');     // calls C::f(char)
    f(2.0);     // ambiguity B::f(double) or C::f(double);
}
```

8 Even in the presence of `using` declarations, member function declarations hide or override members with the same signature in a base class. For example:

```
struct B {
    virtual void f(int);
    void g(int);
};

struct D: B {
    using B::f;
    using B::g;
    void f(int);       // overrides B::f
    void g(int);       // hides B::g
};

void h(D* p) {
    p->f(1);           // OK
    p->g(1);           // OK
}
```

9 Omitting the name before `::` implies a reference to the global namespace:

```
void f();
namespace X {
        using ::f;   // global f
};

main()
{
        X::f();      // calls ::f
}
```

10 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 non-static member of a base class, the member name must be accessible, and if the name is that of a non-static member function, then all functions named must be accessible.

11 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
};
```

12 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 may 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;
}
... N::i ... // 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.

5 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++;       // ambiguous ::d1 or D::d1
        ::d1++;     // OK
        D::d1++;    // OK
        d2++;       // OK: D::d2
        e++;        // OK: E::e
        f(1);       // ambiguous D::f(int) or E::f(int)
        f('a');     // OK D::f(char)
}
```

7.3.5 Explict qualification [namespace.qual]

Box 45

The infomation in this section is very similar to the information provided in section 3.3.8. The information should probably be consolidated in one place.

1 A name in a class or namespace may 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 46

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 are ignored, as is any enclosing declarative region.

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 47
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 *file* 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` within the `{}` of a linkage specification. The linkage directive is ignored for such a function or object. Otherwise, a function declared in a linkage specification behaves as if it was explicitly declared `extern`. 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.[30)]

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* `)`

[30)] 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 `::` (12.1, 12.4). The *cv-qualifier* `const` shall not appear more than once in a *cv-qualifier-seq*; similarly for `volatile`.

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`." 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. 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. If the *constant-expression* (5.19) is present, it must be of enumeration or integral type and have a value 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. 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[31] (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 4.6. 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 appearance, subscripting is a commutative operation.

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

6 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.

7 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.

8 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*[32].

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 17.1.2.

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

Box 48

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 list*parameter*type

Box 49

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`" "reference to array of unknown bound of `T`," the program is ill-formed.[33] 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

[33] 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. Paramaters of a function declared before a default argument expression are in scope and may hide global 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* must contain a declarator with the form

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

as described in 8.3.5.

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.[34] 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 at file 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 "pointer to *cv1* `T`" can initialize a pointer of type "pointer to *cv2* `T`" if the set of cv-qualifiers *cv1* is a subset of *cv2*. An expression of type "*cv1* `T`" can initialize an object of type "*cv2* `T`" independently of the cv-qualifiers *cv1* and *cv2*. For example,

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

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

const int* p0 = &a;
const int* p1 = &b;
int* p2 = &b;          // error: makes a pointer to
                       // nonconst point to a const

int *const p3 = p2;
int *const p4 = p1;    // error: makes a pointer to
                       // nonconst point to a const
const int* p5 = p1;
```

The declarations of `p2` and `p4` are ill-formed for the same reason: had those initializations been allowed, they would have allowed the value of something declared `const` to be changed through an unqualified pointer.

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 storage class static (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 When an initializer applies to a pointer or an object of enumeration or arithmetic type, it consists of a single expression, perhaps in braces. The initial value of the object is taken from the expression; the same conversions as for assignment are performed.

8 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`.

9 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.

10 The semantics of initializers are as follows. In this discussion, the *target type* is the type of the object or reference being initialized and a *class reference type* is any type of the form "reference to *cv* class" type.

— If the target type is neither a class nor a class reference type, and the initializer type is not a class type[35], the behavior of the initialization is determined by the preceding rules of this subclause and Clause 4; no user-defined conversions are considered.

[35] Note that expressions of type "reference to T" are adjusted to be lvalues of type "T", so there are no special rules for initializer expressions of ref-to-class type.

— If the initializer is of type "reference to *cv1* `T1`", and the target type is either "*cv2* `T1`" or "*cv2* `class B`" (where B is an accessible unambiguous base class of the class type denoted by `T1`), then the initialization is accomplished by causing the identifier to denote a reference bound to the object or function denoted by the initializer expression; the restrictions and semantics of reference conversion are applied.

— If the initializer is of class type, a set of candidate functions is created (13.2.1), each of which is a constructor or conversion function that is a valid step in a conversion sequence leading from the initializer type to the target type.[36] From this set of candidate functions a function is chosen as described in 13.2.[37]

Box 50
This must be reconciled with the rules for template argument/parameter matching.

— Otherwise, the initializer type is not a class type. If the target type is a class or class reference type with a constructor, the candidate functions are all the constructors accepting the initializer type as an argument without user-defined conversions. From this set of candidate functions a function is chosen by the process described in 13.2.[38]

Box 51
Revise 12.6.1 (12.6.1) to reference this.

— If the target type is a class (not a class reference) type without a constructor, the behavior is determined by 8.5.1 below.

11 Any other cases are 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.

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 may also be initialized with an object of its class or of a class publicly derived from it (12.8).

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

[36] If the target type is a class or ref-to-nonconst-class type, these candidate functions include constructors of the target type. If the initializer type can be converted to the argument type of a target-type constructor (without using user-defined conversions) then that constructor is a candidate.

[37] Note that as described in 13.2, an extra "tie-breaker" is used in the overload resolution of initialization contexts: a conversion sequence containing a standard conversion after a user-defined conversion is worse than a conversion sequence in which the user-defined conversion is not followed by a standard conversion.

[38] The same "standard conversion tie-breaker" applies here.

left brace, then only enough elements from the list are taken to account for the members of the aggregate; any remaining members 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,

```
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*[39] 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.

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

8.5.2 Character arrays [dcl.init.string]

1 A `char` array (whether signed or unsigned) may 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,

```
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'
}
```

2 A reference cannot be changed to refer to another object after initialization. Note that initialization of a reference is treated very differently from assignment to it. Argument passing (5.2.2) and function value return (6.6.3) are initializations.

3 The initializer 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 If the initializer for a reference to type `T` is an lvalue of type `T` or of a type derived (10) from `T` for which `T` is an unambiguous accessible base (4.6), the reference will refer to the (`T` part of the) initializer; otherwise, if and only if the reference is to a `const` and an object of type `T` can be created from the initializer, such an object will be created. The reference then becomes a name for that object. For example,

```
double d = 2.0;

double& rd = d;          // rd refers to `d'
const double& rcd = d;   // rcd refers to `d'

double& rd2 = 2.0;       // error: not an lvalue
int  i = 2;
double& rd3 = i;         // error: type mismatch
const double& rcd2 = 2;  // rcd2 refers to temporary
                         // with value `2'
```

5 A reference to a `const` object is required to be `const`. Similarly a reference to a `volatile` or `const volatile` object is required to be `volatile` or `const volatile` (respectively). However, a `const`, `volatile`, or `const volatile` reference can refer to a plain object. For example,

```
const double d = 2.0;
double& rd = d;                    // error: non-const reference to const
const volatile double& rcvd = d;   // okay: rcvd refers to 'd'
const double& rcd = rcvd;          // error: non-volatile reference to volatile
```

6 The lifetime of a temporary object created in this way is the scope in which it is created (3.6).

9 Classes [class]

1 A class is a type. Its name becomes a *class-name* (9.1), that is, a reserved word 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.

> *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 *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.

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 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 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'
}
```

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 `friends` 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*; 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 type. In that case, the member can appear in 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 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 52
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 53
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 54
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 other than through an lvalue of 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.[40]

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

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

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;
};
```

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 55
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*

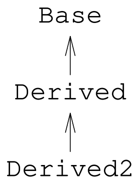
access-specifier:
  `private`
  `protected`
  `public`

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 (4.6). A reference to a derived class may be implicitly converted to a reference to an accessible unambiguous base class (4.7).

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,



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 { /* ... */ };
```

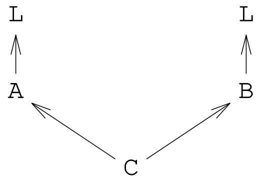
The use of more than one direct base class is often called multiple inheritance.

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
```

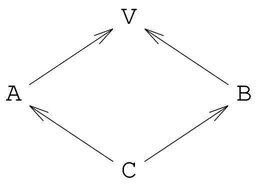
Here, an object of class `C` will have two sub-objects of class `L` as shown below.



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 { /* ... */ };
```

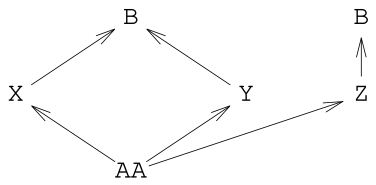
Here class `C` has only one sub-object of class `V`, as shown below.



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.



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 one`f`in `B` *hides* a member name `a`f`in `A`if `a`A`is `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,

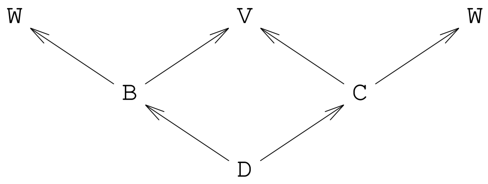
```
class A {
public:
    int f();
};

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(); };
```



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*[41]

[41] 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 non-static 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). 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[42].

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 (4.6), 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
}
```

[42] 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 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*[43] (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
};
```

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

3 If a class or function mentioned as a friend has not been declared, see 7.3.1.

4 A function first declared in a friend declaration is equivalent to an `extern` declaration (3.4, 7.1.1).

5 A global (but not a member) `friend` function may be defined in a class definition other than a local class definition (9.9). The function is then `inline` and the rewriting rule specified for member functions (9.4.2) is applied. 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.[44)] 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[45)]. The definition for an implicitly-declared copy constructor is generated only if that copy constructor is called.

Box 56
Do we need a definition for trivial implicitly-declared copy constructors?

---

44) Volatile semantics might or might not be used.

45) 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 (unique) expression that (lexically) contains the point where they were created and is not a subexpression of another expression. This is true even if that evaluation ends in throwing an exception. Temporaries created while 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.

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.[46] 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 57
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`.

[46] 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.[47)]

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

[47)] 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

Box 58

This intro and section 13.1 need to be rewritten. I would introduce the notion of a *call profile*, which is related to a full parameter type profile, but is defined such that two functions with the same call profile cannot be overloaded.

When several different function declarations are specified for a single name in the same scope, that name is said to be *overloaded*. When that name is used, the correct function is selected by comparing the types of the arguments with the types of the parameters. For example,

```
double abs(double);
int abs(int);

abs(1);        // call abs(int);
abs(1.0);      // call abs(double);
```

Since for any type `T`, a `T` and a `T&` accept the same set of initializer values, functions with parameter types differing only in this respect may not have the same name. For example,

```
int f(int i)
{
    // ...
}

int f(int& r)  // error: function types
               // not sufficiently different
{
    // ...
}
```

It is, however, possible to distinguish between `const T&`, `volatile T&`, and plain `T&` so functions that differ only in this respect may be defined. Similarly, it is possible to distinguish between `const T*`, `volatile T*`, and plain `T*` so functions that differ only in this respect may be defined.

2 Functions that differ only in the return type may not have the same name.

3 Member functions that differ only in that one is a `static` member and the other isn't may not have the same name (9.5).

4 A `typedef` is not a separate type, but only a synonym for another type (7.1.3). Therefore, functions that differ by typedef "types" only may not have the same name. For example,

```
typedef int Int;

void f(int i) { /* ... */ }
void f(Int i) { /* ... */ }      // error: redefinition of f
```

Enumerations, on the other hand, are distinct types and can be used to distinguish overloaded functions. For example,

```
enum E { a };

void f(int i) { /* ... */ }
void f(E i)   { /* ... */ }
```

5 Parameter types that differ only in a pointer `*` versus an array `[]` are identical, 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]);
```

6 Parameter types that differ only in the presence or absence of `const` and/or `volatile` are identical. 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. In particular, for any type `T`, `T*`, `const T*`, and `volatile T*` are considered distinct parameter types, as are `T&`, `const T&`, and `volatile T&`.

13.1 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 identical parameter types (13). 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 Recall from 5.2.2, that a function call is a *postfix-expression* followed by an optional *expression-list* enclosed in parentheses. Of interest in this section are only those function calls in which the *postfix-expression* has the following forms:

postfix-expression:
  primary-expression
  postfix-expression `.` *id-expression*
  postfix-expression `->` *id-expression*

In these cases, the *postfix-expression* ultimately contains a name that must be resolved against visible declarations to identify which function is being called.

2 Since, through overloading declarations, a name may refer to more than one function, the function referenced by a function call is determined not only by the name, but also by the kind of function call, the number of arguments present, and their types. The name and the kind of function call determine a set of *candidate functions* that could be referenced by the name. From this set of candidate functions a function is chosen whose parameters best match the arguments in the call in number and type.

13.2.1 Candidate functions [over.match.funcs]

1 There are two kinds of function calls: member function calls and ordinary (or non-member) function calls.

2 In member 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 chapter assumes, without loss of generality, that all member function calls have been *normalized* to the form that uses an object

pointer and the `->` operator. Furthermore, the left operand of the `->` operator has type `T*`, where `T` denotes some class `X` optionally qualified by `const` and/or `volatile`.[48] Thus, in a member function call, the *id-expression* in the call is looked up as a member function of `X` following the rules for looking up names in classes (10). If a member function is found, that function and its overloaded declarations (in the same scope) constitute the set of candidate functions submitted to argument matching (13.2.2).

3 In non-member 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 function declaration, that function and its overloaded declarations (in the same scope) constitute the set of candidate functions submitted to argument matching (13.2.2).

4 If the name in the ordinary function call resolves to a member function and the keyword `this` is in scope and refers to the class of that member function, then the ordinary-looking function call is actually a member function call using an implicit `this` pointer. In this case, the function call is put into normalized member call form using an explicit `this` pointer.

5 In either kind of function call, the name may resolve to something other than a function name. This section, 13.2, will not consider this case further since such a name cannot be overloaded.

6 Section 13.4.8 describes the set of candidate functions constructed for the resolution of an overloaded operator in an expression.

13.2.2 Argument matching [over.match.args]

1 From the set of candidate functions constructed for a function call (13.2.1) or an operator in an expression (13.4.8), a function is chosen whose parameters best match the arguments in the call according to the rules described in this section.

2 To be considered at all, a candidate function must have enough parameters to satisfy the arguments in the call. If there are m arguments in the call, all candidate functions having exactly m parameters remain candidates unconditionally. A candidate function having fewer than m parameters remains a candidate only if it has an ellipsis in its parameter list (8.3.5). For the purposes of argument matching, 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 remains a candidate only if the $m+1st$ parameter has a default initializer (8.3.6). For the purposes of argument matching, the parameter list is truncated on the right, so that there are exactly m parameters.

3 From the subset of candidate functions with the correct number of parameters a function is chosen that best matches the arguments in the call. The choice is made in two steps. First, for each individual argument in the call, the subset of the candidate functions that best match that argument is determined according to the rules for *best-match* described below. Then, the function that best matches the call is obtained by forming the intersection of the subsets obtained for each argument. Unless this intersection has exactly one function, the call is ill-formed.

4 The function thus selected must be a better match to the call than any other candidate function. Otherwise, the call is ill-formed. One function is a better match to the call than another if for each argument in the call, the first function is at least as good a match as the second function, and for some argument the first function is a better match.

5 For purposes of argument matching, a non-static member function is considered to have an extra parameter, which must match the pointer specified in the normalized member function call (13.2.1) as if the pointer were also an argument in the call. No temporaries will be introduced for this extra parameter and no user-defined conversions will be applied to achieve a type match. The type of this extra parameter is the type of the keyword `this` (9.4.1) within the member function. For example, for a `const` member function of class `X`, the extra parameter is assumed to have type `const X*`.

[48] Note that *cv-qualifiers* on the type of objects are significant in overload resolution for both lvalue and rvalue objects.

6 How well a function *matches* an argument is based on the sequence of implicit conversions that can be applied to the argument to yield a value of the type required by the corresponding parameter of the function. For the purposes of argument matching, no sequence of conversions is considered that

(a) does not lead to the type required by the parameter, or

(b) contains more than one user-defined conversion, or

(c) can be shortened into another considered sequence by deleting one or more conversions. (For example, `int`→`float`→`double` is a sequence of conversions from `int` to `double`, but it is not considered because it contains the shorter sequence `int`→`double`.)

7 Some sequences of conversions are better than others according to rules that are given below. If, according to these rules, there is a single sequence of conversions that is uniquely better than all the rest, it is called the function's *best-matching* sequence for the argument. One function matches an argument better than another if it has a best-matching sequence for that argument and its best-matching sequence is better than the best-matching sequence of the other function. A function is a best match for an argument if it has a best-matching sequence for that argument and no other function is a better match for the argument.

Box 59

I feel I've gone out on a limb with the preceding paragraph. I don't honestly believe that earlier drafts actually explained how a best-matching function is derived from best-matching sequences. Nor did it explain what happens if there is more than one best-matching sequence.

8 An ellipsis in a parameter list (8.3.5) is a match for an argument of any type.

9 Except as mentioned below, the following *trivial conversions* involving a type `T` do not affect which of two conversion sequences is better: the conversion of an argument of type "pointer to *cv1* `T`" to the type "pointer to *cv2* `T`" if the set of cv-qualifiers *cv1* is a subset of *cv2* (7.1.5 see also 8.5). Where necessary, `const` and `volatile` are used as tie-breakers as described in rule [1] below.

Box 60

The table was removed. "T"->"T&", "T&"->"T", "T&"->"const T&", "T&"->"volatile T&", "T&"->"const volatile T&" were removed because a reference initialization is considered a binding and not a conversion. As well, expressions of reference type are transformed into lvalue expressions very early during expression processing, before argument matching takes place. "T[]"->"T*", "T(args)"->"(*T)(args)" were removed because expressions of type "array of" and of type "function of" are transformed into expressions of type "pointer to" and "pointer to function of" very early during expression processing, before argument matching takes place. "T"->"const T", "T"->"volatile T", "T"->"const volatile T" were removed because the cv-qualifiers of pass-by-value parameters do not participate in the function type.

10 If a parameter is of type `const T&`, the effect of binding the reference to a temporary (8.5.3) does not affect argument matching. Any function that would require initializing a non-const reference with a temporary (8.3.2) is excluded as a match during overload resolution.

11 Sequences of conversions are considered according to these rules:

12

[1] Exact match: Sequences of zero or more trivial conversions are better than all other sequences.

[2] Match with promotions: Of sequences not mentioned in [1], those that contain only integral promotions (4.1), conversions from `float` to `double`, and trivial conversions are better than all others.

[3] Match with standard conversions: Of sequences not mentioned in [2], those with only standard (4) and trivial conversions are better than all others. Of these, if `B` is derived directly or

indirectly from A, converting a B* to A* is better than converting to void* or const void*. Further, if C is publicly derived directly or indirectly from B, converting a C* to B* is better than converting to A* and converting a C to B& is better than converting to A&. Similarly, converting an A::* to B::* is better than converting an A::* to C::*.

[4] Match with user-defined conversions: Of sequences not mentioned in [3], those that involve only user-defined conversions (12.3), standard (4) and trivial conversions are better than all other sequences.

[5] Match with ellipsis: Sequences that involve matches with the ellipsis are worse than all others.

13 User-defined conversions are selected based on the type of variable being initialized or assigned to.

14

Box 61

Where did this come from? It relates to conversion sequences and ambiguities therein, but it is not in the context of overload resolution. Are there other places that these conversion sequences are used in the language?

15

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

void f(Y y)
{
    int i = y;     // call Y::operator int()
    double d;
    d = y;         // call Y::operator double()
    float f = y;   // error: ambiguous
}
```

16 Standard conversions (4) may be applied to the argument of a user-defined conversion, and to the result of a user-defined conversion.

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

void f(long), f(char*);
void g(S), g(char*);
void h(const S&), h(char*);

void k(S& a)
{
    f(a);       // f(long(a.operator int()))
    g(1);       // g(S(long(1)))
    h(1);       // h(S(long(1)))
}
```

Except when one conversion sequence is a subsequence of another, if two conversion sequences each contain a user-defined conversion, any standard conversions also used in the sequences do not affect which sequence is better. For example,

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

class Y {
public:
    Y(long);
};
class Z {
public:
    operator int();
};

void f(X);
void f(Y);
void g(int);
void g(double);

void g()
{
    f(1);       // ambiguous
    Z   z;
    g(z);       // okay -- g(int(z))
}
```

The call `f(1)` is ambiguous despite `f(y(long(1)))` needing one more standard conversion than `f(x(1))`, and the call `g(z)` is unambiguous even though `g(double(int(z))` has only one user-defined conversion. The difference is that the two conversion sequences found for `f()` contain two *different* user-defined conversions and neither sequence is a subsequence of the other, while the two conversion sequences found for `g()` contain the same user-defined conversion and one is a subsequence of the other.

17 No preference is given to conversion by constructor (12.1) over conversion by conversion function (12.3.2) or vice versa.

```
struct X {
    operator int();
};

struct Y {
    Y(X);
};

Y operator+(Y,Y);

void f(X a, X b)
{
    a+b;  // error, ambiguous:
          //     operator+(Y(a), Y(b)) or
          //     a.operator int() + b.operator int()
}
```

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 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)`.

3 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.

4 Note also that there are no standard conversions (4) of one pointer to function type into another (4.6). 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
```

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 13.4.8 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 13.4.8 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`.

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`.

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`. 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.

13.4.8 Overloaded operators in expressions [over.oper.funcs]

1 To determine which operator function is to be invoked to implement an expression involving an operator, the operator notation is first transformed to the equivalent function-call notation as summarized in the Table 12 (where @ denotes one of the operators covered in the specified section).

Table 12—relationship between operator and function call notation

Section	Expression	Member function	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)`	`---`
`13.4.4`	`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@ (1)`	`operator@ (a, 1)`

2 If no operand of the operator has a type that is a class, a reference to a class, an enumeration, or a reference to an enumeration, the operator is assumed to be a built-in operator and interpreted accordingly. For example:

```
struct Thing {
    Thing(char*);
    Thing operator+(char*);
    operator char*();
};

void f()
{
    char* p = "one" + "two";        // ill-formed
    int i = 1 + 1;                  // i = 2
}
```

The declaration of `p` is ill-formed because neither operand of `+` is a (pointer or reference to a) class or enum. The operands are not implicitly converted to `Thing` and the `+` does not refer to `Thing::operator+(char*)`. Similarly, `1+1` is always `2` regardless of any other definitions of `operator+`.

3 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`. A set of candidate member functions is constructed for the *operator-function-id* as if it were named in a member call as a member of the first operand according to the rules in 13.2.1.

4 If the operator is either a unary or binary operator (sections 13.4.1, 13.4.2, or 13.4.7) and either operand has a type that is a class, reference to a class, an enumeration, or a reference to an enumeration, the operator could be implemented by a non-member operator function. A set of candidate functions is constructed for the *operator-function-id* as if it were named in an ordinary call according to the rules in 13.2.1.

5 If both sets of candidate functions described above are empty, the operator is assumed to be a built-in operator and interpreted accordingly. Otherwise, the two sets are combined into one set of candidate functions from which an appropriate function is selected according to the argument matching rules defined in 13.2.2.

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.

3 The names of a template obeys the usual scope and access control rules. A *template-declaration* may 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 may not be `virtual`. A destructor may not be a template. A local class may 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-id* `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 may 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 undefined 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 { /* ... */ }
X< Y<1> > x3; // ok
```

Box 62
Should we bless a hack allowing `X<Y<1>>`? (yes)

6 The name of a class template may not be declared to refer to any other template, class, function, object, namespace, value, or type in the same scope. 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 leftmost identifier of the *qualified-name* must be a *template-argument* name.

Box 63
I have chosen the most restrictive variant of this idea. We ought to consider if the construct should be allowed for a nonqualified name, and if the construct would be useful outside templates.

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, cannot be diagnosed until later; such errors may 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, may be diagnosed there or later together with the type errors.

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, and names declared within the template itself.

— Names from the scope of the template definition.

— Names dependent on a *template-argument* 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.

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)
```

Note that if an implementation somehow replicates class or template definitions so that names used in the class or template bind to different names in different compilations, the one-definition rule has been violated and any use of the class or template is ill-formed. Violation of the one-definition rule by template instantiation is a non-required diagnostic.

Box 64
Are violations of the one-definition rule required if violation is in a single file? (no)

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 65**
>
> 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 eventhough `g(int)` wasn't in scope at the point of the template definition.

5 A name from a base class may hide the name of a *template-parameter*. For example:

```
struct A {
        struct B { /* ... */ };
};

template<class B> struct X : A {
        B b;  // A's B
};
```

A name of a member may hide the name of a *template-parameter* in a member function definition. 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 the 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 X operator+(const X&, const X&); // 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)
}
```

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; see _temp.dcls_.

2 The act of generating a class, function, or static data member from a template is commonly referred to as template instantiation.

3 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 global declaration 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. For example:

```
// void g(int); not declared here

template<class T> class Y {
public:
        void f() { g(1); }
};

void k(const Z<int>& h)
{
        void g(int);
        h.f(); // error: g(int) called by g(1) not found
}
```

Each compilation unit in which the definition of a template is used has a point of instantiation for the class. 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.

4 A template can be either explicitly instantiated for a given argument list or be implicitly instantiated. A template that has been used in a way that require a specialization of its definition will have the specialization implicitly generated unless it has either been explicitly instantiated (14.4) or explicitly specialized (14.5). A specialization will not be implicitly generated unless the definition of a template specialization is required. 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 may be instantiated for implementation purposes.

5 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>*
}
```

6 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
```

7 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 ...
};
```

8 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-arguments*. An implementation is not required to diagnose a violation of this rule.

9 An explicit specialization or explicit instantiation of a template must be in the namespace in which the template was defined. Implicitly generated template classes, functions, and static data members are placed in the namespace where 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 namspace
}

template class Y<int> { // error: explicit instantiation
                        //        in different namespace
        // ...
};

void g(int);
namespace M {
        void g(int);
        N::Z<int> nz; // which g() does N::Z<int>::f() call?
                      // ::g()
        nx.f(2);
}
```

The reason `::g()` is called is that the point of instantiation is before M.

> **Box 66**
> This resolution hasn't specifically been voted on. The behavior is chosen to match template uses in classes and functions.

10 Recursive instantiation is possible. For example:

```
template<int i> int fac() { return i>1 ? i*fac<i-1>() : 1; }

int f()
{
        return fac<17>();
}
```

There shall be an implementation quantity that specifies the limit on the depth of recursive instantiations.

14.4 Explicit instantiation [temp.explicit]

1 The syntax for explicit instantiation is:

instantiation:
    `template` *specialization*

For example:

```
template class vector<char> { /* ... */ };

        // instantiate sort(vector<char>&):
template void sort<char>(vector<char>&);
```

2 A trailing *template-argument* may 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:

```
        // deduce template-argument:
template void sort<>(vector<int>&);
```

Box 67

Can we instantiate if there is no definition in scope? Yes, but answering this question requires a model for compilation of templates. See §4 of N0413/94−0026.

3 The explicit instantiation of a class implies the instantiation of all of its members. Thus, it is not possible to both explicitly instantiate a class and to specialize some of its members for a given *template-argument-list*.

Box 68

Can we instantiate a class if the definition of some of its member functions are not in scope? Yes, but answering this question requires a model for compilation of templates. See §4 of ANSI X3J16/94-0026, ISO WG21/N0413.

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:
  template-name < *template-argument-list* > *declaration*

For example:

```
template<class T> class stream;

class stream<char> { /* ... */ };

template<class T> void sort(vector<T>& v) { /* ... */ }

void sort<char*>(vector<char*>& v)  { /* ... */ }
```

Given these declarations, `stream<char>` will be used as the definition of streams of `chars`; 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 An explicitly specialized class or an explicitly specialized function must be declared before it can be used. Specializing a class or a function after it has been used in a translation unit or in another translation unit is ill-formed. 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-name*
  `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 69
This grammar leaves out namespace *template-parameter*s. See §2 of ANSI X3J16/94-0026, ISO WG21/N0413.

Box 70
This grammar should be modified to accept `struct` as well as `class` for template *template-parameter*s.

2 Default arguments may not be specified in a declaration or a definition of a specialization.

3 A *type-parameter* defines its *identifier* to be a *type-id* in the scope of the template declaration. A *type-parameter* may not be redeclared within its scope (including nested scopes). A non-type *type-parameter* may 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 Note that because there are no constant expression of floating type and standard conversions are not applied to *template-argument*s a *template-parameter* cannot be of floating type. 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* may be specified in a template declaration or a template definition. A function template may not have default *template-argument*s. The set of default *template-argument*s 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* may 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 Similarly, a *template-argument* may 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-arguments* specified in a *template-id* must match the types specified for the template in its *template-parameter-list*. For example, `vectors` 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 Non-type non-reference *template-arguments* must be *constant-expressions* or addresses of objects or functions with external linkage. 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
```

Nor is a local type or an unnamed type an acceptable *template-argument*. For example:

```
void f()
{
        struct S { /* ... */ };

        X<S,p> x3; // error: local type used as template-argument
}
```

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
```

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 A template *type-parameter* can be used in an *elaborated-type-specifier*. 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:

```
template<class T> class X {
        class T* p;
};

struct S { /* ... */ };
union U { /* ... */ };
enum E { /* ... */ };

X<S> s;   // fine
X<int> i; // error: template-argument must be a class
X<U> u;   // error: template-argument must be a class
X<E> e;   // error: template-argument must be a class
```

3 An argument for a *template-parameter* of reference type must be a *constant-expression*, 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.

4 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; see _temp.dcls_. 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 need not be explicitly specified. For example,

```
void f(vector<complex>& cv, vector<int>& ci)
{
    sort(cv);   // sort(vector<complex>)
    sort(ci);   // sort(vector<int>)
}
```

and

```
void g(double d)
{
        int i = convert<int>(d);  // int convert(double)
        int c = convert<char>(d);  // char convert(double)
}
```

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 arguments 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].

2 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
}
```

Nontype parameters may 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
```

3 Every *template-parameter* specified in the *template-parameter-list* must be either explicitly specified or deduced from a function argument. If function *template-arguments* are specified in a call they are specified in declaration order. Trailing arguments can be left out of a list of explicit *template-arguments*. 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

}
```

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)
}
```

14.9.3 Overload resolution [temp.over]

1 A template function may 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 (13.2.2) is used to look for a best match; if found, call it.

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, and argument of type ‘‘class derived from B<params>’’ may 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>’’ may be converted to B<params>*. Similarly for references.

— A pointer (reference) may be converted to a more qualified pointer (reference) type, according to the rules in 4.6 (4.7).

— ‘‘array of T’’ to ‘‘pointer to T.’’

— ‘‘function ...’’ to ‘‘pointer to function to ... .’’

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 (_temp.dcls_). Not even trivial conversions (13.2) will be applied in this case.

Box 71

This is too strict. To match existing usage, a proposal for allowing at least some trivial conversions will undoubtedly be accepted. 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).

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<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);
template<class T> void f(T,char);

void h(int* pi, int i, char c)
{
        f(pi,i);  // f<int>(pi,i)
        f(pi,c);  // f<char*>(pi,c)
        f(i,c);   // f<int>(i,c);
        f(i,i);   // f<int>(i,char(i))
}
```

10 A function template definition is needed to generate specific versions of the template; only a function template declaration is needed to generate calls to specific versions.

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, class Z> void f(X,Y*,Z);
template<class X, class Y, class Z> void f(X*,Y,Z);

void g(char* pc, int* pi)
{
        f(0,0,0); // error: ambiguous: f<int,int,int>(int,int*,int)
                  //                or f<int,int,int>(int*,int,int) ?
        f<char>(pc,pi,0);  // f<char,int*,int>(char*,int*,int)
        f<char*>(pc,pi,0); // f<char*,int,int>(char*,int*,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 not a 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 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);
```

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 is 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, char y)
{
        max(x,y); // max<char>(a,b)
}

int max(int,int);

void f(char x, char y)
{
        max(x,y); // max<int>(iny(x),int(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 may or may not be a 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 may 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*$_{opt}$

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.[49)]

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:

49) 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 sub-objects. 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 (4.6) 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 (4.6) 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 72**
> 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.[50)]

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 73
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 Z: public X { };
class W { };

void f() throw (X,Y)
{
    int n = 0;
    if (n) throw X();        // OK
    if (n) throw Y();        // 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.

50) 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>` (17.3.2).

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,[51)]
- 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.

[51)] 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.[52)]

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*

52) Thus, preprocessing directives are commonly called “lines.” These “lines” have no other syntactic significance, as all white space is equivalent except in certain situations during preprocessing (see the `#` character string literal creation operator in 16.3.2, for example).

control-line:
`# include` *pp-tokens new-line*
`# define` *identifier replacement-list new-line*
`# define` *identifier lparen identifier-list*$_{opt}$ `)` *replacement-list new-line*
`# undef` *identifier new-line*
`# line` *pp-tokens new-line*
`# error` *pp-tokens*$_{opt}$ *new-line*
`# pragma` *pp-tokens*$_{opt}$ *new-line*
`#` *new-line*

lparen:
the left-parenthesis character without preceding white-space

replacement-list:
pp-tokens$_{opt}$

pp-tokens:
preprocessing-token
pp-tokens preprocessing-token

new-line:
the new-line character

2 The only white-space characters that shall appear between preprocessing tokens within a preprocessing directive (from just after the introducing `#` preprocessing token through just before the terminating new-line character) are space and horizontal-tab (including spaces that have replaced comments or possibly other white-space characters in translation phase 3).

3 The implementation can process and skip sections of source files conditionally, include other source files, and replace macros. These capabilities are called *preprocessing*, because conceptually they occur before translation of the resulting translation unit.

4 The preprocessing tokens within a preprocessing directive are not subject to macro expansion unless otherwise stated.

16.1 Conditional inclusion [cpp.cond]

1 The expression that controls conditional inclusion shall be an integral constant expression except that: it shall not contain a cast; identifiers (including those lexically identical to keywords) are interpreted as described below;[53)] 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.

[53)] 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.[54] 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.[55]

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.

---

54) 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)
```

55) 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.[56] 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 74
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.

[56] 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[57)] 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

[57)] 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 [lib.library]

Box 75

Library WG issue: Michael Vilot, January 14, 1994

This section ordering has not been discussed by the Library Working Group. Once they do have a chance to discuss it, the section order, numbering, and names are likely to be changed.

17.1 Introduction [lib.introduction]

Box 76

Library WG issue: Michael Vilot, November 22, 1993 *

How much of ''Introduction'' section has to be made global to the entire clasue? The various C rules about reserved identifiers could be made irrelevant if C in C++ programs were prohibited from defining macros (except, presumably, for a few things like `assert`) If we don't define the standard namespace in a way that obviates the need for so many rules, then we haven't used the language feature effectively.

1 The *Standard C++ library* contains components for: language support, predefined exceptions, iostreams, strings, bit sets, bit strings, dynamic arrays, complex numbers, and locale objects. 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_). The predefined exceptions provide support for uniform error reporting from the Standard C++ library. The iostreams components are the primary mechanism for C++ program input/output. The strings and other containers provide some of the most commonly used data types not directly defined in the C++ language. And the complex components provide support for numeric processing. This library also makes available the facilities of the Standard C library, suitably adjusted to ensure static type safety.

17.1.1 Standard C library [lib.intro.standard.c]

1 This International Standard includes by reference clause 7 of the C Standard and clause 4 of Amendment 1 to the C Standard (1.2). The combined library described in those clauses is hereinafter called the *Standard C library.* With the qualifications noted in this subclause 17.1 and in 17.2, the Standard C library is a subset of the Standard C++ library.

17.1.2 Headers [lib.headers]

Box 77

Library WG issue: Michael Vilot, November 22, 1993

The rule that ‘‘any of the C++ headers can include any of the other C++ headers’’ imposes a restriction on C++ programmers beyond any that C programmers must endure. Since we are changing the names of the headers from current usage anyway (by dropping the .h), we can be unambiguous about the declarations used across components in the standard library. Implementations that support precompiled headers will do just fine with a more precise specification.

1 The elements of the Standard C++ library are declared or defined (as appropriate) in a *header,* whose contents are made available to a translation unit when it contains the appropriate #include preprocessing directive.[58] Objects and functions defined in the library and required by a C++ program are included in the program prior to program startup.

2 The Standard C++ library provides 41 *C++ headers,* as shown in Table 13:

Table 13—C++ Headers

HEADER	HEADER	HEADER	HEADER
<all>	<complex>17.5.7	<cwctype>	<new>17.3.3
<bits>17.5.3	<csetjmp>	<defines>17.3.1	<objcpy>17.5.8
<bitstring>17.5.4	<csignal>	<dynarray>17.5.5	<ostream>17.4.4
<cassert>	<cstdarg>	<exception>17.3.2	<ptrdynarray>17.5.6
<cctype>	<cstddef>	<fstream>17.4.8	<sstream>17.4.7
<cerrno>	<cstdio>	<iomanip>17.4.5	<streambuf>17.4.2
<cfloat>	<cstdlib>	<ios>17.4.1	<string>17.5.1
<ciso646>	<cstring>	<iostream>17.4.9	<strstream>17.4.6
<climits>	<ctime>	<istream>17.4.3	<typeinfo>17.3.4
<clocale>	<cwchar>	<locale>17.5.9	<wstring>17.5.2
<cmath>			

3 For compatibility with the Standard C library, the Standard C++ library provides the 18 *C headers,* as shown in Table 14:

Table 14—C Headers

HEADER	HEADER	HEADER	HEADER
<assert.h>	<limits.h>	<stdarg.h>	<string.h>
<ctype.h>	<locale.h>	<stddef.h>	<time.h>
<errno.h>	<math.h>	<stdio.h>	<wchar.h>
<float.h>	<setjmp.h>	<stdlib.h>	<wctype.h>
<iso646.h>	<signal.h>		

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

4 The header <all> includes all the other C++ headers.

5 If a header is implemented as a source file, the derivation of the file name from the header name is implementation-defined. If a file has a name equivalent to the derived file name for one of the above 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, the behavior is undefined.

6 A translation unit may include these headers in any order. 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. 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.

7 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.

8 None of the C headers includes any of the other headers, except that each C header includes its corresponding C++ header, as described above, followed by an explicit using-directive (7.3.3) for each name that the C++ header declares or defines in the namespace std (17.1.4). Except for the header <all>, 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. [59)]

17.1.3 Processor Compliance [lib.compliance]

1 Two kinds of implementations are defined: *hosted* and *freestanding*. For a hosted implementation, this International Standard defines the set of available headers. 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 headers that provide C++ language support (as described in 17.3)

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

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

17.1.4 Reserved names [lib.reserved.names]

> **Box 78**
>
> **Library WG issue:** Michael Vilot, January 14, 1994
>
> This section has not been discussed by the Library Working Group. Once they do have a chance to discuss it, the contents are likely to be removed or changed.

1 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.

2 Each C++ header defines the namespace std. Each C++ header declares or defines all names listed in its associated subclause within that namespace.

3 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.

59) 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.

4 Each name defined as a macro in a header is reserved to the implementation for any use if the translation unit includes the header.[60)]

5 Certain sets of names and function signatures are reserved whether or not a translation unit includes a header:

— Each name that begins with an underscore and either an uppercase letter or another underscore is reserved to the implementation for any use.

— 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.

— 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.[61)]

— Each global function signature declared with external linkage in a header is reserved to the implementation to designate that function signature with external linkage. [62)]

— Each name having two consecutive underscores is reserved to the implementation for use as a name with both `extern "C"` and `extern "C++"` linkage.

— 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.[63)]

6 It is unspecified whether a name declared with external linkage in a C header has either `extern "C"` or `extern "C++"` linkage.[64)]

7 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.

8 No other names or global function signatures are reserved to the implementation. [65)]

17.1.5 Restrictions and conventions [lib.res.and.conventions]

Box 79

Library WG issue: Michael Vilot, January 14, 1994

This section has not been discussed by the Library Working Group. Once they do have a chance to discuss it, the contents of this section and its subsections are likely to be removed or changed.

---

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

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

62) 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>`.

63) The function signatures declared in `<cwchar>` and `<cwctype>` are always reserved, notwithstanding the restrictions imposed in subclause 4.5.1 of Amendment 1 to the C Standard for these headers.

64) 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.

65) 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*.

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

1 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.

17.1.5.2 Restrictions on 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.

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

Box 80

Library WG issue: Dag Brück, January 23, 1994

> Jerry Schwarz writes:
>
>> I think this should be changed to allow any function
>> to throw `xalloc`.
>
> ‘‘Any of the functions defined in the Standard C++ library
> can report a failure to allocate storage by calling ex.raise()
> for an object ex of type xalloc.

Pardon me for being picky and generally difficult, but I think Jerry’s wording is significantly superior, and I ask for a change.

I think the current wording is circuitous, and the prevailing terminology is "throw an exception" when talking about the concept, not the actual implementation.

Here’s my suggested wording:

‘‘Any of the functions defined in the Standard C++ library can report a failure to allocate storage by throwing xalloc.’’

1 Any of the functions defined in the Standard C++ library can report a failure to allocate storage by calling *`ex.raise()`* for an object *ex* of type *alloc*. 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.

2 None of the functions defined in the Standard C++ library catch any exceptions, unless explicitly stated otherwise.[66]

[66] A function can catch an exception not documented in this clause provided it rethrows the exception.

17.1.5.4 Alternate definitions for functions [lib.alternate.definitions.for.functions]

Box 81

Library WG issue: Michael Vilot, November 22, 1993

It took us 9 months or so to work out the wording in 93-0148/N0355 to describe ''installing'' handler functions in such a way as to get reasonably clear semantics without overly constraining a multithreded implementation. There is no reason to discard that work lightly, although I would like to see a more precise description of ''installing'' and ''invoking'' a handler function that doesn't involve the overspecification of requiring a global pointer.

The following changes added in 93-0108 should be removed:
''Certain handler functions are determined by the values stored in pointer objects within the Standard C++ library. Initially, these pointer objects 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.''

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 functions defined in the program:

— Four function signatures defined in the Standard C++ library may be displaced by definitions in the program. Such displacement occurs prior to program startup. [67)]

— 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.

— 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.

2 In all such cases, this clause distinguishes two behaviors for the functions in question:

— *Required behavior* describes both the behavior provided by the implementation and the behavior that shall be provided by any function definition in the program.

— *Default behavior* describes any specific behavior provided by the implementation, within the scope of the required behavior.

3 Where no distinction is explicitly made in the description, the behavior described is the required behavior.

4 If a function defined in the program fails to meet the required behavior when it executes, the behavior is undefined.

17.1.5.5 Objects within classes [lib.objects.within.classes]

67) The function signatures, all declared in `<new>`, are `operator delete(void*)`, `operator delete[](void*)`, `operator new(size_t)`, and `operator new[](size_t)`.

Box 82

Library WG issue: Tom Keffer, March 8, 1994

The San Diego rewrite dropped all uses of pre- and post-condition specifications on member functions. [See: X3J16/93-0013R1, 93-0060, and 93-0064, as voted upon and accepted.]

Comment (Library WG meeting, San Diego):

The general concern is that the text describes specifics of what happens to the ''exposition only'' member data, rather than behavior.

Example:

17.5.1.1.1 describes the action of the default constructor in terms of how the ''exposition only'' data should be initialized. It doesn't say whether the string is the null string, an unitialized string of unspecified length, or what...

Recommend:

Generic behaviour should be specified, possibly with the aid of the exposition implementation.

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

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

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

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

Box 83

Library WG issue: Tom Keffer, March 8, 1994

All classes should explicitly list the copy constructor, assignment operator, and destructor in their description.

But: 17.1.5.6 should state that an implementation can rely on the compiler to actually generate such functions.

1 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
```

2 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.

3 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;[68)]

[68)] 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.

4 A call to a member function signature described in this clause behaves the same as if the implementation declares no additional member function signatures.[69]

5 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.

17.1.5.7 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.[70]

17.1.5.8 Unreserved 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.

17.1.5.9 Implementation types [lib.implementation.types]

1 Certain types defined in this clause are based on other types, but with added constraints.

---

[69] 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.

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

17.1.5.9.1 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.

17.1.5.9.2 Bitmask types [lib.bitmask.types]

Box 84

Library WG issue: Mark Terribile, December 20, 1993

>Bitmask types
...
>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.
^^^^^^^^^^

'If the expression ... is non-zero' or 'if the expression ... is equal to Y' ? The former only works if the value Y is restricted to a single bit. I think that the I/O system requires multibit values (but I could be mistaken).

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.

17.1.5.9.3 Derived classes [lib.derived.classes]

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.[71)]

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

17.1.5.10 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.

17.1.5.11 Definitions [lib.definitions]

> **Box 85**
>
> **Library WG issue:** Michael Vilot, November 22, 1993
>
> This subclause should be merged with Section 1.3.

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.

— 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>`.

— 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.

— 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).[72) *

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

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

— A *static NTBS* is an NTBS with static storage duration.[73)]

— 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.[74)]

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

— A *wide-character sequence* is an array object *A* 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.

72) 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.
73) A string literal, such as `"abc"`, is a static NTBS.
74) 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.

— A *null-terminated wide-character string,* or *NTWCS,* is a wide-character sequence whose highest-addressed element with defined content has the value zero.[75) *

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

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

— A *static NTWCS* is an NTWCS with static storage duration.[76)

17.2 Standard C library [lib.standard.c.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 17.1 imposes some \&\f2implicit\fP\& changes in the behavior of the Standard C library.)

17.2.1 Modifications to headers [lib.mods.to.headers]

1 Each C header, whose name has the form \&\f6name\&\fP\f5.h\fP\&, includes its corresponding *C header \&\f5c\&\fP\f6name\fP\&, followed by an explicit using-declaration (7.3.3) for each name placed in the standard library namespace by the header.*f

17.2.2 Modifications to definitions [lib.mods.to.definitions]

17.2.2.1 Type \&\f7wchar_t\fP\& [lib.wchar.t]

1 \&\f5wchar_t\fP\& is a keyword in this International Standard. It does not appear as a type name defined in any of \&\f5<stddef.h>\fP\&, \&\f5<stdlib.h>\fP\&, or \&\f5<wchar.h>\fP\&.

17.2.2.2 Macro \&\f7NULL\fP\& [lib.null]

1 The macro \&\f5NULL\fP\&, defined in any of \&\f5<locale.h>\fP\&, \&\f5<stddef.h>\fP\&, \&\f5<stdio.h>\fP\&, \&\f5<stdlib.h>\fP\&, \&\f5<string.h>\fP\&, \&\f5<time.h>\fP\&, or \&\f5<wchar.h>\fP\&, is an implementation-defined *C null-pointer constant in this International Standard.*f

17.2.2.3 Header \&\f7<iso646.h>\fP\& [lib.header.iso646.h]

1 The tokens \&\f5and\fP\&, \&\f5and_eq\fP\&, \&\f5bitand\fP\&, \&\f5bitor\fP\&, \&\f5compl\fP\&, \&\f5not_eq\fP\&, \&\f5not\fP\&, \&\f5or\fP\&, \&\f5or_eq\fP\&, \&\f5xor\fP\&, and \&\f5xor_eq\fP\& are keywords in this International Standard. They do not appear as macro names defined in \&\f5<iso646.h>\fP\&.

17.2.3 Modifications to declarations [lib.mods.to.declarations]

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

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

[77) The header \&\f5<stdlib.h>\fP\&, for example, makes all declarations and definitions available in the global name space, much as in the C Standard. The header \&\f5<cstdlib>\fP\& provides the same declarations and definitions within the namespace \&\f5std\fP\&.

[78) Possible definitions include \&\f50\fP\& and \&\f50L\fP\&, but not \&\f5(void*)0\fP\&.

17.2.3.1 \&\f7memchr(const\ void*, int, size_t)\fP\& [lib.memchr]

1 The function signature \&\f5memchr(const void*, int, size_t)\fP\&, declared in \&\f5<string.h>\fP\& in the C Standard, does not have the declaration

```
void* memchr(const void* \&\fP\f6s\&\fP\f5, int \&\fP\f6c\&\fP\f5, size_t \&\fP\f
```

2 in this International Standard. Its declaration in \&\f5<string.h>\fP\& is replaced by the two declarations:

```
const void* memchr(const void* \&\fP\f6s\&\fP\f5, int \&\fP\f6c\&\fP\f5, size_t \
```

3 both of which have the same behavior as the original declaration.

17.2.3.2 \&\f7strchr(const\ char*, int)\fP\& [lib.strchr]

1 The function signature \&\f5strchr(const char*, int)\fP\&, declared in \&\f5<string.h>\fP\& in the C Standard, does not have the declaration:

```
char* strchr(const char* \&\fP\f6s\&\fP\f5, int \&\fP\f6c\&\fP\f5);.ix "[strchr]"
```

2 in this International Standard. Its declaration in \&\f5<string.h>\fP\& is replaced by the two declarations:

```
const char* strchr(const char* \&\fP\f6s\&\fP\f5, int \&\fP\f6c\&\fP\f5);.ix "[st
```

3 both of which have the same behavior as the original declaration.

17.2.3.3 \&\f7strpbrk(const\ char*, const\ char*)\fP\& [lib.strpbrk]

1 The function signature \&\f5strpbrk(const char*, const char*)\fP\&, declared in \&\f5<string.h>\fP\& in the C Standard, does not have the declaration:

```
char* strpbrk(const char* \&\fP\f6s1\&\fP\f5, const char* \&\fP\f6s2\&\fP\f5);.ix
```

2 in this International Standard. Its declaration in \&\f5<string.h>\fP\& is replaced by the two declarations:

```
const char* strpbrk(const char* \&\fP\f6s1\&\fP\f5, const char* \&\fP\f6s2\&\fP\f
```

3 both of which have the same behavior as the original function signature.

17.2.3.4 \&\f7strrchr(const\ char*, int)\fP\& [lib.strrchr]

1 The function signature \&\f5strrchr(const char*, int)\fP\&, declared in \&\f5<string.h>\fP\& in the C Standard, does not have the declaration:

```
char* strrchr(const char* \&\fP\f6s\&\fP\f5, int \&\fP\f6c\&\fP\f5);.ix "[strrchr
```

2 in this International Standard. Its declaration in \&\f5<string.h>\fP\& is replaced by the two declarations:

```
const char* strrchr(const char* \&\fP\f6s\&\fP\f5, int \&\fP\f6c\&\fP\f5);.ix "[s
```

3 both of which have the same behavior as the original declaration.

17.2.3.5 \&\f7strstr(const\ char*, const\ char*)\fP\& [lib.strstr]

1 The function signature \&\f5strstr(const char*, const char*)\fP\&, declared in \&\f5<string.h>\fP\& in the C Standard, does not have the declaration:

```
char* strstr(const char* \&\fP\f6s1\&\fP\f5, const char* \&\fP\f6s2\&\fP\f5);.ix
```

2 in this International Standard. Its declaration in \&\f5<string.h>\fP\& is replaced by the two declarations:

```
const char* strstr(const char* \&\fP\f6s1\&\fP\f5, const char* \&\fP\f6s2\&\fP\f5
```

3 both of which have the same behavior as the original declaration.

17.2.4 Modifications to behavior [lib.mods.to.behavior]

17.2.4.1 Macro \&\f7offsetof\fP\& [lib.offsetof]

1 The macro \&\f5offsetof(\&\fP\f6type\&\fP\f5, \&\fP\f6member-designator\&\fP\f5)\fP\&, defined in \&\f5<stddef.h>\fP\&, accepts a restricted set of \&\f6type\fP\& arguments in this International Standard. \&\f6type\fP\& shall be a POD structure or a POD union.

17.2.4.2 \&\f7longjmp(jmp_buf, int)\fP\& [lib.longjmp]

1 The function signature \&\f5longjmp(jmp_buf \&\fP\f6jbuf\&\fP\f5, int \&\fP\f6val\&\fP\f5)\fP\&, declared in \&\f5<setjmp.h>\fP\&, 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 \&\f5longjmp(\&\fP\f6jbuf\&\fP\f5, \&\fP\f6val\&\fP\f5)\fP\& at the throw point that transfers control to the same (destination) point has undefined behavior.

17.2.4.3 Storage allocation functions [lib.storage.allocation.functions]

1 The function signatures \&\f5calloc(size_t)\fP\&, \&\f5malloc(size_t)\fP\&, and \&\f5realloc(void*, size_t)\fP\&, declared in \&\f5<stdlib.h>\fP\&, do not attempt to allocate storage by calling \&\f5operator new(size_t)\fP\&, declared in \&\f5<new>\fP\&.

17.2.4.4 \&\f7atexit(void (*)(void))\fP\& [lib.atexit]

1 The function signature \&\f5atexit(void (*\&\fP\f6f\&\fP\f5)(void))\fP\&, declared in \&\f5<stdlib.h>\fP\&, has additional behavior in this International Standard:

— For the execution of a function registered with \&\f5atexit\fP\&, if control leaves the function because it provides no handler for a thrown exception, \&\f5terminate()\fP\& is called.

17.2.4.5 \&\f7exit(int)\fP\& [lib.exit]

1 The function signature \&\f5exit(int \&\fP\f6status\&\fP\f5)\fP\&, declared in \&\f5<stdlib.h>\fP\&, has additional behavior in this International Standard:

— First, all functions \&\f6f\fP\& registered by calling \&\f5atexit(\&\fP\f6f\&\fP\f5)\fP\&, are called, in the reverse order of their registration.*f The function signature \&\f5atexit(void (*)())\fP\&, is declared in \&\f5<stdlib.h>\fP\&.

— Next, all static objects are destroyed in the reverse order of their construction. (Automatic objects are not destroyed as a result of calling \&\f5exit(int)\fP\&.)*f

— Next, all open C streams (as mediated by the function signatures declared in \&\f5<stdio.h>\fP\&) with unwritten buffered data are flushed, all open C streams are closed, and all files created by calling \&\f5tmpfile()\fP\& are removed.*f The function signature \&\f5tmpfile()\fP\& is declared in \&\f5<stdio.h>\fP\&.

— Finally, control is returned to the host environment. If \&\f6status\fP\& is zero or \&\f5EXIT_SUCCESS\fP\&, an implementation-defined form of the status \&\f2successful termination\fP\& is returned. If \&\f6status\fP\& is \&\f5EXIT_FAILURE\fP\&, an implementation-

[79] A function is called for every time it is registered.

[80] Automatic objects are all destroyed in a program whose function \&\f5main\fP\& contains no automatic objects and executes the call to \&\f5exit\fP\&. Control can be transferred directly to such a \&\f5main\fP\& by throwing an exception that is caught in \&\f5main\fP\&.

[81] Any C streams associated with \&\f5cin\fP\&, \&\f5cout\fP\&, etc. are flushed and closed when static objects are destroyed in the previous phase.

defined form of the status \&\f2unsuccessful termination\fP\& is returned. Otherwise the status returned is implementation-defined. The macros \&\f5EXIT_FAILURE\fP\& and \&\f5EXIT_SUCCESS\fP\& are defined in \&\f5<stdlib.h>\fP\&.

2 The function signature \&\f5exit(int)\fP\& never returns to its caller.

17.3 Language support [lib.language.support]

1 This subclause 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.

17.3.1 Header <defines> [lib.header.defines]

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

2 The constant is:

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

3 which is the largest representable value of type `size_t`.

17.3.1.1 Type `fvoid_t` [lib.fvoid.t]

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

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

17.3.1.2 Type `ptrdiff_t` [lib.ptrdiff.t]

```
typedef T ptrdiff_t;
```

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

17.3.1.3 Type `size_t` [lib.size.t]

```
typedef T size_t;
```

1 The type `size_t` is a synonym for *T*, the implementation-defined unsigned integral type of the result of the `sizeof` operator.

17.3.1.4 Type `wint_t` [lib.wint.t]

```
typedef T wint_t;
```

1 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.[82)]

[82)] 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`.

17.3.1.5 Type `capacity` [lib.capacity]

```
typedef T capacity;
static const capacity default_size;
static const capacity reserve;
```

1 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.

17.3.2 Header `<exception>` [lib.header.exception]

> **Box 86**
>
> **Library WG issue:** Michael Vilot, November 22, 1993
>
> The San Diego rewrite dropped all uses of exception specifications. [See: X3J16/93-0012R1, 93-0013R1, 93-0060, and 93-0064, as voted upon and accepted.]
>
> Dropping exception specifications was not a decision the Library WG reached. They need to be retained until we make an explicit decision to remove them.

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

17.3.2.1 Class `exception` [lib.exception]

> **Box 87**
>
> **Library WG issue:** Charles Allison, January 3, 1994
>
> What is the current state of `char *` vs. `string` arguments to xmsg and xalloc constructors. Did we officially decide that we shouldn't use string? I notice that 17 use null-terminated strings.

> **Box 88**
>
> **Library WG issue:** Michael Vilot, November 22, 1993
>
> The use of a virtual `.raise()` member function, instead of actually throwing exceptions, is a significant departure from the intent of the language. The rationale, ‘‘to provide a central point for debugging hooks,’’ seems to be inappropriate overspecification. It precludes other options that would achieve the same goal.

```
class exception {
public:
        typedef void (*raise_handler)(exception&);
        static raise_handler set_raise_handler(raise_handler handler_arg);
        exception(const string& what_arg);
        virtual ~exception();
        void raise();
        virtual string what() const;
protected:
        exception();
        virtual void do_raise();
private:
//      static raise_handler handler;     exposition only
//      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. Every exception *ex* thrown by a function defined within the Standard C++ library is thrown by evaluating an expression of the form `ex.raise()`. The class maintains a ttatic *raise handler* that designates a function to be called by the member function `raise`.

2 The class defines a member type `raise_handler` and maintains several kinds of data. For the sake of exposition, the stored data is presented here as:

— `static raise_handler` *handler*, points to the function called by the member function `raise`. Its initial value designates no function to be called;

— `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`.

17.3.2.1.1 Type `exception::raise_handler` [lib.exception::raise.handler]

```
typedef void (*raise_handler)(exception&);
```

1 The type `raise_handler` describes a pointer to a function called by the member function `raise` to perform operations common to all objects of class `exception`.

17.3.2.1.2 `exception::set_raise_handler(raise_handler)` [lib.exception::set.raise.handler]

```
static raise_handler set_raise_handler(raise_handler handler_arg);
```

1 Assigns *handler_arg* to *handler* and then returns the previous value stored in *handler*.

17.3.2.1.3 `exception::exception(const string&)` [lib.cons.exception.str]

```
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.

17.3.2.1.4 `exception::~exception()` [lib.des.exception]

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

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

17.3.2.1.5 `exception::raise()` [lib.exception::raise]

```
void raise();
```

1 If *`handler`* is nonzero, calls `(*`*`handler`*`)(*this)`. The function then calls `do_raise()`, then evaluates the expression `throw *this`.

17.3.2.1.6 `exception::what()` [lib.exception::what]

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

1 If *`desc`* is not a null pointer, returns `string(`*`desc`*`)`. Otherwise, the value returned is implementation defined.

17.3.2.1.7 `exception::exception()` [lib.cons.exception]

```
exception();
```

1 Constructs an object of class `exception` and initializes *`desc`* to an unspecified value and *`alloced`* to zero.[83)]

17.3.2.1.8 `exception::do_raise()` [lib.exception::do.raise]

```
virtual void do_raise();
```

1 Called by the member function `raise` to perform operations common to all objects of a class derived from `exception`. The default behavior is to return.

17.3.2.2 Class `logic` [lib.logic]

```
class logic : public exception {
public:
        logic(const string& what_arg);
        virtual ~logic();
//      virtual string what() const;     inherited
protected:
//      virtual void do_raise();         inherited
};
```

1 The class `logic` 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.

17.3.2.2.1 `logic::logic(const string&)` [lib.cons.logic]

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

1 Constructs an object of class `logic`, initializing the base class with `exception(`*`what_arg`*`)`.

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

17.3.2.2.2 `logic::~logic()` [lib.des.logic]

```
virtual ~logic();
```

1 Destroys an object of class `logic`.

17.3.2.2.3 `logic::what()` [lib.logic::what]

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

1 Behaves the same as `exception::what()`.

17.3.2.2.4 `logic::do_raise()` [lib.logic::do.raise]

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

1 Behaves the same as `exception::do_raise()`.

17.3.2.3 Class `runtime` [lib.runtime]

```
class runtime : public exception {
public:
        runtime(const string& what_arg);
        virtual ~runtime();
//      virtual string what();  inherited
protected:
//      virtual void do_raise();        inherited
        runtime();
};
```

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

17.3.2.3.1 `runtime::runtime(const string&)` [lib.cons.runtime.str]

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

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

17.3.2.3.2 `runtime::~runtime()` [lib.des.runtime]

```
virtual ~runtime();
```

1 Destroys an object of class `runtime`.

17.3.2.3.3 `runtime::what()` [lib.runtime::what]

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

1 Behaves the same as `exception::what()`.

17.3.2.3.4 `runtime::do_raise()` [lib.runtime::do.raise]

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

1 Behaves the same as `exception::do_raise()`.

17.3.2.3.5 runtime::runtime() | [lib.cons.runtime]

```
runtime();
```

1 Constructs an object of class runtime, initializing the base class with exception().

17.3.2.4 Class bad_cast | [lib.bad.cast]

```
class bad_cast : public logic {
public:
        bad_cast(const string& what_arg);
        virtual ~bad_cast();
//      virtual string what() const;     inherited
protected:
//      virtual void do_raise();         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.

17.3.2.4.1 bad_cast::bad_cast(const string&) | [lib.cons.bad.cast]

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

1 Constructs an object of class bad_cast, initializing the base class with logic(*what_arg*.

17.3.2.4.2 bad_cast::~bad_cast() | [lib.des.bad.cast]

```
virtual ~bad_cast();
```

1 Destroys an object of class bad_cast.

17.3.2.4.3 bad_cast::what() | [lib.bad.cast::what]

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

1 Behaves the same as exception::what().

17.3.2.4.4 bad_cast::do_raise() | [lib.bad.cast::do.raise]

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

1 Behaves the same as exception::do_raise().

17.3.2.5 Class invalid_argument | [lib.invalid.argument]

```
class invalid_argument : public domain {
public:
        invalid_argument(const string& what_arg);
        virtual ~invalid_argument();
//      virtual string what() const;     inherited
protected:
//      virtual void do_raise();         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.

17.3.2.5.1 invalid_argument::invalid_argument(const string&) [lib.cons.invalid.argument]

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

1 Constructs an object of class `invalid_argument`, initializing the base class with `domain(what_arg)`.

17.3.2.5.2 invalid_argument::~invalid_argument() [lib.des.invalid.argument]

```
virtual ~invalid_argument();
```

1 Destroys an object of class `invalid_argument`.

17.3.2.5.3 invalid_argument::what() [lib.invalid.argument::what]

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

1 Behaves the same as `exception::what()`.

17.3.2.5.4 invalid_argument::do_raise() [lib.invalid.argument::do.raise]

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

1 Behaves the same as `exception::do_raise()`.

17.3.2.6 Class length_error [lib.length.error]

```
class length_error : public domain {
public:
        length_error(const string& what_arg);
        virtual ~length_error();
//      virtual string what() const;    inherited
protected:
//      virtual void do_raise();        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`.

17.3.2.6.1 length_error::length_error(const string&) [lib.cons.length.error]

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

1 Constructs an object of class `length_error`, initializing the base class with `domain(what_arg)`.

17.3.2.6.2 length_error::~length_error() [lib.des.length.error]

```
virtual ~length_error();
```

1 Destroys an object of class `length_error`.

17.3.2.6.3 length_error::what() [lib.length.error::what]

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

1 Behaves the same as `exception::what()`.

17.3.2.6.4 `length_error::do_raise()` [lib.length.error::do.raise]

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

1 Behaves the same as `exception::do_raise()`.

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

```
class out_of_range : public domain {
public:
        out_of_range(const string& what_arg);
        virtual ~out_of_range();
//      virtual string what() const;    inherited
protected:
//      virtual void do_raise();        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.

17.3.2.7.1 `out_of_range::out_of_range(const string&)` [lib.cons.out.of.range]

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

1 Constructs an object of class `out_of_range`, initializing the base class with `domain(`*`what_arg`*`)`.

17.3.2.7.2 `out_of_range::~out_of_range()` [lib.des.out.of.range]

```
virtual ~out_of_range();
```

1 Destroys an object of class `out_of_range`.

17.3.2.7.3 `out_of_range::what()` [lib.out.of.range::what]

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

1 Behaves the same as `exception::what()`.

17.3.2.7.4 `out_of_range::do_raise()` [lib.out.of.range::do.raise]

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

1 Behaves the same as `exception::do_raise()`.

17.3.2.8 Class `overflow` [lib.overflow]

```
class overflow : public range {
public:
        overflow(const string& what_arg);
        virtual ~overflow();
//      virtual string what() const;    inherited
protected:
//      virtual void do_raise();        inherited
};
```

1 The class `overflow` 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.

17.3.2.8.1 `overflow::overflow(const string&)` | [lib.cons.overflow]

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

1 Constructs an object of class `overflow`, initializing the base class with `range(what_arg)`. *

17.3.2.8.2 `overflow::~overflow()` [lib.des.overflow]

```
virtual ~overflow();
```

1 Destroys an object of class `overflow`.

17.3.2.8.3 `overflow::what()` | [lib.overflow::what]

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

1 Behaves the same as `exception::what()`.

17.3.2.8.4 `overflow::do_raise()` [lib.overflow::do.raise]

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

1 Behaves the same as `exception::do_raise()`.

17.3.2.9 Class `alloc` | [lib.alloc]

```
class alloc : public runtime {
public:
        alloc();
        virtual ~alloc();
//      virtual void what() const;       inherited
protected:
//      virtual void do_raise();         inherited
private:
//      static string alloc_msg;         exposition only
};
```

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

17.3.2.9.1 `alloc::alloc()` | [lib.cons.alloc]

```
alloc();
```

1 Constructs an object of class `alloc`, initializing the base class with `runtime()`.

17.3.2.9.2 `alloc::~alloc()` | [lib.des.alloc]

```
virtual ~alloc();
```

1 Destroys an object of class `alloc`.

17.3.2.9.3 alloc::what() [lib.alloc::what]

```
//      virtual int what() const;       inherited
```

1 Returns an implementation-defined value.[84)]

17.3.2.9.4 alloc::do_raise() [lib.alloc::do.raise]

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

1 Behaves the same as `exception::do_raise()`.

17.3.2.10 Class domain [lib.domain]

```
class domain : public logic {
public:
        domain(const string& what_arg);
        virtual ~domain();
//      virtual string what() const;    inherited
protected:
//      virtual void do_raise();        inherited
};
```

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

17.3.2.10.1 domain::domain(const string&) [lib.cons.domain]

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

1 Constructs an object of class `domain`, initializing the base class with `logic(`*what_arg*`)`.

17.3.2.10.2 domain::~domain() [lib.des.domain]

```
virtual ~domain();
```

1 Destroys an object of class `domain`.

17.3.2.10.3 domain::what() [lib.domain::what]

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

1 Behaves the same as `exception::what()`.

17.3.2.10.4 domain::do_raise() [lib.domain::do.raise]

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

1 Behaves the same as `exception::do_raise()`.

17.3.2.11 Class range [lib.range]

84) A possible return value is `&`*alloc_msg*.

```
class range : public runtime {
public:
        range(const string& what_arg);
        virtual ~range();
//      virtual string what() const;     inherited
protected:
//      virtual void do_raise();         inherited
};
```

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

17.3.2.11.1 `range::range(const string&)` [lib.cons.range]

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

1 Constructs an object of class `range`, initializing the base class with `runtime(`*what_arg*`)`.

17.3.2.11.2 `range::~range()` [lib.des.range]

```
virtual ~range();
```

1 Destroys an object of class `range`.

17.3.2.11.3 `range::what()` [lib.range::what]

```
//      virtual int what() const;        inherited
```

1 Behaves the same as `exception::what()`.

17.3.2.11.4 `range::do_raise()` [lib.range::do.raise]

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

1 Behaves the same as `exception::do_raise()`.

17.3.2.12 `set_terminate(fvoid_t*)` [lib.set.terminate]

Box 89

Library WG issue: Michael Vilot, November 22, 1993

Requiring global objects is an overspecification:

17.3.2.12 set_terminate ‘‘The function stores *new_p* in a static object
... The function returns the previous contents of terminate_handler.’’

17.3.2.13 ditto for unexpected_handler.

17.3.3.1 ditto for new_handler.

The treatment of all 3 handlers in 93-0148/N0355 was simpler and clearer. The San Diego rewrite amounts to overspecification, particularly in light of the ongoing interest in keeping this library viable in multi-threaded environments.

```
fvoid_t* set_terminate(fvoid_t* new_p);
```

1 Establishes a new handler for terminating exception processing. The function stores *new_p* in a static object that, for the sake of exposition, can be declared as:

```
fvoid_t* terminate_handler = &abort;
```

2 where the function signature `abort()` is defined in `<cstdlib>` (17.2.4.5). *new_p* shall not be a null pointer.

3 The function returns the previous contents of *terminate_handler*.

17.3.2.13 `set_unexpected(fvoid_t*)` [lib.set.unexpected]

```
fvoid_t* set_unexpected(fvoid_t* new_p);
```

1 Establishes a new handler for an unexpected exception thrown by a function with an *exception-specification.* The function stores *new_p* in a static object that, for the sake of exposition, can be declared as:

```
fvoid_t* unexpected_handler = &terminate;
```

2 *new_p* shall not be a null pointer.

3 The function returns the previous contents of *unexpected_handler*.

17.3.2.14 `terminate()` [lib.terminate]

```
void terminate();
```

1 Called by the implementation when exception handling must be abandoned for any of several reasons, such as:

— when a thrown exception has no corresponding handler;

— when a thrown exception determines that the the execution stack is corrupted;

— when a thrown exception calls a destructor that tries to transfer control to a calling function by throwing another exception.

2 Using the notation of subclause 17.3.2.12, the function evaluates the expression:

```
(*terminate_handler)()
```

3 The required behavior of any function called by this expression is to terminate execution of the program without returning to the caller. The default behavior is to call `abort()`, declared in `<cstdlib>` (17.2.4.5).

17.3.2.15 `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.* Using the notation of subclause 17.3.2.13, the function evaluates the expression:

```
(*unexpected_handler)()
```

2 The required behavior of any function called by this expression is to throw an exception or terminate execution of the program without returning to the caller. The called function may perform any of the following operations:

— rethrow the exception;

— throw another exception;

— call terminate();

— call either abort() or exit(int), declared in <cstdlib> (17.2.4.5).

3 The default behavior is to call terminate().

17.3.3 Header <new> [lib.header.new]

Box 90

Library WG issue: Michael Vilot, November 22, 1993

The wording has disappeared that required an implementation that uses the global versions of operator new and delete to pick up program-supplied versions that replace them.

1 The header <new> defines several functions that manage the allocation of storage in a program, as described in subclauses 5.3 and 12.5.

17.3.3.1 set_new_handler(fvoid_t*) [lib.set.new.handler]

Box 91

Library WG issue: Michael Vilot, November 22, 1993

This is part of a general issue on stating the requirements on types and functions *used by* the library.

Keeping a separate subsection for the handlers in 93-0148/N0355 also served two other purposes.:

First, it gave us a place to introduce appropriate typedefs. As indicated, ‘‘the type fvoid_t needs to be defined or replaced.’’ I suggest *replaced*. Actually, the use of fvoid_t is *less* precise than the use of the three handler typedefs in 93-0148/N0355.

Second, it gave us a place to describe the default implementation: the description of the new-handler in 93-0108 section 17.3.2.5 seems out of place, and artifically removed from 17.3.2.2.

We should retain the wording in 93-0148/N0355, because it avoids another global name and it conveys the semantics of each handler more succinctly.

```
fvoid_t* set_new_handler(fvoid_t* new_p);
```
*

1 Establishes a new handler to be called by the default versions of operator new(size_t) and operator new[](size_t) when they cannot satisfy a request for additional storage. The function stores *new_p* in a static object that, for the sake of exposition, can be called *new_handler* and can be declared as:

```
fvoid_t* new_handler = &new_hand;
```

2 where, in turn, *new_hand* can be defined as:

```
static void new_hand()
{       // raise alloc exception
        static const alloc ex("operator new");
        ex.raise();
}
```

3 The function returns the previous contents of *new_handler*.

17.3.3.2 `operator delete(void*)` [lib.op.delete]

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

1 Called by a `delete` expression to render the value of *ptr* invalid. The program can define a function with this function signature that displaces the default version defined by the Standard C++ library. The required behavior is to accept a value of *ptr* that is null or that was returned by an earlier call to `operator new(size_t)`.

2 The default behavior for a null value of *ptr* is to do nothing. Any other value of *ptr* shall be a value returned earlier by a call to the default `operator new(size_t)`.[85] The default behavior for such a non-null value of *ptr* is to reclaim storage allocated by the earlier call to the default `operator new(size_t)`. 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(size_t)`, `malloc(size_t)`, or `realloc(void*, size_t)`, declared in `<cstdlib>` (17.2.4.3).

17.3.3.3 `operator delete[](void*)` [lib.op.delete.array]

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

1 Called by a `delete[]` expression to render the value of *ptr* invalid. The program can define a function with this function signature that displaces the default version defined by the Standard C++ library.

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

3 The default behavior for a null value of *ptr* is to do nothing. Any other value of *ptr* shall be a value returned earlier by a call to the default `operator new[](size_t)`.[86] The default behavior for such a non-null value of *ptr* is to reclaim storage allocated by the earlier call to the default `operator new[](size_t)`. 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(size_t)`, `malloc(size_t)`, or `realloc(void*, size_t)`, declared in `<cstdlib>` (17.2.4.3).

17.3.3.4 `operator new(size_t)` [lib.op.new]

[85] 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.

[86] 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.

Box 92

Library WG issue: Michael Vilot, November 22, 1993

The 3 paragraphs of 93-0148/N0355 section 17.1.1 should be retained.

The change to split these out and reorder them is counterproductive. By repeating the descriptions, you've introduced a lot of wordiness and potential for error. In particular, the wording about storage allocation and reclamation lost something in the translation.

The words in 93-0148/N0355 section 17.1.1.1, paragraph 4, were intentionally copied, in order, from the C standard. The Rationale statement clearly expresses our intent to pattern our description of storage management after the same words for `malloc/calloc/free` (17.2.4.3).

The concept of ''invalidating'' is probably more appropriate wording. Let's see if we can't keep the advantages of the wording of 93-0148/N0355 with this suggested improvement.

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

1 Called by a `new` expression to allocate *`size`* bytes of storage suitably aligned to represent any object of that size. The program can define a function with this function signature that displaces the default version defined by the Standard C++ library.

2 The required behavior is to return a non-null pointer only if storage can be allocated as requested. Each such allocation shall yield a pointer to storage disjoint from any other allocated storage. The order and contiguity of storage allocated by successive calls to `operator new(size_t)` is unspecified. The initial stored value is unspecified. The returned pointer points to the start (lowest byte address) of the allocated storage. If *`size`* is zero, the value returned shall not compare equal to any other value returned by `operator new(size_t)`.[87)]

3 The default behavior is to execute a loop. Within the loop, the function first attempts to allocate the requested storage. Whether the attempt involves a call to the Standard C library function `malloc` is unspecified. If the attempt is successful, the function returns a pointer to the allocated storage. Otherwise (using the notation of subclause 17.3.3.1), if *`new_handler`* is a null pointer, the result is implementation-defined.[88)] Otherwise, the function evaluates the expression `(*`*`new_handler`*`)()`. If the called function returns, the loop repeats. The loop terminates when an attempt to allocate the requested storage is successful or when a called function does not return.

4 The required behavior of a function called by `(*`*`new_handler`*`)()` is to perform one of the following operations:

— make more storage available for allocation and then return;

— execute an expression of the form *`ex`*`.raise()`, where *`ex`* is an object of type `alloc`, declared in `<exception>`;

— call either `abort()` or `exit(int)`, declared in `<cstdlib>` (17.2.4.5).

5 The default behavior of a function called by `(*`*`new_handler`*`)()` is described by the function *`new_hand`*, as shown in subclause 17.3.3.1.

87) The value cannot legitimately compare equal to one that has been invalidated by a call to `operator delete(size_t)`, since any such comparison is an invalid operation.

88) 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++.

6 The order and contiguity of storage allocated by successive calls to `operator new(size_t)` is unspecified, as are the initial values stored there.

17.3.3.5 `operator new[](size_t)` [lib.op.new.array]

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

1 Called by a `new[]` expression to allocate *`size`* bytes of storage suitably aligned to represent any array object of that size or smaller. [89] The program can define a function with this function signature that displaces the default version defined by the Standard C++ library.

2 The required behavior is the same as for `operator new(size_t)`.

3 The default behavior is to return `operator new(`*`size`*`)`.

17.3.3.6 `operator new(size_t, void*)` [lib.placement.op.new]

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

1 Returns *`ptr`*.

17.3.3.7 `operator new[](size_t, void*)` [lib.placement.op.new.array]

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

1 Returns *`ptr`*.

17.3.4 Header `<typeinfo>` [lib.header.typeinfo]

1 The header `<typeinfo>` defines two types associated with type information generated by the implementation.

17.3.4.1 Class `bad_type_id` [lib.bad.type.id]

```
class bad_type_id : public logic {
public:
        bad_type_id();
        virtual ~bad_type_id();
protected:
//      virtual void do_raise();         inherited
};
```

1 The class `bad_type_id` 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)`.

17.3.4.1.1 `bad_type_id::bad_type_id()` [lib.cons.bad.type.id]

```
bad_type_id();
```

1 Constructs an object of class `bad_type_id`, initializing the base class `logic` with an unspecified constructor.

---

[89] 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.

17.3.4.1.2 bad_type_id::~bad_type_id() [lib.des.bad.type.id]

```
virtual ~bad_type_id();
```

1 Destroys an object of class bad_type_id.

17.3.4.1.3 bad_type_id::do_raise() [lib.bad.type.id::do.raise]

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

1 Behaves the same as exception::do_raise().

17.3.4.2 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 char* name() const;
private:
//      const char* name;          exposition only
//      T desc;  exposition only
        type_info(const type_info& rhs);
        type_info& operator=(const type_info& rhs);
};
```

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*.

17.3.4.2.1 type_info::~type_info() [lib.des.type.info]

```
virtual ~type_info();
```

1 Destroys an object of type type_info.

17.3.4.2.2 type_info::operator==(const type_info&) [lib.type.info::op==]

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

1 Compares the value stored in *desc* with rhs.*desc*. Returns a nonzero value if the two values represent the same type.

17.3.4.2.3 type_info::operator!=(const type_info&) [lib.type.info::op!=]

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

1 Returns a nonzero value if !(*this == *rhs*).

17.3.4.2.4 `type_info::before(const type_info&)` [lib.type.info::before]

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

1 Compares the value stored in *desc* with *rhs.desc*. Returns a nonzero value if `*this` precedes *rhs* in the collation order.

17.3.4.2.5 `type_info::name()` [lib.type.info::name]

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

1 Returns *name*.

17.3.4.2.6 `type_info::type_info(const type_info&)` [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*. [90]

17.3.4.2.7 `type_info::operator=(const type_info&)` [lib.type.info::op=]

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

1 Assigns *rhs.name* to *name* and *rhs.desc* to *desc*. The function returns `*this`.

17.4 Input/output [lib.input/output]

Box 93

Library WG issue: Nobuo Saito, January 17, 1994

In the current library draft, there is nothing about the I/O functions for wide characters. For Asian nations like Japan, it is crucial to be able to use the multibyte characters flexibly in all the areas like I/O functions. Therefore, it is very important to prepare I/O functions for the wide characters in the current library draft.

We also want to prepare the sophisticated solutions using the high functionalities in the C++ language(like the overloading). Then, the following design policy will be reasonable.

1) Use the overloaded function names both for characters and wide characters.

2) Use the character base buffers in the streambuf.

Comment (Library WG meeting, San Diego, 3/8/94):

See pending proposals:
94-0050/N0437 Takanori Adachi "An inserter and extractor for the unified string class"
94-0052/N0439 Norohiro Kumagai "A Proposal for Widechar IOstream"

1 This subclause describes a number of headers that together support input, output, and internal data conversions.

90) 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.

17.4.1 Header `<ios>` [lib.header.ios]

1 The Header `<ios>` defines a type and several function signatures for controlling how to interpret text input from a sequence of characters and how to generate text output to a sequence of characters.

17.4.1.1 Class `ios` [lib.ios]

```
class ios {
public:
        class failure : public exception {
        public:
                failure(const string& what_arg);
                virtual ~failure();
//              virtual string what() const;     inherited
        protected:
//              virtual void do_raise();         inherited
        };
        typedef T1 fmtflags;
        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
        };
        ios(streambuf* sb_arg);
        virtual ~ios();
        operator bool() const
        bool operator!() const
        ios& copyfmt(const ios& rhs);
        ostream* tie() const;
        ostream* tie(ostream* tiestr_arg);
        streambuf* rdbuf() const;
        streambuf* rdbuf(streambuf* sb_arg);
        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);
        int fill() const;
        int fill(int ch);
        int precision() const;
        int precision(int prec_arg);
        int width() const;
        int width(int wide_arg);
        locale imbue(const locale& loc_arg);
        locale rdloc() const;
        static int xalloc();
        long& iword(int index_arg);
        void*& pword(int index_arg);
protected:
        ios();
        void init(streambuf* sb_arg);
private:
//      streambuf* sb;    exposition only
//      ostream* 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
};
```

1 The class `ios` serves as a base class for the classes `istream` and `ostream`. It defines several member types:

— a class `failure` derived from `exception`;

— a class `Init`;

— three bitmask types, `fmtflags`, `iostate`, and `openmode`;

— an enumerated type, `seekdir`.

2 It maintains several kinds of data:

— a pointer to a *stream buffer,* an object of class `streambuf`, that controls sources (input) and sinks (output) of character sequences;

— 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 programffor its private use.

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

— `streambuf*` *sb*, points to the stream buffer;

— <F1s2B>ostream* >`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.

17.4.1.1.1 Class `ios::failure` [lib.ios::failure]

Box 94

Library WG issue: Jerry Schwarz, September 28, 1993

The San Diego rewrite drops the `ios` component from `ios::failure`.

```
class failure : public exception {
public:
        failure(const string& where_arg);
        virtual ~failure();
//      virtual string what() const;     inherited
protected:
//      virtual void do_raise();         inherited
};
```

1 The class `failure` defines the base class for the types of all objects thrown as exceptions, by functions in the Standard C++ library, to report errors detected during stream buffer operations.

17.4.1.1.1.1 `ios::failure::failure(const string&)` [lib.cons.ios::failure]

```
failure(const char* where_arg = 0, const char* why_arg = 0);
```

1 Constructs an object of class `failure`, initializing the base class with `exception(`*what_arg*`)`.

17.4.1.1.1.2 `ios::failure::~failure()` [lib.des.ios::failure]

```
virtual ~failure();
```

1 Destroys an object of class `failure`.

17.4.1.1.1.3 `ios::failure::what()` [lib.ios::failure::what]

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

1 Behaves the same as `exception::what()`.

17.4.1.1.1.4 `ios::failure::do_raise()` [lib.ios::failure::do.raise]

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

1 Behaves the same as `exception::do_raise()`.

17.4.1.1.2 Type `ios::fmtflags` [lib.ios::fmtflags]

```
typedef T1 fmtflags;
```

1 The type `fmtflags` is a bitmask type (indicated here as *T1*) with the elements:

— `dec`, set to convert integer input or to generate integer output in decimal base;

— `fixed`, set to generate floating-point output in fixed-point notation;

— `hex`, set to convert integer input or to generate integer output in hexadecimal base;

— `internal`, set to add fill characters at a designated internal point in certain generated output;

— `left`, set to add fill characters on the left (initial positions) of certain generated output;

— `oct`, set to convert integer input or to generate integer output in octal base;

— `right`, set to add fill characters on the right (final positions) of certain generated output;

— `scientific`, set to generate floating-point output in scientific notation;

— `showbase`, set to generate a prefix indicating the numeric base of generated integer output;

— `showpoint`, set to generate a decimal-point character unconditionally in generated floating-point output;

— `showpos`, set to generate a + sign in non-negative generated numeric output;

— `skipws`, set to skip leading white space before certain input operations;

— `unitbuf`, set to flush output after each output operation;

— `uppercase`, set to replace certain lowercase letters with their uppercase equivalents in generated output.

2 Type `fmtflags` also defines the constants:

— `adjustfield`, the value `left | right | internal`;

— `basefield`, the value `dec | oct | hex`;

— `floatfield`, the value `scientific | fixed`.

17.4.1.1.3 Type `ios::iostate` [lib.ios::iostate]

```
typedef T2 iostate;
```

1 The type `iostate` is a bitmask type (indicated here as *T2*) with the elements:

— `badbit`, set to indicate a loss of integrity in an input or output sequence (such as an irrecoverable read error from a file);

— `eofbit`, set to indicate that an input operation reached the end of an input sequence;

— `failbit`, set to indicate 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.

17.4.1.1.4 Type `ios::openmode` [lib.ios::openmode]

> **Box 95**
>
> **Library WG issue:** Jerry Schwarz, January 3, 1994
>
> `openmode`'s are used in contexts that have nothing to do with files (or open for that matter). The name is obviously a misnomer (as are many of the names in iostreams).
>
> Not fixed.

```
typedef T3 openmode;
```

1 The type `openmode` is a bitmask type (indicated here as *T3*) with the elements:

— `app`, set to seek to end-of-file before each write to the file;

— `ate`, set to open a file and seek to end-of-file immediately after opening the file;

— `binary`, set to perform input and output in binary mode (as opposed to text mode);

— `in`, set to open a file for input;

— `out`, set to open a file for output;

— `trunc`, set to truncate an existing file when opening it.

17.4.1.1.5 Type `ios::seekdir` [lib.ios::seekdir]

```
typedef T4 seekdir;
```

1 The type `seekdir` is an enumerated type (indicated here as *T4*) with the elements:

— `beg`, to request a seek (positioning for subsequent input or output within a sequence) relative to the beginning of the stream;

— `cur`, to request a seek relative to the current position within the sequence;

— `end`, to request a seek relative to the current end of the sequence.

17.4.1.1.6 Class `ios::Init` [lib.ios::init]

```
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>` (17.2). 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.

17.4.1.1.6.1 `ios::Init::Init()` [lib.cons.ios::init]

```
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` (17.4.9.1), `cout` (17.4.9.2), `cerr` (17.4.9.3), and `clog` (17.4.9.4). In any case, the function then adds one to the value stored in *`init_cnt`*.

17.4.1.1.6.2 `ios::Init::~Init()` [lib.des.ios::init]

```
~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()`.

17.4.1.1.7 `ios::ios(streambuf*)` [lib.cons.ios.sb]

```
ios(streambuf* sb_arg);
```

1 Constructs an object of class `ios`, assigning initial values to its member objects by calling `init(`*`sb_arg`*`)`.

17.4.1.1.8 `ios::~ios()` [lib.des.ios]

```
virtual ~ios();
```

1 Destroys an object of class `ios`.

17.4.1.1.9 `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`*.

17.4.1.1.10 `ios::operator!()` [lib.ios::operator!]

```
bool operator!() const
```

1 Returns a nonzero value if `failbit | badbit` is set in *state*.

17.4.1.1.11 **ios::copyfmt(const ios&)** [lib.ios::copyfmt]

```
ios& copyfmt(const ios& 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*`)`.

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 The function returns `*this`.

17.4.1.1.12 **ios::tie()** [lib.ios::tie]

```
ostream* tie() const;
```

1 Returns *tiestr*.

17.4.1.1.13 **ios::tie(ostream*)** [lib.ios::tie.os]

```
ostream* tie(ostream* tiestr_arg);
```

1 Assigns *tiestr_arg* to *tiestr* and then returns the previous value stored in *tiestr*.

17.4.1.1.14 **ios::rdbuf()** [lib.ios::rdbuf]

```
streambuf* rdbuf() const;
```

1 Returns *sb*.

17.4.1.1.15 **ios::rdbuf(streambuf*)** [lib.ios::rdbuf.sb]

```
streambuf* rdbuf(streambuf* sb_arg);
```

1 Assigns *sb_arg* to *sb*, then calls `clear()`. The function returns the previous value stored in *sb*.

17.4.1.1.16 **ios::rdstate()** [lib.ios::rdstate]

```
iostate rdstate() const;
```

1 Returns *state*.

17.4.1.1.17 **ios::clear(iostate)** [lib.ios::clear.ios]

> **Box 96**
>
> **Library WG issue:** Jerry Schwarz, September 28, 1993
>
> The San Diego rewrite adds `xmsg` arguments to `ios::clear` and `ios::setstate`.

```
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, the function returns. Otherwise, the function calls *fail*`.raise()` for an object *fail* of class *failure*, constructed with argument values that are implementation-defined.

17.4.1.1.18 `ios::setstate(iostate)` [lib.ios::setstate.ios]

```
void setstate(iostate state_arg);
```

1 Calls `clear(`*state* `|` *state_arg*`)`.

17.4.1.1.19 `ios::good()` [lib.ios::good]

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

1 Returns a nonzero value if *state* is zero.

17.4.1.1.20 `ios::eof()` [lib.ios::eof]

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

1 Returns a nonzero value if `eofbit` is set in *state*.

17.4.1.1.21 `ios::fail()` [lib.ios::fail]

Box 97

Library WG issue: Jerry Schwarz, September 28, 1993

Should set `failbit` when the input can't be represented in the object.

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

1 Returns a nonzero value if `failbit` is set in *state*.

17.4.1.1.22 `ios::bad()` [lib.ios::bad]

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

1 Returns a nonzero value if `badbit` is set in *state*.

17.4.1.1.23 `ios::exceptions()` [lib.ios::exceptions]

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

1 Returns *except*.

17.4.1.1.24 `ios::exceptions(iostate)` [lib.ios::exceptions.ios]

```
void exceptions(iostate except_arg);
```

1 Assigns *except_arg* to *except*, then calls `clear(`*state*`)`.

17.4.1.1.25 `ios::flags()` [lib.ios::flags]

```
fmtflags flags() const;
```

1 Returns *fmtfl*.

17.4.1.1.26 `ios::flags(fmtflags)` [lib.ios::flags.f]

```
fmtflags flags(fmtflags fmtfl_arg);
```

1 Assigns *fmtfl_arg* to *fmtfl* and then returns the previous value stored in *fmtfl*.

17.4.1.1.27 `ios::setf(fmtflags)` [lib.ios::setf.f]

```
fmtflags setf(fmtflags fmtfl_arg);
```

1 Sets *fmtfl_arg* in *fmtfl* and then returns the previous value stored in *fmtfl*.

17.4.1.1.28 `ios::setf(fmtflags, fmtflags)` [lib.ios::setf.ff]

```
fmtflags setf(fmtflags fmtfl_arg, fmtflags mask);
```

1 Clears *mask* in *fmtfl*, sets *fmtfl_arg* `&` *mask* in *fmtfl*, and then returns the previous value stored in *fmtfl*.

17.4.1.1.29 `ios::unsetf(fmtflags)` [lib.ios::unsetf]

```
void unsetf(fmtflags mask);
```

1 Clears *mask* in *fmtfl*.

17.4.1.1.30 `ios::fill()` [lib.ios::fill]

```
int fill() const;
```

1 Returns *fill*.

17.4.1.1.31 `ios::fill(int)` [lib.ios::fill.i]

```
int fill(int fillch_arg);
```

1 Assigns *fillch_arg* to *fillch* and then returns the previous value stored in *fillch*.

17.4.1.1.32 `ios::precision()` [lib.ios::precision]

```
int precision() const;
```

1 Returns *prec*.

17.4.1.1.33 `ios::precision(int)` [lib.ios::precision.i]

```
int precision(int prec_arg);
```

1 Assigns *prec_arg* to *prec* and then returns the previous value stored in *prec*.

17.4.1.1.34 `ios::width()` [lib.ios::width]

```
int width() const;
```

1 Returns *wide*.

17.4.1.1.35 `ios::width(int)` [lib.ios::width.i]

```
int width(int wide_arg);
```

1 Assigns *wide_arg* to *wide* and then returns the previous value stored in *wide*.

17.4.1.1.36 `ios::imbue(const locale&)` [lib.ios::imbue]

```
locale imbue(const locale loc_arg);
```

1 Assigns *loc_arg* to *loc* and then returns the previous value stored in *loc*.

17.4.1.1.37 `ios::rdloc()` [lib.ios::rdloc]

```
locale rdloc() const;
```

1 Returns *loc*.

17.4.1.1.38 `ios::xalloc()` [lib.ios::xalloc]

Box 98

Library WG issue: Jerry Schwarz, September 28, 1993

Is it clear that `xalloc` doesn't have to start at zero?

```
static int xalloc();
```

1 Returns *index*++.

17.4.1.1.39 `ios::iword(int)` [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. The function returns *iarray*[*idx*]. After a subsequent call to `iword(int)` for the same object, the earlier return value may no longer be valid.[91)]

17.4.1.1.40 `ios::pword(int)` [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. The function returns *parray*[*idx*]. After a subsequent call to `pword(int)` for the same object, the earlier return value may no longer be valid.

91) 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.

17.4.1.1.41 ios::ios() [lib.cons.ios]

```
ios();
```
*

1 Constructs an object of class `ios`, assigning initial values to its member objects by calling `init(0)`.

17.4.1.1.42 ios::init(streambuf*) [lib.ios::init.sb]

```
void init(streambuf* sb_arg);
```

1 Assigns:

— *sb_arg* to *sb*;

— a null pointer to *tiestr*;

— `goodbit` to *state* if *sb_arg* is not a null pointer, otherwise `badbit` to *state*;

— `goodbit` to *except*;

— `skipws | dec` to *fmtfl*;

— zero to *wide*;

— 6 to *prec*;

— the space character to *fillch*;

— `locale::classic()` to *loc*;

— a null pointer to *iarray*;

— a null pointer to *parray*.

17.4.1.2 dec(ios&) [lib.dec]

```
ios& dec(ios& str);
```

1 Calls *str*`.setf(ios::dec, ios::basefield)` and then returns *str*.[92]

17.4.1.3 fixed(ios&) [lib.fixed]

```
ios& fixed(ios& str);
```

1 Calls *str*`.setf(ios::fixed, ios::floatfield)` and then returns *str*.

17.4.1.4 hex(ios&) [lib.hex]

```
ios& hex(ios& str);
```

1 Calls *str*`.setf(ios::hex, ios::basefield)` and then returns *str*.

17.4.1.5 internal(ios&) [lib.internal]

```
ios& internal(ios& str);
```

1 Calls *str*`.setf(ios::internal, ios::adjustfield)` and then returns *str*.

[92] The function signature `dec(ios&)` can be called by the function signature `ostream& stream::operator<<(ostream& (*)(ostream&))` to permit expressions of the form `cout << dec` to change the format flags stored in `cout`.

17.4.1.6 `left(ios&)` [lib.left]

```
ios& left(ios& str);
```

1 Calls *str*`.setf(ios::left, ios::adjustfield)` and then returns *str*.

17.4.1.7 `noshowbase(ios&)` [lib.noshowbase]

```
ios& noshowbase(ios& str);
```

1 Calls *str*`.unsetf(ios::showbase)` and then returns *str*.

17.4.1.8 `noshowpoint(ios&)` [lib.noshowpoint]

```
ios& noshowpoint(ios& str);
```

1 Calls *str*`.unsetf(ios::showpoint)` and then returns *str*.

17.4.1.9 `noshowpos(ios&)` [lib.noshowpos]

```
ios& noshowpos(ios& str);
```

1 Calls *str*`.unsetf(ios::showpos)` and then returns *str*.

17.4.1.10 `noskipws(ios&)` [lib.noskipws]

```
ios& noskipws(ios& str);
```

1 Calls *str*`.unsetf(ios::skipws)` and then returns *str*.

17.4.1.11 `nouppercase(ios&)` [lib.nouppercase]

```
ios& nouppercase(ios& str);
```

1 Calls *str*`.unsetf(ios::uppercase)` and then returns *str*.

17.4.1.12 `oct(ios&)` [lib.oct]

```
ios& oct(ios& str);
```

1 Calls *str*`.setf(ios::oct, ios::basefield)` and then returns *str*.

17.4.1.13 `right(ios&)` [lib.right]

```
ios& right(ios& str);
```

1 Calls *str*`.setf(ios::right, ios::adjustfield)` and then returns *str*.

17.4.1.14 `scientific(ios&)` [lib.scientific]

```
ios& scientific(ios& str);
```

1 Calls *str*`.setf(ios::scientific, ios::floatfield)` and then returns *str*.

17.4.1.15 `showbase(ios&)` [lib.showbase]

```
ios& showbase(ios& str);
```

1 Calls *str*`.setf(ios::showbase)` and then returns *str*.

17.4.1.16 showpoint(ios&) [lib.showpoint]

```
ios& showpoint(ios& str);
```

1 Calls *str*.setf(ios::showpoint) and then returns *str*.

17.4.1.17 showpos(ios&) [lib.showpos]

```
ios& showpos(ios& str);
```

1 Calls *str*.setf(ios::showpos) and then returns *str*.

17.4.1.18 skipws(ios&) [lib.skipws]

```
ios& skipws(ios& str);
```

1 Calls *str*.setf(ios::skipws) and then returns *str*.

17.4.1.19 uppercase(ios&) [lib.uppercase]

```
ios& uppercase(ios& str);
```

1 Calls *str*.setf(ios::uppercase) and then returns *str*.

17.4.2 Header <streambuf> [lib.header.streambuf]

1 The header <streambuf> defines a macro and three types that control input from and output to character sequences. |

2 The macro is:

— EOF, which expands to a negative integral constant expression, representable as type int, that is returned by several functions to indicate end-of-file (no more input from an input sequence or no more output permitted to an output sequence), or to indicate an invalid return value.[93] |

17.4.2.1 Type streamoff [lib.streamoff]

```
typedef T1 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. It is used to represent:

— a signed displacement, measured in bytes, from a specified position within a sequence;

— an absolute position within a sequence, not necessarily measured in uniform units.

2 In the second case, the value (streamoff)(-1) indicates an invalid position, or a position that cannot be represented as a value of type streamoff.

17.4.2.2 Class streampos [lib.streampos]

93) This macro is also defined, with the same value and meaning, in <cstdio>. |

Box 99

Library WG issue: Jerry Schwarz, January 3, 1994

Bill has a lot more experience with `fpos_t` than I do, but the reference to "`streamoff` that represents the position in `fp`" doesn't make sense to me. I thought that `fpos_t`'s could be magic cookies. What is important is the identity

```
long(streampos(n)) == n
```

Is it really possible in general to add an offset to an `fpos_t` without having a file to which it is attached?

Even if it is possible to do this arithmetic for `fpos_t`, it isn't necessarily the case for arbitrary `streambuf`'s. In particular it isn't possible for the `mbstreambuf` class proposed in x3j16/93- 0125.

The immediate problem is solved, but there is still a lot of discussion of adding offsets to `fpos_t`'s. This isn't an operation that the C standard allows, and I think it is a mistake to go beyond the C standard here. I'm not sure of the operational consequence of what Bill is doing.

Box 100

Library WG issue: Jerry Schwarz, January 3, 1994

streampos: This is a substantial change from rev 7.

I think what Rev 7 is trying to say is more like

```
class streampos {
    union { fpos_t fp; long n; };
    friend class filebuf ; // so it can get at fp
  public:
     streampos(long i) { n = i; }
     operator long() { return n; }
};
```

The draft uses `streamoff` where I have `long`. I don't think there is a guarantee that `sizeof(streamoff)` is at least `sizeof(long)` so there is a problem. (E.g. `stringbuf` stores `size_t`'s in `streampos`'s)

1 In this subclause, the type name *fpos_t* is a synonym for the type `fpos_t` defined in `<cstdio>` (17.2).

```
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
};
```

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*.[94] 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.

17.4.2.2.1 `streampos::streampos(streamoff)` [lib.cons.streampos]

Box 101
Library WG issue: Jerry Schwarz, September 28, 1993

`streampos::streampos` talks about conversion states for multibyte.

```
streampos(streamoff off = 0);
```

1 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.[95] The constructor then evaluates the expression `*this += off`.

17.4.2.2.2 `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, the function returns `(streamoff)(-1)`. 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.

17.4.2.2.3 `streampos::operator-(streampos&)` [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, the function returns `(streamoff)(-1)`. For a sequence that does not represent stream positions in uniform units, even a representable value need not be meaningful.

17.4.2.2.4 `streampos::operator+=(streamoff)` [lib.streampos::op+=]

[94] 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.
[95] The next character to read or write is the first character in the sequence.

Box 102

Library WG issue: Jerry Schwarz, January 3, 1994

At any rate, the wording needs to be clarified. E.g.

```
streampos& streampos::operator+=(streampos& rhs)
```

Adds off to the stream offset stored in `pos` and `fp`, if possible, and replaces the stored value. Otherwise ...

The problem is that this wording seems to say that if you can't add the offset to fp you take the otherwise.

```
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*. For a sequence that does not represent stream positions in uniform units, the resulting stream position need not be meaningful. The function returns `*this`.

17.4.2.2.5 streampos::operator-=(streamoff) [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*. For a sequence that does not represent stream positions in uniform units, the resulting stream position need not be meaningful. The function returns `*this`.

17.4.2.2.6 streampos::operator+(streamoff) [lib.streampos::op+]

```
streampos operator+(streamoff off) const;
```

1 Returns `streampos(*this) += off`.

17.4.2.2.7 streampos::operator-(streamoff) [lib.streampos::op-.off]

```
streampos operator-(streamoff off) const;
```

1 Returns `streampos(*this) -= off`.

17.4.2.2.8 streampos::operator==(const streampos&) [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 a nonzero value if the two correspond to the same position within a file or if both store an invalid stream position.

17.4.2.2.9 streampos::operator!=(const streampos&) [lib.op!=.streampos]

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

1 Returns a nonzero value if `!(*this == rhs)`.

17.4.2.3 Class streambuf [lib.streambuf]

```
class streambuf {
public:
        virtual ~streambuf();
        streampos pubseekoff(streamoff off, ios::seekdir way,
                ios::openmode which = ios::in | ios::out);
        streampos pubseekpos(streampos sp,
                ios::openmode which = ios::in | ios::out);
        streambuf* pubsetbuf(char* s, int n);
        int in_avail();
        int pubsync();
        int sbumpc();
        int sgetc();
        int sgetn(char* s, int n);
        int snextc();
        int sputbackc(char c);
        int sungetc();
        int sputc(int c);
        int sputn(const char* s, int n);
protected:
        streambuf();
        char* eback() const;
        char* gptr() const;
        char* egptr() const;
        void gbump(int n);
        void setg(char* gbeg_arg, char* gnext_arg, char* gend_arg);
        char* pbase() const;
        char* pptr() const;
        char* epptr() const;
        void pbump(int n);
        void setp(char* pbeg_arg, char* pend_arg);
        virtual int overflow(int c = EOF);
        virtual int pbackfail(int c = EOF);
        virtual int showmany();
        virtual int underflow();
        virtual int uflow();
        virtual int xsgetn(char* s, int n);
        virtual int xsputn(const char* s, int n);
        virtual streampos seekoff(streamoff off, ios::seekdir way,
                ios::openmode which = ios::in | ios::out);
        virtual streampos seekpos(streampos sp,
                ios::openmode which = ios::in | ios::out);
        virtual streambuf* setbuf(char* s, int n);
        virtual int sync();
private:
//      char* gbeg;     exposition only
//      char* gnext;    exposition only
//      char* gend;     exposition only
//      char* pbeg;     exposition only
//      char* pnext;    exposition only
//      char* pend;     exposition only
};
```

1 The class `streambuf` serves as an abstract base class for deriving various *stream buffers* whose objects each control two character sequences:

— a (single-byte) character input sequence;

— a (single-byte) character output sequence.

2 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.

3 Each sequence is characterized by three pointers which, if non-null, all point into the same 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).

4 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 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).

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

— `char*` *gbeg*, the beginning pointer for the input sequence;

— `char*` *gnext*, the next pointer for the input sequence;

— `char*` *gend*, the end pointer for the input sequence;

— `char*` *pbeg*, the beginning pointer for the output sequence;

— `char*` *pnext*, the next pointer for the output sequence;

— `char*` *pend*, the end pointer for the output sequence.

17.4.2.3.1 `streambuf::~streambuf()` [lib.des.streambuf]

```
virtual ~streambuf();
```

1 Destroys an object of class `streambuf`.

17.4.2.3.2 `streambuf::pubseekoff(streamoff, ios::seekdir, ios::openmode)` [lib.streambuf::pubseekoff]

```
streampos pubseekoff(streamoff off, ios::seekdir way,
        ios::openmode which = ios::in | ios::out);
```

1 Returns `seekoff(`*off*`,` *way*`,` *which*`)`. *

17.4.2.3.3 `streambuf::pubseekpos(streampos, ios::openmode)` [lib.streambuf::pubseekpos]

```
streampos pubseekpos(streampos sp,
        ios::openmode which = ios::in | ios::out);
```

1 Returns `seekpos(`*sp*`,` *which*`)`. *

17.4.2.3.4 `streambuf::pubsetbuf(char*, int)` [lib.streambuf::pubsetbuf]

```
streambuf* pubsetbuf(char* s, int n);
```

1 Returns `setbuf(`*s*`,` *n*`)`. |

17.4.2.3.5 `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, the function returns *gend* `-` *gnext*. |

17.4.2.3.6 `streambuf::pubsync()` [lib.streambuf::pubsync]

```
int pubsync();
```

1 Returns `sync()`.

17.4.2.3.7 `streambuf::sbumpc()` [lib.streambuf::sbumpc]

```
int sbumpc();
```

1 If the input sequence does not have a read position available, returns `uflow()`. Otherwise, the function returns `(unsigned char)*`*gnext*`++`.

17.4.2.3.8 `streambuf::sgetc()` [lib.streambuf::sgetc]

```
int sgetc();
```

1 If the input sequence does not have a read position available, returns `underflow()`. Otherwise, the function returns `(unsigned char)*`*gnext*.

17.4.2.3.9 `streambuf::sgetn(char*, int)` [lib.streambuf::sgetn]

```
int sgetn(char* s, int n);
```

1 Returns `xsgetn(s, n)`.

17.4.2.3.10 `streambuf::snextc()` [lib.streambuf::snextc]

```
int snextc();
```

1 Calls `sbumpc()` and, if that function returns `EOF`, returns `EOF`. Otherwise, the function returns `sgetc()`.

17.4.2.3.11 `streambuf::sputbackc(char)` [lib.streambuf::sputbackc]

```
int sputbackc(char c);
```

1 If the input sequence does not have a putback position available, or if `c != gnext[-1]`, returns `pbackfail(c)`. Otherwise, the function returns `(unsigned char)*--gnext`.

17.4.2.3.12 `streambuf::sungetc()` [lib.streambuf::sungetc]

```
int sungetc();
```

1 If the input sequence does not have a putback position available, returns `pbackfail()`. Otherwise, the function returns `(unsigned char)*--gnext`.

17.4.2.3.13 `streambuf::sputc(int)` [lib.streambuf::sputc]

```
int sputc(int c);
```

1 If the output sequence does not have a write position available, returns `overflow(c)`. Otherwise, the function returns `(unsigned char)(*pnext++ = c)`.

17.4.2.3.14 `streambuf::sputn(const char*, int)` [lib.streambuf::sputn]

```
int sputn(const char* s, int n);
```

1 Returns `xsputn(s, n)`.

17.4.2.3.15 `streambuf::streambuf()` [lib.cons.streambuf]

Box 103
Library WG issue: Jerry Schwarz, September 28, 1993

`streambuf` copy constructor explicitly undefined.

Also `operator=()`.

```
streambuf();
```

1 Constructs an object of class `streambuf()` and initializes all its pointer member objects to null pointers.[96)]

96) The default constructor is protected for class `streambuf` to assure that only objects for classes derived from this class may be constructed.

17.4.2.3.16 `streambuf::eback()` [lib.streambuf::eback]

```
char* eback() const;
```

1 Returns *gbeg*.

17.4.2.3.17 `streambuf::gptr()` [lib.streambuf::gptr]

```
char* gptr() const;
```

1 Returns *gnext*.

17.4.2.3.18 `streambuf::egptr()` [lib.streambuf::egptr]

```
char* egptr() const;
```

1 Returns *gend*.

17.4.2.3.19 `streambuf::gbump(int)` [lib.streambuf::gbump]

```
void gbump(int n);
```

1 Assigns *gnext + n* to *gnext*.

17.4.2.3.20 `streambuf::setg(char*, char*, char*)` [lib.streambuf::setg]

```
void setg(char* gbeg_arg, char* gnext_arg, char* gend_arg);
```

1 Assigns *gbeg_arg* to *gbeg*, *gnext_arg* to *gnext*, and *gend_arg* to *gend*.

17.4.2.3.21 `streambuf::pbase()` [lib.streambuf::pbase]

```
char* pbase() const;
```

1 Returns *pbeg*.

17.4.2.3.22 `streambuf::pptr()` [lib.streambuf::pptr]

```
char* pptr() const;
```

1 Returns *pnext*.

17.4.2.3.23 `streambuf::epptr()` [lib.streambuf::epptr]

```
char* epptr() const;
```

1 Returns *pend*.

17.4.2.3.24 `streambuf::pbump(int)` [lib.streambuf::pbump]

```
void pbump(int n);
```

1 Assigns *pnext + n* to *pnext*.

17.4.2.3.25 `streambuf::setp(char*, char*)` [lib.streambuf::setp]

```
void setp(char* pbeg_arg, char* pend_arg);
```

1 Assigns *pbeg_arg* to *pbeg*, *pbeg_arg* to *pnext*, and *pend_arg* to *pend*.

17.4.2.3.26 `streambuf::overflow(int)` [lib.streambuf::overflow]

Box 104

Library WG issue: Jerry Schwarz, January 3, 1994

In any event the protocol in the draft has some defects:

A) In case `c==EOF`, the draft doesn't allow the function to fail. My protocol does.

B) In the draft's first case, the protocol doesn't say anything about what happens when an output position is made available.

C) The draft's second case doesn't say anything about how `pbeg` and `pnext` are modified. Since it doesn't say they presumably must be left unchanged, but that is obviously a mistake.

D) Most importantly, I have indicated exactly what information must be supplied in order to specialize the protocol.

I want to emphasize (D). Even if Bill doesn't like my version of the protocol, I think it is essentially that there be some indication of what has to be specified to specialize it.

Box 105

Library WG issue: Jerry Schwarz, January 3, 1994

overflow: Rev 7 simply requires the return is not `EOF` if `c==EOF`. Requiring it to be 0 is a change.

More generally I think the San Diego rewrite over specifies the protocol in many places. Since this is the contract with user defined virtuals I think over specification here is wrong.

The only obligation of `overflow(c)` is to eventually append the characters between `pbeg` and `pptr` and `c` to the output sequence followed by `c`.

It is not (for example) required to return immediately if `c==EOF`.

Nor is it required to put `c` into the array even if it makes an output position available.

I think the San Diego rewrite over specified all the virtuals. I consider this a serious issue.

Box 106

Library WG issue: Jerry Schwarz, January 2, 1994

The San Diego rewrite has modified the description of `overflow`, but I think it still overspecifies in some ways, and under specifies in others. Also it doesn't make it clear that what is being described is a ''protocol'', that derived classes are required to implement. It hasn't been solicited, but here is my version of the `underflow` protocol (using the vocabulary of the draft).

The pending sequence of characters 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`.

`overflow` may consume some initial subsequence of the pending sequence. Consuming a character means either appending it to the associated output stream or discarding it.

In case some characters of the pending sequence have not been appended to the associated output stream, let `r` be the number of characters in the pending sequence not appended to the output stream. Then `pbeg` and `pnext` must be set so that `pnext-pbeg==r` and the `r` characters starting at `pbeg` are the same as the subsequence that has not been appended to the associated output stream.

In case all characters of the pending sequence have been appended to the associated output stream, then either `pbeg` is set to `NULL`, or `pbeg` and `pnext` are both set to (the same) non-`NULL` value.

The function may fail if either appending some character to the associated output stream fails or for some reason [I have in mind out of memory] it is unable to establish `pbeg` and `pnext` according to the above rules.

If the function fails it may signal that by returning `EOF` or throwing an exception.

Otherwise the function returns some value (other than `EOF`) to indicate success

To specialize this proposal you must specify.

a) What possible subsequences will be disposed of.
b) When are characters discarded and when are they
appended to the associated output stream.
c) The associated output stream. (This need not
be specified if
d) How failure is signaled.
e) The effect, if any on `gbeg, gnext, gend`

I believe this protocol is easier to work with than the one in the draft.

```
virtual int overflow(int c = EOF);
```
*

1 Appends the character designated by *c* to the output sequence, if possible, in one of three ways:

— If `c != EOF` and if either the output sequence has a write position available or the function makes a write position available, the function assigns *c* to `*pnext++`. The function signals success by returning `(unsigned char)c`.

— If `c != EOF` and if the function can append a character directly to the associated output sequence, the function appends *c* directly to the associated output sequence. If `pbeg < pnext`, the `pnext - pbeg` characters beginning at *pbeg* shall be first appended directly to the associated output sequence, beginning with the character at *pbeg*. The function signals success by returning `(unsigned char)c`.

— If `c == EOF`, there is no character to append. The function signals success by returning a value other than `EOF`.

2 If the function can succeed in more than one of these ways, it is unspecified which way is chosen. The function can alter the number of write positions available as a result of any call. How (or whether) the function makes a write position available or appends a character directly to the output sequence is defined separately for each class derived from `streambuf` in this clause.

3 The function returns `EOF` to indicate failure.

4 The default behavior is to return `EOF`.

17.4.2.3.27 `streambuf::pbackfail(int)` [lib.streambuf::pbackfail]

```
virtual int pbackfail(int c = EOF);
```

1 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 `(unsigned char)c == (unsigned char)gnext[-1]`, the function assigns `gnext - 1` to *gnext*. The function signals success by returning `(unsigned char)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, the function assigns *c* to `*--gnext`. The function signals success by returning `(unsigned char)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, the function puts back *c* directly to the associate input sequence. The function signals success by returning `(unsigned char)c`.

— If `c == EOF` and if either the input sequence has a putback position available or the function makes a putback position available, the function assigns `gnext - 1` to *gnext*. The function signals success by returning a value other than `EOF`.

— 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, the function puts back *x* directly to the associated input sequence. The function signals success by returning a value other than `EOF`.

2 If the function can succeed in more than one of these ways, it is unspecified which way is chosen. The function can alter the number of putback positions available as a result of any call. 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 `streambuf` in this clause.

3 The function returns `EOF` to indicate failure.

4 The default behavior is to return `EOF`.

17.4.2.3.28 `streambuf::showmany()` [lib.streambuf::showmany]

```
virtual int showmany();
```

1 Eeturns a count of the minimum number of characters that can be read from the input sequence before a call to `uflow()` or `underflow()` returns `EOF`. A return value of <196>1 indicates that the next such call will return `EOF`.

2 The default behavior is to return zero.

17.4.2.3.29 `streambuf::underflow()` [lib.streambuf::underflow]

Box 107

Library WG issue: Jerry Schwarz, January 3, 1994

Footnote 43: ‘‘The public streambuf member functions call `underflow` only if the increment `gnext` before returning’’

Must be raised to the body of the text.

And it has to be reworded because `underflow` can now return with `gnext` not being set.

Box 108

Library WG issue: Jerry Schwarz, January 3, 1994

`underflow`: The over specification here is really bad. I've written streambuf classes where underflow always guarantees some minimum amount of characters will be put in the buffer. Thus it may do lots of stuff even if there is a read position available.

My version of `underflow`:

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.

If the pending sequence is null then the function fails.

Otherwise the first character of the pending sequence is called the result character.

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.

The function sets up the `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.

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 a `gnext-n` agree with the backup sequence (where `n` is the length of the backup sequence)

> **Box 109**
>
> **Library WG issue:** Jerry Schwarz, January 2, 1994
>
> To specialize this protocol you must specify
>
> a) How a character is read from the input stream.
>
> b) How many characters are read from the input stream under various conditions
>
> d) Which alternative for case (b) of the rules for setting up `gnext` and `gend` are
>
> c) Whether the normal backup condition is satisfied.
>
> d) The effect on `pbeg,pnext,pend` if any

```
virtual int 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 `(unsigned char)*`*`gnext`*.

— Otherwise, if the function can determine the character *x* at the current position in the associated input sequence, it signals success by returning `(unsigned char)`*x*. If the function makes a read position available, it also assigns *x* to `*`*`gnext`*.

2 The function can alter the number of read positions available as a result of any call. How (or whether) the function makes a read position available or determines the character *x* at the current position in the associated input sequence is defined separately for each class derived from `streambuf` in this clause.

3 The function returns `EOF` to indicate failure.

4 The default behavior is to return `EOF`.

17.4.2.3.30 `streambuf::uflow()` [lib.streambuf::uflow]

> **Box 110**
>
> **Library WG issue:** Jerry Schwarz, January 3, 1994
>
> `streambuf::uflow` is supposed to be defined as
>
> Call `underflow(EOF)`. If `underflow` returns `EOF`, return `EOF`. If there is a read position available then do `gbump(-1)` and return `(unsigned char)*gnext++`

```
virtual int uflow();
```

1 Reads a character from the input sequence, if possible, and moves the stream position past it, as follows:

— If the input sequence has a read position available the function signals success by returning `(unsigned char)*`*`gnext`*`++`.

— Otherwise, if the function can read the character *x* directly from the associated input sequence, it signals success by returning `(unsigned char)`*x*. If the function makes a read position available, it also assigns *x* to `*`*`gnext`*.

2 The function can alter the number of read positions available as a result of any call. How (or whether) the function makes a read position available or reads a character directly from the input sequence is defined separately for each class derived from `streambuf` in this clause.

3 The function returns `EOF` to indicate failure.

4 The default behavior is to call `underflow()` and, if that function returns `EOF` or fails to make a read position available, return `EOF`. Otherwise, the function signals success by returning `(unsigned char)*`*`gnext`*`++`.[97)]

17.4.2.3.31 `streambuf::xsgetn(char*, int)` [lib.streambuf::xsgetn]

```
virtual int xsgetn(char* s, int 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`. The function returns the number of characters assigned.[98)]

17.4.2.3.32 `streambuf::xsputn(const char*, int)` [lib.streambuf::xsputn]

```
virtual int xsputn(const char* s, int 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`. The function returns the number of characters written.

17.4.2.3.33 `streambuf::seekoff(streamoff, ios::seekdir, ios::openmode)` [lib.streambuf::seekoff]

```
virtual streampos seekoff(streamoff off, ios::seekdir way,
        ios::openmode which = ios::in | ios::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 `streambuf` in this clause. The default behavior is to return an object of class `streampos` that stores an invalid stream position.

17.4.2.3.34 `streambuf::seekpos(streampos, ios::openmode)` [lib.streambuf::seekpos]

```
virtual streampos seekpos(streampos sp,
        ios::openmode which = ios::in | ios::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 `streambuf` in this clause. The default behavior is to return an object of class `streampos` that stores an invalid stream position.

[97)] A class derived from `streambuf` can override the virtual member function `underflow()` with a function that returns a value other than `EOF` without making a read position available. In that event, `streambuf::uflow()` must also be overridden since the default behavior is inadequate.

[98)] Classes derived from `streambuf` can provide more efficient ways to implement `xsgetn` and `xsputn` by overriding these definitions in the base class.

17.4.2.3.35 `streambuf::setbuf(char*, int)` [lib.streambuf::setbuf]

```
virtual streambuf* setbuf(char* s, int n);
```

1 Performs an operation that is defined separately for each class derived from `streambuf` in this clause.

2 The default behavior is to return `this`.

17.4.2.3.36 `streambuf::sync()` [lib.streambuf::sync]

```
virtual int sync();
```

1 Synchronizes the controlled sequences with any associated external sources and sinks of characters in a way that is defined separately for each class derived from `streambuf` in this clause. The function returns `EOF` if it fails. The default behavior is to return zero.

17.4.3 Header `<istream>` [lib.header.istream]

1 The header `<istream>` defines a type and a function signature that control input from a stream buffer.

17.4.3.1 Class `istream` [lib.istream]

Box 111

Library WG issue: Per Bothner, March 8, 1994

The members of class `istream` should not be allowed to call `sputback()`.

Box 112

Library WG issue: Jerry Schwarz, January 3, 1994

Rev 7 defined a bunch of terms like ''extracting a character.'' I can't find the equivalent here. In specifying members of istream, the San Diego rewrite uses phrases like ''characters are read .. until end-of-file'' without ever defining them (at least as far as I can find.) In particular Rev 7's definitions specified what happens when a virtual throws an exception, and I can't find that in the San Diego rewrite.

This is still not fixed. As far as I can determine, the draft doesn't say what happens when a virtual throws an exception.

Box 113

Library WG issue: Jerry Schwarz, January 2, 1994

Rev 7 also contained an explicit statement that except where explicitly noted none of the istream members call `pbackfail, seekoff,` or `seekpos`. This is an important constraint.

The draft now says ''All input characters are obtained or extracted by calls to the function signatures `sb.sbumpc(), sb.sgetc(), sputbackc()`''.

Perhaps that sentence is intended to address this issue, but it doesn't. Note that what is important is the virtuals that might be called, not the non-virtuals. And note that Rev 7 explicitly prohibit pbackfail from being called. That was deliberate.

```
class istream : virtual public ios {
public:
        istream(streambuf* sb);
        virtual ~istream();
        bool ipfx(bool noskipws = 0);
        void isfx();
        istream& operator>>(istream& (*pf)(istream&))
        istream& operator>>(ios& (*pf)(ios&))
        istream& operator>>(char* s);
        istream& operator>>(unsigned char* s)
        istream& operator>>(signed char* s);
        istream& operator>>(char& c);
        istream& operator>>(unsigned char& c)
        istream& operator>>(signed char& c)
        istream& operator>>(bool& n);
        istream& operator>>(short& n);
        istream& operator>>(unsigned short& n);
        istream& operator>>(int& n);
        istream& operator>>(unsigned int& n);
        istream& operator>>(long& n);
        istream& operator>>(unsigned long& n);
        istream& operator>>(float& f);
        istream& operator>>(double& f);
        istream& operator>>(long double& f);
        istream& operator>>(void*& p);
        istream& operator>>(streambuf& sb);
        int get();
        istream& get(char* s, int n, char delim = '\n');
        istream& get(unsigned char* s, int n, char delim = '\n')
        istream& get(signed char* s, int n, char delim = '\n')
        istream& get(char& c);
        istream& get(unsigned char& c);
        istream& get(signed char& c);
        istream& get(streambuf& sb, char delim = '\n');
        istream& getline(char* s, int n, char delim = '\n');
        istream& getline(unsigned char* s, int n, char delim = '\n')
        istream& getline(signed char* s, int n, char delim = '\n')
        istream& ignore(int n = 1, int delim = EOF);
        istream& read(char* s, int n);
        istream& read(unsigned char* s, int n)
        istream& read(signed char* s, int n)
        int readsome(char* s, int n);
        int peek();
        istream& putback(char c);
        istream& unget();
        int gcount() const;
        int sync();
private:
//      int chcount;      exposition only
};
```

1 The class `istream` defines a number of member function signatures that assist in reading and interpreting input from sequences controlled by a stream buffer.

2 Two groups of member function signatures share common properties: the *formatted input functions* (or *extractors*) and the *unformatted input functions.* Both groups of input functions obtain (or *extract*) input characters by calling the function signatures *sb*`.sbumpc()`, *sb*`.sgetc()`, and *sb*`.sputbackc(char)`. If one of these called functions throws an exception, the input function calls `setstate(badbit)` and rethrows the exception.

— The formatted input functions are:

```
istream& operator>>(char* s);
istream& operator>>(unsigned char* s)
istream& operator>>(signed char* s);
istream& operator>>(char& c);
istream& operator>>(unsigned char& c)
istream& operator>>(signed char& c)
istream& operator>>(bool& n);
istream& operator>>(short& n);
istream& operator>>(unsigned short& n);
istream& operator>>(int& n);
istream& operator>>(unsigned int& n);
istream& operator>>(long& n);
istream& operator>>(unsigned long& n);
istream& operator>>(float& f);
istream& operator>>(double& f);
istream& operator>>(long double& f);
istream& operator>>(void*& p);
istream& operator>>(streambuf& sb);
```

— The unformatted input functions are:

```
int get();
istream& get(char* s, int n, char delim = '\n');
istream& get(unsigned char* s, int n, char delim = '\n')
istream& get(signed char* s, int n, char delim = '\n')
istream& get(char& c);
istream& get(unsigned char& c);
istream& get(signed char& c);
istream& get(streambuf& sb, char delim = '\n');
istream& getline(char* s, int n, char delim = '\n');
istream& getline(unsigned char* s, int n, char delim = '\n')
istream& getline(signed char* s, int n, char delim = '\n')
istream& ignore(int n = 1, int delim = EOF);
istream& read(char* s, int n);
istream& read(unsigned char* s, int n)
istream& read(signed char* s, int n)
int readsome(char* s, int n);
int peek();
istream& putback(char c);
istream& unget();
```

3 Each formatted input function begins execution by calling `ipfx()`. If that function returns nonzero, the function endeavors to obtain the requested input. In any case, the formatted input function ends by calling `isfx()`, then returning the value specified for the formatted input function.

4 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. The behavior of such functions is described in terms of the conversion specification for an equivalent call to the function signature `fscanf(FILE*, const char*, ...)`, declared in `<cstdio>` (17.2), 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.[99)]

— 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.

99) The stream buffer can, of course, be associated with an input file, but it need not be.

— If the converted data value cannot be represented as a value of the specified scalar data type, a scan failure occurs.

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 follows:

— If `(flags() & basefield) == oct`, the conversion specifier is `o`.

— If `(flags() & basefield) == hex`, the conversion specifier is `x`.

— If `(flags() & basefield) == 0`, the conversion specifier is `i`.

7 Otherwise, the integral conversion specifier is `d` for conversion to a signed integral type, or `u` for conversion to an unsigned integral type.

8 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.

9 For the sake of exposition, the data maintained by an object of class `istream` is presented here as:

— `int` *`chcount`*, stores the number of characters extracted by the last unformatted input member function called for the object.

17.4.3.1.1 `istream::istream()` [lib.cons.istream]

```
istream(streambuf* sb);
```

1 Constructs an object of class `istream`, assigning initial values to the base class by calling `ios::init(`*`sb`*`)`, then assigning zero to *`chcount`*.

17.4.3.1.2 `istream::~istream()` [lib.des.istream]

```
virtual ~istream();
```

1 Destroys an object of class `istream`.

17.4.3.1.3 `istream::ipfx(bool)` [lib.istream::ipfx]

```
bool ipfx(bool noskipws = 0);
```

1 If `good()` is nonzero, 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. (The call `tie()->flush()` does not necessarily occur if the function can determine that no synchronization is necessary.) If *`noskipws`* is zero and `flags() & skipws` is nonzero, the function extracts and discards each character as long as `isspace(`*`c`*`)` is nonzero for the next available input character *c*. The function signature `isspace(int)` is declared in `<cctype>` (17.2).

2 If, after any preparation is completed, `good()` is nonzero, the function returns a nonzero value. Otherwise, it calls `setstate(failbit)` and returns zero.[100)]

[100)] The function signatures `ipfx(int)` and `isfx()` can also perform additional implementation-dependent operations.

17.4.3.1.4 istream::isfx() [lib.istream::isfx]

```
void isfx();
```

1 Returns.

17.4.3.1.5 istream::operator>>(istream& (*)(istream&)) [lib.istream::ext.imanip]

```
istream& operator>>(istream& (*pf)(istream&))
```

1 Returns (*pf)(*this).[101]

17.4.3.1.6 istream::operator>>(ios& (*)(ios&)) [lib.istream::ext.iomanip]

```
istream& operator>>(ios& (*pf)(ios&))
```

1 Calls (*(ios*)pf)(*this), then returns *this.[102]

17.4.3.1.7 istream::operator>>(char*) [lib.istream::ext.str]

```
istream& operator>>(char* s);
```

1 A formatted input function, 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, defined in <climits> (17.2).

2 Characters are extracted and stored until any of the following occurs:

— *n* - 1 characters are stored;

— end-of-file occurs on the input sequence;

— isspace(*c*) is nonzero for the next available input character *c*.

3 The function signature isspace(int) is declared in <cctype> (17.2).

4 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). The function returns *this.

17.4.3.1.8 istream::operator>>(unsigned char*) [lib.istream::ext.ustr]

```
istream& operator>>(unsigned char* s)
```

1 Returns operator>>((char*)*s*).

17.4.3.1.9 istream::operator>>(signed char*) [lib.istream::ext.sstr]

```
istream& operator>>(signed char* s);
```

1 Returns operator>>((char*)*s*).

17.4.3.1.10 istream::operator>>(char&) [lib.istream::ext.c]

```
istream& operator>>(char& c);
```

101) See, for example, the function signature ws(istream&).
102) See, for example, the function signature dec(ios&).

1 A formatted input function, extracts a character, if one is available, and stores it in *c*. Otherwise, the function calls `setstate(failbit)`. The function returns `*this`.

17.4.3.1.11 `istream::operator>>(unsigned char&)` [lib.istream::ext.uc]

```
istream& operator>>(unsigned char& c)
```

1 Returns `operator>>((char&)`*c*`)`.

17.4.3.1.12 `istream::operator>>(signed char&)` [lib.istream::ext.sc]

```
istream& operator>>(signed char& c)
```

1 Returns `operator>>((char&)`*c*`)`.

17.4.3.1.13 `istream::operator>>(bool&)` [lib.istream::ext.bool]

```
istream& operator>>(bool& n);
```

1 A formatted input function, converts a signed short integer, if one is available, and stores it in *x*. If *x* has a value other than 0 or 1, a scan failure occurs. Otherwise, the function stores *x* in *n*. The function returns `*this`.

17.4.3.1.14 `istream::operator>>(short&)` [lib.istream::ext.si]

```
istream& operator>>(short& n);
```

1 A formatted input function, converts a signed short integer, if one is available, and stores it in *n*. The function returns `*this`.

17.4.3.1.15 `istream::operator>>(unsigned short&)` [lib.istream::ext.usi]

```
istream& operator>>(unsigned short& n);
```

1 A formatted input function, converts an unsigned short integer, if one is available, and stores it in *n*. The function returns `*this`.

17.4.3.1.16 `istream::operator>>(int&)` [lib.istream::ext.i]

```
istream& operator>>(int& n);
```

1 A formatted input function, converts a signed integer, if one is available, and stores it in *n*. The function returns `*this`.

17.4.3.1.17 `istream::operator>>(unsigned int&)` [lib.istream::ext.ui]

```
istream& operator>>(unsigned int& n);
```

1 A formatted input function, converts an unsigned integer, if one is available, and stores it in *n*. The function returns `*this`.

17.4.3.1.18 `istream::operator>>(long&)` [lib.istream::ext.li]

```
istream& operator>>(long& n);
```

1 A formatted input function, converts a signed long integer, if one is available, and stores it in *n*. The function returns `*this`.

17.4.3.1.19 `istream::operator>>(unsigned long&)` [lib.istream::ext.uli]

```
istream& operator>>(unsigned long& n);
```

1 A formatted input function, converts an unsigned long integer, if one is available, and stores it in *n*. The function returns `*this`.

17.4.3.1.20 `istream::operator>>(float&)` [lib.istream::ext.f]

```
istream& operator>>(float& f);
```

1 A formatted input function, converts a `float`, if one is available, and stores it in *f*. The function returns `*this`.

17.4.3.1.21 `istream::operator>>(double&)` [lib.istream::ext.d]

```
istream& operator>>(double& f);
```

1 A formatted input function, converts a `double`, if one is available, and stores it in *f*. The function returns `*this`.

17.4.3.1.22 `istream::operator>>(long double&)` [lib.istream::ext.ld]

```
istream& operator>>(long double& f);
```

1 A formatted input function, converts a `long double`, if one is available, and stores it in *f*. The function returns `*this`.

17.4.3.1.23 `istream::operator>>(void*&)` [lib.istream::ext.ptr]

```
istream& operator>>(void*& p);
```

1 A formatted input function, converts a pointer to `void`, if one is available, and stores it in *p*. The function returns `*this`.

17.4.3.1.24 `istream::operator>>(streambuf&)` [lib.istream::ext.sb]

```
istream& operator>>(streambuf& sb);
```

1 A formatted input function, 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 but not rethrown).

2 If the function inserts no characters, it calls `setstate(failbit)`. The function returns `*this`.

17.4.3.1.25 `istream::get()` [lib.istream::get]

Box 114

Library WG issue: Greg Bentz, October 22, 1993

I have been consulting the C++ library draft (X3J16/93-108,WG21/NO315) and I think I have found a statement which is inconsistent with most existing implementations. While that doesn't say much, it also seems to go against what I feel is the desired behaviour.

The functions:
`istream::get( char *, int, char )` (was 17.4.1.8.27)
`istream::getline( char *, int, char )` (was 17.4.1.8.34)

both declare the following:

‘‘If the function stores no characters, it calls ’setstate(failbit)’.’’

I believe the line should read:

‘‘If the function stores no characters and ’c != delim’, it calls ’setstate(failbit)’.’’

This change, particularly for ’istream::getline(char *, int, char)’, allows line oriented reading of input files that have ’delim’ terminated lines, some of which may be empty.

If the call ’getline(buf, sizeof(buf), ’0);’ is made when the next character in the input stream is ’0 the current wording causes ’failbit’ to be set. The proposed wording allows ’getline’ to return with no characters in ’buf’, but having consumed the ’0 character.

In support of this proposal I also refer to the "C++ IOStreams Handbook" by Steve Teale (ISBN 0-201-59641-5) pages 288-290. (example source t6.cpp) Mr. Teale indicates that the proposed wording is, in his opinion, the correct behaviour.

```
int get();
```

1 An unformatted input function, 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`.

17.4.3.1.26 `istream::get(char*, int, char)` [lib.istream::get.str]

```
istream& get(char*  s, int n, char delim = '\n');
```

1 An unformatted input function, 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).

2 If the function stores no characters, it calls `setstate(failbit)`. In any case, it then stores a null character into the next successive location of the array. The function returns `*this`.

17.4.3.1.27 `istream::get(unsigned char*, int, char)` [lib.istream::get.ustr]

```
istream& get(unsigned char* s, int n, char delim = '\n')
```

1 Returns `get((char*)s, n, delim)`.

17.4.3.1.28 `istream::get(signed char*, int, char)` [lib.istream::get.sstr]

```
istream& get(signed char* s, int n, char delim = '\n')
```

1 Returns `get((char*)s, n, delim)`.

17.4.3.1.29 `istream::get(char&)` [lib.istream::get.c]

```
istream& get(char& c);
```

1 An unformatted input function, extracts a character, if one is available, and assigns it to *c*. Otherwise, the function calls `setstate(failbit)`. The function returns `*this`.

17.4.3.1.30 `istream::get(unsigned char&)` [lib.istream::get.uc]

```
istream& get(unsigned char& c);
```

1 Returns `get((char&)c)`.

17.4.3.1.31 `istream::get(signed char&)` [lib.istream::get.sc]

```
istream& get(signed char& c);
```

1 Returns `istream::get((char&)c)`.

17.4.3.1.32 `istream::get(streambuf&, char)` [lib.istream::get.sb]

```
istream& get(streambuf& sb, char delim = '\n');
```

1 An unformatted input function, 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).

2 If the function inserts no characters, it calls `setstate(failbit)`. The function returns `*this`.

17.4.3.1.33 `istream::getline(char*, int, char)` [lib.istream::getline.str]

```
istream& getline(char* s, int n, char delim = '\n');
```

1 An unformatted input function, 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 (in which case the function calls `setstate(failbit)`);

— 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 the input character is extracted but not stored).

2 If the function stores no characters, it calls `setstate(failbit)`. In any case, it then stores a null character into the next successive location of the array. The function returns `*this`.

17.4.3.1.34 `istream::getline(unsigned char*, int, char)` [lib.istream::getline.ustr]

```
istream& getline(unsigned char* s, int n, char delim = '\n')
```

1 Returns `getline((char*)s, n, delim)`.

17.4.3.1.35 `istream::getline(signed char*, int, char)` [lib.istream::getline.sstr]

```
istream& getline(signed char* s, int n, char delim = '\n')
```

1 Returns `getline((char*)s, n, delim)`.

17.4.3.1.36 `istream::ignore(int, int)` [lib.istream::ignore]

```
istream& ignore(int n = 1, int delim = EOF);
```

1 An unformatted input function, 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).

2 The last condition will never occur if *delim* `== EOF`.

3 The macro `INT_MAX` is defined in `<climits>`.

4 The function returns `*this`.

17.4.3.1.37 `istream::read(char*, int)` [lib.istream::read.str]

```
istream& read(char* s, int n);
```

1 An unformatted input function, 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 The function returns `*this`.

17.4.3.1.38 `istream::read(unsigned char*, int)` [lib.istream::read.ustr]

```
istream& read(unsigned char* s, int n)
```

1 Returns `read((char*)s, n)`.

17.4.3.1.39 `istream::read(signed char*, int)` [lib.istream::read.sstr]

```
istream& read(signed char* s, int n)
```

1 Returns `read((char*)s, n)`.

17.4.3.1.40 `istream::readsome(char*, int)` [lib.istream::readsome]

```
int readsome(char* s, int n);
```

1 An unformatted input function, 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 <196>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)`.

17.4.3.1.41 `istream::peek()` [lib.istream::peek]

```
int peek();
```

1 An unformatted input function, returns the next available input character, if possible.

2 If `good()` is zero, the function returns `EOF`. Otherwise, it returns `rdbuf()->sgetc()`.

17.4.3.1.42 `istream::putback(char)` [lib.istream::putback]

```
istream& putback(char c);
```

1 An unformatted input function, calls `rdbuf->sputbackc(c)`. If that function returns `EOF`, the function calls `setstate(badbit)`. The function returns `*this`.

17.4.3.1.43 `istream::unget()` [lib.istream::unget]

```
istream& unget();
```

1 An unformatted input function, calls `rdbuf->sungetc()`. If that function returns `EOF`, the function calls `setstate(badbit)`. The function returns `*this`.

17.4.3.1.44 `istream::gcount()` [lib.istream::gcount]

```
int gcount() const;
```

1 Returns *chcount*.

17.4.3.1.45 `istream::sync()` [lib.istream::sync]

```
int sync();
```

1 If `rdbuf()` is a null pointer, returns `EOF`. Otherwise, the function calls `rdbuf()->pubsync()` and, if that function returns `EOF`, calls `setstate(badbit)` and returns `EOF`. Otherwise, the function returns zero.

17.4.3.2 `ws(istream&)` [lib.ws]

```
istream& ws(istream& is);
```

1 Saves a copy of *is.fmtflags*, then clears *is*.`skipws` in *is.fmtflags*. The function then calls *is*.`ipfx()` and *is*.`isfx()`, and restores *is.fmtflags* to its saved value. The function returns *is*.[103]

[103] The effect of `cin >> ws` is to skip any white space in the input sequence controlled by `cin`.

17.4.4 Header `<ostream>` [lib.header.ostream]

1 The header `<ostream>` defines a type and several function signatures that control output to a stream buffer.

17.4.4.1 Class `ostream` [lib.ostream]

Box 115

Library WG issue: Jerry Schwarz, January 3, 1994

Again the San Diego rewrite omits definitions. In particular it is silent on what happens when exceptions are thrown by virtuals.

Not fixed.

```
class ostream : virtual public ios {
public:
        ostream(streambuf* sb);
        virtual ~ostream();
        bool opfx();
        void osfx();
        ostream& operator<<(ostream& (*pf)(ostream&));
        ostream& operator<<(ios& (*pf)(ios&));
        ostream& operator<<(const char* s);
        ostream& operator<<(char c);
        ostream& operator<<(unsigned char c);
        ostream& operator<<(signed char c);
        ostream& operator<<(bool n);
        ostream& operator<<(short n);
        ostream& operator<<(unsigned short n);
        ostream& operator<<(int n);
        ostream& operator<<(unsigned int n);
        ostream& operator<<(long n);
        ostream& operator<<(unsigned long n);
        ostream& operator<<(float f);
        ostream& operator<<(double f);
        ostream& operator<<(long double f);
        ostream& operator<<(void* p);
        ostream& operator<<(streambuf& sb);
        int put(char c);
        ostream& write(const char* s, int n);
        ostream& write(const unsigned char* s, int n);
        ostream& write(const signed char* s, int n);
        ostream& flush();
};
```

1 The class `ostream` defines a number of member function signatures that assist in formatting and writing output to output sequences controlled by a stream buffer.

2 Two groups of member function signatures share common properties: the *formatted output functions* (or *inserters*) and the *unformatted output functions.* Both groups of output functions generate (or *insert*) output characters by calling the function signature *sb*`.sputc(int)`. If the called function throws an exception, the output function calls `setstate(badbit)` and rethrows the exception.

— The formatted output functions are:

```
ostream& operator<<(const char* s);
ostream& operator<<(char c);
ostream& operator<<(unsigned char c);
ostream& operator<<(signed char c);
ostream& operator<<(bool n);
ostream& operator<<(short n);
ostream& operator<<(unsigned short n);
ostream& operator<<(int n);
ostream& operator<<(unsigned int n);
ostream& operator<<(long n);
ostream& operator<<(unsigned long n);
ostream& operator<<(float f);
ostream& operator<<(double f);
ostream& operator<<(long double f);
ostream& operator<<(void* p);
ostream& operator<<(streambuf* sb);
```

— The unformatted output functions are:

```
ostream& put(char c);
ostream& write(const char* s, int n);
ostream& write(const unsigned char* s, int n);
ostream& write(const signed char* s, int n);
```

3 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.

4 Some formatted output functions endeavor to generate the requested output by converting a value from some scalar or NTBS type to text form and inserting the converted text in the output sequence. The behavior of such functions is described in terms of the conversion specification for an equivalent call to the function signature `fprintf(FILE*, const char*, ...)`, declared in `<cstdio>` (17.2), 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.[104)]

— The formatted output function uses the fill character returned by `fill()` as the padding character (rather than the space character for left or right padding, or `0` for internal padding).

5 If the operation fails for any reason, the formatted output function calls `setstate(badbit)`.

6 For conversion from an integral type other than a character type, the function determines the integral conversion specifier as follows:

— If `(flags() & basefield) == oct`, the integral conversion specifier is `o`.

— If `(flags() & basefield) == hex`, the integral conversion specifier is `x`. If `flags() & uppercase` is nonzero, `x` is replaced with `X`.

7 Otherwise, the integral conversion specifier is `d` for conversion from a signed integral type, or `u` for conversion from an unsigned integral type.

8 For conversion from a floating-point type, the function determines the floating-point conversion specifier as follows:

104) The stream buffer can, of course, be associated with an output file, but it need not be.

— If `(flags() & floatfield) == fixed`, the floating-point conversion specifier is `f`.

— If `(flags() & floatfield) == scientific`, the floating-point conversion specifier is `e`. If `flags() & uppercase` is nonzero, `e` is replaced with `E`.

9 Otherwise, the floating-point conversion specifier is `g`. If `flags() & uppercase` is nonzero, `g` is replaced with `G`.

10 The conversion specifier has the following additional qualifiers prepended to make a conversion specification:

— For conversion from an integral type other than a character type, if `flags() & showpos` is nonzero, the flag + is prepended to the conversion specification; and if `flags() & showbase` is nonzero, the flag # is prepended to the conversion specification.

— For conversion from a floating-point type, if `flags() & showpos` is nonzero, the flag + is prepended to the conversion specification; and if `flags() & showpoint` is nonzero, the flag # is prepended to the conversion specification.

— 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()`.

11 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.[105)]

12 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.

13 Unless explicitly stated otherwise for a particular inserter, each formatted output function calls `width(0)` after determining the field width. |

14 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. |

17.4.4.1.1 `ostream::ostream(streambuf*)` [lib.cons.ostream.sb]

```
ostream(streambuf* sb);
```

1 Constructs an object of class `ostream`, assigning initial values to the base class by calling `ios::init(sb)`. | *

105) The conversion specification `#o` generates a leading `0` which is *not* a padding character.

17.4.4.1.2 `ostream::~ostream()` [lib.des.ostream]

```
virtual ~ostream();
```

1 Destroys an object of class `ostream`.

17.4.4.1.3 `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, the function calls `tie()->flush()`. It returns `good()`.[106]

17.4.4.1.4 `ostream::osfx()` [lib.ostream::osfx]

```
void osfx();
```

1 If `flags() & unitbuf` is nonzero, calls `flush()`.

17.4.4.1.5 `ostream::operator<<(ostream& (*)(ostream&))` [lib.ostream::ins.omanip]

```
ostream& operator<<(ostream& (*pf)(ostream&))
```

1 Returns `(*pf)(*this)`.[107]

17.4.4.1.6 `ostream::operator<<(ios& (*)(ios&))` [lib.ostream::ins.iomanip]

```
ostream& operator<<(ios& (*pf)(ios&))
```

1 Calls `(*(ios*)pf)(*this)`, then returns `*this`.[108]

17.4.4.1.7 `ostream::operator<<(const char*)` [lib.ostream::ins.str]

```
ostream& operator<<(const char* s);
```

1 A formatted output function, converts the NTBS *s* with the conversion specifier `s`. The function returns `*this`.

17.4.4.1.8 `ostream::operator<<(char)` [lib.ostream::ins.c]

```
ostream& operator<<(char c);
```

1 A formatted output function, converts the `char` *c* with the conversion specifier `c` and a field width of zero. The stored field width (`ios::`*wide*) is *not* set to zero. The function returns `*this`.

17.4.4.1.9 `ostream::operator<<(unsigned char)` [lib.ostream::ins.uc]

```
ostream& operator<<(unsigned char c)
```

1 Returns `operator<<((char)`*c*`)`.

---

[106] The function signatures `opfx()` and `osfx()` can also perform additional implementation-dependent operations.
[107] See, for example, the function signature `endl(ostream&)`.
[108] See, for example, the function signature `::dec(ios&)`.

17.4.4.1.10 `ostream::operator<<(signed char)` [lib.ostream::ins.sc]

```
ostream& operator<<(signed char c)
```

1 Returns `operator<<((char)c)`.

17.4.4.1.11 `ostream::operator<<(bool)` [lib.ostream::ins.bool]

```
ostream& operator<<(bool n);
```

1 A formatted output function, converts the expression `n != 0` with the integral conversion specifier. The function returns `*this`.

17.4.4.1.12 `ostream::operator<<(short)` [lib.ostream::ins.si]

```
ostream& operator<<(short n);
```

1 A formatted output function, converts the signed short integer *n* with the integral conversion specifier preceded by `h`. The function returns `*this`.

17.4.4.1.13 `ostream::operator<<(unsigned short)` [lib.ostream::ins.usi]

```
ostream& operator<<(unsigned short n);
```

1 A formatted output function, converts the unsigned short integer *n* with the integral conversion specifier preceded by `h`. The function returns `*this`.

17.4.4.1.14 `ostream::operator<<(int)` [lib.ostream::ins.i]

```
ostream& operator<<(int n);
```

1 A formatted output function, converts the signed integer *n* with the integral conversion specifier. The function returns `*this`.

17.4.4.1.15 `ostream::operator<<(unsigned int)` [lib.ostream::ins.ui]

```
ostream& operator<<(unsigned int n);
```

1 A formatted output function, converts the unsigned integer *n* with the integral conversion specifier. The function returns `*this`.

17.4.4.1.16 `ostream::operator<<(long)` [lib.ostream::ins.li]

```
ostream& operator<<(long n);
```

1 A formatted output function, converts the signed long integer *n* with the integral conversion specifier preceded by `l`. The function returns `*this`.

17.4.4.1.17 `ostream::operator<<(unsigned long)` [lib.ostream::ins.uli]

```
ostream& operator<<(unsigned long n);
```

1 A formatted output function, converts the unsigned long integer *n* with the integral conversion specifier preceded by `l`. The function returns `*this`.

17.4.4.1.18 `ostream::operator<<(float)` [lib.ostream::ins.f]

```
ostream& operator<<(float f);
```

1 A formatted output function, converts the `float` *f* with the floating-point conversion specifier. The function returns `*this`.

17.4.4.1.19 `ostream::operator<<(double)` [lib.ostream::ins.d]

```
ostream& operator<<(double f);
```

1 A formatted output function, converts the `double` *f* with the floating-point conversion specifier. The function returns `*this`.

17.4.4.1.20 `ostream::operator<<(long double)` [lib.ostream::ins.ld]

```
ostream& operator<<(long double f);
```

1 A formatted output function, converts the `long double` *f* with the floating-point conversion specifier preceded by `L`. The function returns `*this`.

17.4.4.1.21 `ostream::operator<<(void*)` [lib.ostream::ins.ptr]

```
ostream& operator<<(void* p);
```

1 A formatted output function, converts the pointer to `void` *p* with the conversion specifier `p`. The function returns `*this`.

17.4.4.1.22 `ostream::operator<<(streambuf&)` [lib.ostream::ins.sb]

```
ostream& operator<<(streambuf& sb);
```

1 A formatted output function, extracts characters from the input sequence controlled by *sb* and inserts them in `*this`. 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 rethrown).[109)]

2 If the function inserts no characters, it calls `setstate(failbit)`. The function returns `*this`.

17.4.4.1.23 `ostream::put(char)` [lib.ostream::put]

```
int put(char c);
```

1 An unformatted output function, inserts the character *c*, if possible. The function then returns `(unsigned char)`*c*. Otherwise, the function calls `setstate(badbit)`. It then returns `EOF`.

17.4.4.1.24 `ostream::write(const char*, int)` [lib.ostream::write.str]

```
ostream& write(const char* s, int n);
```

[109)] This behavior differs from that for `istream::istream& operator>>(streambuf&)`, which does *not* rethrow the exception.

1 An unformatted output function, 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 The function returns `*this`.

17.4.4.1.25 `ostream::write(const unsigned char*, int)` [lib.ostream::write.ustr]

```
ostream& write(const unsigned char* s, int n)
```

1 Returns `write((const char*)s, n)`.

17.4.4.1.26 `ostream::write(const signed char*, int)` [lib.ostream::write.sstr]

```
ostream& write(const signed char* s, int n)
```

1 Returns `write((const char*)s, n)`.

17.4.4.1.27 `ostream::flush()` [lib.ostream::flush]

```
ostream& flush();
```

1 If `rdbuf()` is not a null pointer, calls `rdbuf()->pubsync()`. If that function returns `EOF`, the function calls `setstate(badbit)`.

2 The function returns `*this`.

17.4.4.2 `endl(ostream&)` [lib.endl]

```
ostream& endl(ostream& os);
```

1 Calls `os.put('\n')`, then `os.flush()`. The function returns *os*.[110]

17.4.4.3 `ends(ostream&)` [lib.ends]

```
ostream& ends(ostream& os);
```

1 Calls `os.put('\0')`. The function returns *os*.[111]

17.4.4.4 `flush(ostream&)` [lib.flush]

```
ostream& flush(ostream& os);
```

1 Calls `os.flush()`. The function returns *os*.

17.4.5 Header `<iomanip>` [lib.header.iomanip]

1 The header `<iomanip>` defines three template classes and several related functions that use these template classes to provide extractors and inserters that alter information maintained by class `ios` and its derived classes. It also defines several instantiations of these template classes and functions.

[110] 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.

[111] 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 `strstreambuf`, the null character can terminate an NTBS constructed in an array object.

17.4.5.1 Template class smanip<T> [lib.template.smanip]

```
template<class T> class smanip {
public:
        smanip(ios& (*pf_arg)(ios&, T), T);
//      ios& (*pf)(ios&, T);      exposition only
//      T manarg;           exposition only
};
```

1 The template class smanip<*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. For the sake of exposition, the maintained data is presented here as:

— ios& (**pf*)(ios&, *T*), the function pointer;

— *T manarg*, the object of type *T*.

17.4.5.1.1 smanip<T>::smanip(ios& (*)(ios&, T), T) [lib.cons.smanip.ios]

```
smanip(ios& (*pf_arg)(ios&, T), T manarg_arg);
```

1 Constructs an object of class smanip<*T*>, initializing *pf* to *pf_arg* and *manarg* to *manarg_arg*.

17.4.5.1.2 operator>>(istream&, const smanip<T>&) [lib.ext.smanip]

```
template <class T> istream& operator>>(istream& is, const smanip<T>& 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(ios::failbit) (the exception is not rethrown). The function returns *is*.

17.4.5.1.3 operator<<(ostream&, const smanip<T>&) [lib.ins.smanip]

```
template <class T> ostream& operator<<(ostream& os, const smanip<T>& 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(ios::failbit) (the exception is not rethrown). The function returns *os*.

17.4.5.2 Template class imanip<T> [lib.template.imanip]

```
template<class T> class imanip {
public:
        imanip(ios& (*pf_arg)(ios&, T), T);
//      ios& (*pf)(ios&, T);      exposition only
//      T manarg;           exposition only
};
```

1 The template class imanip<*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. For the sake of exposition, the maintained data is presented here as:

— ios& (**pf*)(ios&, *T*), the function pointer;

— *T manarg*, the object of type *T*.

17.4.5.2.1 imanip<T>::imanip(ios& (*)(ios&, T), T) [lib.cons.imanip.ios]

```
imanip<T>::imanip(ios& (*pf_arg)(ios&, T), T manarg_arg);
```

1 Constructs an object of class imanip<T>, initializing pf to *pf_arg* and manarg to *manarg_arg*.

17.4.5.2.2 operator>>(istream&, const imanip<T>&) [lib.ext.imanip]

```
template <class T> istream& operator>>(istream& is, const imanip<T>& 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(ios::failbit) (the exception is not rethrown). The function returns *is*.

17.4.5.3 Template class omanip<T> [lib.template.omanip]

```
template<class T> class omanip {
public:
        omanip(ios& (*pf_arg)(ios&, T), T);
//      ios& (*pf)(ios&, T);      exposition only
//      T manarg;           exposition only
};
```

1 The template class 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. For the sake of exposition, the maintained data is presented here as:

— ios& (*pf)(ios&, T), the function pointer;

— T manarg, the object of type *T*.

17.4.5.3.1 omanip<T>::omanip(ios& (*)(ios&, T), T) [lib.cons.omanip.ios]

```
omanip<T>::omanip(ios& (*pf_arg)(ios&, T), T manarg_arg);
```

1 Constructs an object of class omanip<T>, initializing pf to *pf_arg* and manarg to *manarg_arg*.

17.4.5.3.2 operator<<(istream&, const omanip<T>&) [lib.ins.omanip]

```
template <class T> ostream& operator<<(ostream& os, const omanip<T>& 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(ios::failbit) (the exception is not rethrown). The function returns *os*.

17.4.5.4 Instantiations of manipulators [lib.instantiations.of.manipulators]

17.4.5.4.1 resetiosflags(ios::fmtflags) [lib.resetiosflags]

```
smanip<ios::fmtflags> resetiosflags(ios::fmtflags mask);
```

1 Returns smanip<ios::fmtflags>(&f, mask), where *f* can be defined as: [112]

[112] The expression cin >> resetiosflags(ios::skipws) clears ios::skipws in the format flags stored in the istream object cin (the same as cin >> noskipws), and the expression cout << resetiosflags(ios::showbase) clears ios::showbase in the format flags stored in the ostream object cout (the same as cout << noshowbase).

```
ios& f(ios& str, ios::fmtflags mask)
{       // reset specified flags
        str.setf((ios::fmtflags)0, mask);
        return (str);
}
```

17.4.5.4.2 `setiosflags(ios::fmtflags)` [lib.setiosflags]

```
smanip<ios::fmtflags> setiosflags(ios::fmtflags mask);
```

1 Returns `smanip<ios::fmtflags>(&f, mask)`, where *f* can be defined as:

```
ios& f(ios& str, ios::fmtflags mask)
{       // set specified flags
        str.setf(mask);
        return (str);
}
```

17.4.5.4.3 `setbase(int)` [lib.setbase]

```
smanip<int> setbase(int base);
```

1 Returns `smanip<int>(&f, base)`, where *f* can be defined as:

```
ios& f(ios& str, int base)
{       // set basefield
        str.setf(n == 8 ? ios::oct : n == 10 ? ios::dec
                : n == 16 ? ios::hex : (ios::fmtflags)0, ios::basefield);
        return (str);
}
```

17.4.5.4.4 `setfill(int)` [lib.setfill]

```
smanip<int> setfill(int c);
```

1 Returns `smanip<int>(&f, c)`, where *f* can be defined as:

```
ios& f(ios& str, int c)
{       // set fill character
        str.fill(c);
        return (str);
}
```

17.4.5.4.5 `setprecision(int)` [lib.setprecision]

```
smanip<int> setprecision(int n);
```
**

1 Returns `smanip<int>(&f, n)`, where *f* can be defined as:

```
ios& f(ios& str, int n)
{       // set precision
        str.precision(n);
        return (str);
}
```

17.4.5.4.6 `setw(int)` [lib.setw]

```
smanip<int> setw(int n);
```

1 Returns `smanip<int>(&f, n)`, where *f* can be defined as:

```
ios& f(ios& str, int n)
{       // set width
        str.width(n);
        return (str);
}
```

17.4.6 Header `<strstream>` [lib.header.strstream]

1 The header `<strstream>` defines three types that associate stream buffers with (single-byte) character array objects and assist reading and writing such objects.

17.4.6.1 Class `strstreambuf` [lib.strstreambuf]

```
class strstreambuf : public streambuf {
public:
        strstreambuf(int alsize_arg = 0);
        strstreambuf(void* (*palloc_arg)(size_t),
                void (*pfree_arg)(void*));
        strstreambuf(char* gnext_arg, int n, char* pbeg_arg = 0);
        strstreambuf(unsigned char* gnext_arg, int n,
                unsigned char* pbeg_arg = 0);
        strstreambuf(signed char* gnext_arg, int n,
                signed char* pbeg_arg = 0);
        strstreambuf(const char* gnext_arg, int n);
        strstreambuf(const unsigned char* gnext_arg, int n);
        strstreambuf(const signed char* gnext_arg, int n);
        virtual ~strstreambuf();
        void freeze(bool = 1);
        char* str();
        int pcount();
protected:
//      virtual int overflow(int c = EOF);      inherited
//      virtual int pbackfail(int c = EOF);     inherited
//      virtual int showmany(); inherited
//      virtual int underflow();        inherited
//      virtual int uflow();    inherited
//      virtual int xsgetn(char* s, int n);     inherited
//      virtual int xsputn(const char* s, int n);       inherited
//      virtual streampos seekoff(streamoff off, ios::seekdir way,
//              ios::openmode which = ios::in | ios::out);      inherited
//      virtual streampos seekpos(streampos sp,
//              ios::openmode which = ios::in | ios::out);      inherited
//      virtual streambuf* setbuf(char* s, int 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
//      int alsize;     exposition only
//      void* (*palloc)(size_t);        exposition only
//      void (*pfree)(void*);   exposition only
};
```

1 The class `strstreambuf` is derived from `streambuf` 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. For the sake of exposition, these are represented as elements of a bitmask type (indicated here as *T1*) called *strstate*. The elements are:

— *allocated*, set when a dynamic array object has been allocated, and hence should be freed by the destructor for the `strstreambuf` object;

— *constant*, set when the array object has `const` elements, so the output sequence cannot be written;

— *dynamic*, set when the array object is allocated (or reallocated) as necessary to hold a character sequence that can change in length;

— *frozen*, set when the program has requested that the array object not be altered, reallocated, or freed.

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

— *strstate strmode*, the attributes of the array object associated with the `strstreambuf` object;

— `int` *alsize*, the suggested minimum size for a dynamic array object;

— `void* (*`*palloc*`)(size_t)`, points to the function to call to allocate a dynamic array object;

— `void (*`*pfree*`)(void*)`, points to the function to call to free a dynamic array object.

3 Each object of class `strstreambuf` has a *seekable area,* delimited by the pointers *seeklow* and *seekhigh*. If *gnext* is a null pointer, the seekable area is undefined. Otherwise, *seeklow* equals *gbeg* and *seekhigh* is either *pend*, if *pend* is not a null pointer, or *gend*.

17.4.6.1.1 `strstreambuf::strstreambuf(int)` [lib.cons.strstreambuf.i]

```
strstreambuf(int alsize_arg = 0);
```

1 Constructs an object of class `strstreambuf`, initializing the base class with `streambuf()`, and initializing:

— *strmode* with *dynamic*;

— *alsize* with *alsize_arg*;

— *palloc* with a null pointer;

— *pfree* with a null pointer.

17.4.6.1.2 `strstreambuf::strstreambuf(void* (*)(size_t), void (*)(void*))` [lib.cons.strstreambuf.ff]

```
strstreambuf(void* (*palloc_arg)(size_t), void (*pfree_arg)(void*));
```

1 Constructs an object of class `strstreambuf`, initializing the base class with `streambuf()`, and initializing:

— *strmode* with *dynamic*;

— *alsize* with an unspecified value;

— *palloc* with *palloc_arg*;

— *pfree* with *pfree_arg*.

17.4.6.1.3 strstreambuf::strstreambuf(char*, int, char*) [lib.cons.strstreambuf.str]

```
strstreambuf(char* gnext_arg, int n, char *pbeg_arg = 0);
```

1 Constructs an object of class strstreambuf, initializing the base class with streambuf(), and initializing:

— *strmode* with zero;

— *alsize* with an unspecified value;

— *palloc* with a null pointer;

— *pfree* with a null pointer.

2 *gnext_arg* shall point to the first element of an array object whose number of elements *N* is determined as follows:

— If *n* > 0, *N* is *n*.

— If *n* == 0, *N* is strlen(*gnext_arg*).

— If *n* < 0, *N* is INT_MAX.

3 The function signature strlen(const char*) is declared in <cstring> (17.2). The macro INT_MAX is defined in <climits>.

4 If pbeg_arg is a null pointer, the function executes:

```
setg(gnext_arg, gnext_arg, gnext_arg + N);
```

5 Otherwise, the function executes:

```
setg(gnext_arg, gnext_arg, pbeg_arg);
setp(pbeg_arg, pbeg_arg + N);
```

17.4.6.1.4 strstreambuf::strstreambuf(unsigned char*, int, unsigned char*) [lib.cons.strstreambuf.ustr]

```
strstreambuf(unsigned char* gnext_arg, int n,
        unsigned char* pbeg_arg = 0);
```

1 Behaves the same as strstreambuf((char*)*gnext_arg*, *n*, (char*)*pbeg_arg*).

17.4.6.1.5 strstreambuf::strstreambuf(signed char*, int, signed char*) [lib.cons.strstreambuf.sstr]

```
strstreambuf(signed char* gnext_arg, int n,
        signed char* pbeg_arg = 0);
```

1 Behaves the same as strstreambuf((char*)*gnext_arg*, *n*, (char*)*pbeg_arg*).

17.4.6.1.6 strstreambuf::strstreambuf(const char*, int) [lib.cons.strstreambuf.cstr]

```
strstreambuf(const char* gnext_arg, int n);
```

1 Behaves the same as strstreambuf((char*)*gnext_arg*, *n*), except that the constructor also sets *constant* in *strmode*.

17.4.6.1.7 `strstreambuf::strstreambuf(const unsigned char*, int)` [lib.cons.strstreambuf.custr]

```
strstreambuf(const unsigned char* gnext_arg, int n);
```

1 Behaves the same as `strstreambuf((const char*)`*gnext_arg*`,` *n*`)`.

17.4.6.1.8 `strstreambuf::strstreambuf(const signed char*, int)` [lib.cons.strstreambuf.csstr]

```
strstreambuf(const signed char* gnext_arg, int n);
```

1 Behaves the same as `strstreambuf((const char*)`*gnext_arg*`,` *n*`)`.

17.4.6.1.9 `strstreambuf::~strstreambuf()` [lib.des.strstreambuf]

```
virtual ~strstreambuf();
```

1 Destroys an object of class `strstreambuf`. The function frees the dynamically allocated array object only if *strmode* `&` *allocated* is nonzero and *strmode* `&` *frozen* is zero. (Subclause _strstreambuf::overflow_ describes how a dynamically allocated array object is freed.)

17.4.6.1.10 `strstreambuf::freeze(int)` [lib.strstreambuf::freeze]

```
void freeze(bool freezefl = 1);
```

1 If *strmode* `&` *dynamic* is nonzero, alters the freeze status of the dynamic array object as follows: If *freezefl* is nonzero, the function sets *frozen* in *strmode*. Otherwise, it clears *frozen* in `strmode`.

17.4.6.1.11 `strstreambuf::str()` [lib.strstreambuf::str]

```
char* str();
```

1 Calls `freeze()`, then returns the beginning pointer for the input sequence, *gbeg*.[113)]

17.4.6.1.12 `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, the function returns the current effective length of the array object as the next pointer minus the beginning pointer for the output sequence, *pnext* `-` *pbeg*.

17.4.6.1.13 `strstreambuf::overflow(int)` [lib.strstreambuf::overflow]

113) The return value can be a null pointer.

Box 116

Library WG issue: Jerry Schwarz, January 3, 1994

`overflow`: *

This is essentially editorial. I think the words Library uses here (and in general describing specializations of `streambuf`) are wrong. *Library* says ''Behaves the same as `streambuf::underflow(int)` with the following specific behavior.'' But `streambuf::underflow(int)` returns `EOF` unconditionally.

What *Library* is trying to say is something like ''it implements the protocol defined for `streambuf::underflow` with the following specific behavior.''

I think the right thing to do is make these descriptions self contained.

I was wrong here. Sorry. Comparing *Library* with the current draft convinces me that when the function can be described as a specialization of a protocol it is better to do that. All the repetitions of the protocol in the current draft mean you have to compare lots of identical verbiage to see how various functions differ >from each other. |

But I think it is essential that the protocol itself indicate what needs to be specified in a specialization.

```
//      virtual int overflow(int 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 `(unsigned char)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 The function returns `EOF` to indicate failure.

4 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.[114] 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 char[n]`. In either case, if the allocation fails, the function returns `EOF`. Otherwise, it sets `allocated` in *strmode*.

5 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`.

6 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.

[114] An implementation should consider `alsize` in making this decision.

17.4.6.1.14 `strstreambuf::pbackfail(int)` [lib.strstreambuf::pbackfail]

`//      virtual int pbackfail(int c = EOF);` ***inherited***

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 `(unsigned char)c == unsigned char)gnext[-1]`, the function assigns `gnext - 1` to *gnext*. The function signals success by returning `(unsigned char)c`.

— If `c != EOF`, if the input sequence has a putback position available, and if `strmode & constant` is zero, the function assigns *c* to `*--gnext`. The function signals success by returning `(unsigned char)c`.

— If `c == EOF` and if the input sequence has a putback position available, the function assigns `gnext - 1` to *gnext*. The function signals success by returning `(unsigned char)c`.

2 If the function can succeed in more than one of these ways, it is unspecified which way is chosen. The function can alter the number of putback positions available as a result of any call.

3 The function returns `EOF` to indicate failure. |

17.4.6.1.15 `strstreambuf::showmany()` | [lib.strstreambuf::showmany]

`//      virtual int showmany();` ***inherited*** |

1 Behaves the same as `streambuf::showmany(int)`. |

17.4.6.1.16 `strstreambuf::underflow()` [lib.strstreambuf::underflow]

`//      virtual int underflow();` ***inherited***

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 `(unsigned char)*gnext`.

— Otherwise, if the current write next pointer *pnext* is not a null pointer and is greater than the current read end pointer *gend*, the function makes a read position available by assigning to *gend* a value greater than *gnext* and no greater than *pnext*. The function signals success by returning `(unsigned char)*gnext`.

2 The function can alter the number of read positions available as a result of any call.

3 The function returns `EOF` to indicate failure.

17.4.6.1.17 `strstreambuf::uflow()` [lib.strstreambuf::uflow]

`//      virtual int uflow();` ***inherited***

1 Behaves the same as `streambuf::uflow(int)`.

17.4.6.1.18 `strstreambuf::xsgetn(char*, int)` [lib.strstreambuf::xsgetn]

`//      virtual int xsgetn(char* s, int n);` ***inherited***

1 Behaves the same as `streambuf::xsgetn(char*, int)`.

17.4.6.1.19 `strstreambuf::xsputn(const char*, int)` [lib.strstreambuf::xsputn]

```
//      virtual int xsputn(const char* s, int n);          inherited
```

1 Behaves the same as `streambuf::xsputn(char*, int)`.

17.4.6.1.20 `strstreambuf::seekoff(streamoff, ios::seekdir, ios::openmode)` [lib.strstreambuf::seekoff]

```
//      virtual streampos seekoff(streamoff off, ios::seekdir way,       *
//              ios::openmode which = ios::in | ios::out);      inherited
```

1 Alters the stream position within one of the controlled sequences, if possible, as described below. The function returns `streampos(`*newoff*`)`, constructed from the resultant offset *newoff* (of type `streamoff`), 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.

2 If *which* `& ios::in` is nonzero, the function positions the input sequence. Otherwise, if *which* `& ios::out` is nonzero, the function positions the output sequence. Otherwise, if *which* `& (ios::in | ios::out)` equals `ios::in | ios::out` and if *way* equals either `ios::beg` or `ios::end`, the function positions both the input and the output sequences. Otherwise, the positioning operation fails.

3 For a sequence to be positioned, if its next pointer is a null pointer, the positioning operation fails. Otherwise, the function determines *newoff* in one of three ways:

— If *way* `== ios::beg`, *newoff* is zero.

— If *way* `== ios::cur`, *newoff* is the next pointer minus the beginning pointer (*xnext* `-` *xbeg*).

— If *way* `== ios::end`, *newoff* is *seekhigh* minus the beginning pointer (*seekhigh* `-` *xbeg*). |

4 If *newoff* `+` *off* is less than *seeklow* `-` *xbeg*, or if *seekhigh* `-` *xbeg* is less than *newoff* `+` *off*, the positioning operation fails. Otherwise, the function assigns *xbeg* `+` *newoff* `+` *off* to the next pointer *xnext*.

17.4.6.1.21 `strstreambuf::seekpos(streampos, ios::openmode)` [lib.strstreambuf::seekpos]

```
//      virtual streampos seekpos(streampos sp,
//              ios::openmode which = ios::in | ios::out);      inherited
```

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). The function returns `streampos(`*newoff*`)`, constructed from the resultant offset *newoff* (of type `streamoff`), 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.

2 If *which* `& ios::in` is nonzero, the function positions the input sequence. If *which* `& ios::out` is nonzero, the function positions the output sequence. If the function positions neither sequence, the positioning operation fails.

3 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*.

17.4.6.1.22 `strstreambuf::setbuf(char*, int)` [lib.strstreambuf::setbuf]

```
//      virtual streambuf* setbuf(char* s, int n);        inherited
```

1 Performs an operation that is defined separately for each class derived from `strstreambuf`.

2 The default behavior is the same as for `streambuf::setbuf(char*, int)`.

17.4.6.1.23 `strstreambuf::sync()` [lib.strstreambuf::sync]

```
//      virtual int sync();       inherited
```

1 Behaves the same as `streambuf::sync()`.

17.4.6.2 Class `istrstream` [lib.istrstream]

```
class istrstream : public istream {
public:
        istrstream(const char* s);
        istrstream(const char* s, int n);
        istrstream(char* s);
        istrstream(char* s, int n);
        virtual ~istrstream();
        strstreambuf* rdbuf() const;
        char *str();
private:
//      strstreambuf sb;          exposition only
};
```

1 The class `istrstream` is a derivative of `istream` that assists in the reading of objects of class `strstreambuf`. It supplies a `strstreambuf` object to control the associated array object. For the sake of exposition, the maintained data is presented here as:

— *sb*, the `strstreambuf` object.

17.4.6.2.1 `istrstream::istrstream(const char*)` [lib.cons.istrstream.cstr]

```
istrstream(const char* s);
```

1 Constructs an object of class `istrstream`, initializing the base class with `istream(&sb)`, and initializing *sb* with `sb(s, 0)`. *s* shall designate the first element of an NTBS.

17.4.6.2.2 `istrstream::istrstream(const char*, int)` [lib.cons.istrstream.cstrn]

```
istrstream(const char* s, int n);
```

1 Constructs an object of class `istrstream`, initializing the base class with `istream(&sb)`, and initializing *sb* with `sb(s, n)`. *s* shall designate the first element of an array whose length is *n* elements, and *n* shall be greater than zero.

17.4.6.2.3 `istrstream::istrstream(char*)` [lib.cons.istrstream.str]

```
istrstream(char* s);
```

1 Constructs an object of class `istrstream`, initializing the base class with `istream(&sb)`, and initializing *sb* with `sb((const char*)s, 0)`. *s* shall designate the first element of an NTBS.

17.4.6.2.4 `istrstream::istrstream(char*, int)` [lib.cons.istrstream.strn]

```
istrstream(char* s, int n);
```

1 Constructs an object of class `istrstream`, initializing the base class with `istream(&sb)`, and initializing *sb* with `sb((const char*)s, n)`. *s* shall designate the first element of an array whose length is *n* elements, and *n* shall be greater than zero.

17.4.6.2.5 `istrstream::~istrstream()` [lib.des.istrstream]

```
virtual ~istrstream();
```

1 Destroys an object of class `istrstream`.

17.4.6.2.6 `istrstream::rdbuf()` [lib.istrstream::rdbuf]

```
strstreambuf* rdbuf() const;
```

1 Returns `(strstreambuf*)&sb`.

17.4.6.2.7 `istrstream::str()` [lib.istrstream::str]

```
char* str();
```

1 Returns `sb.str()`.

17.4.6.3 Class `ostrstream` [lib.ostrstream]

```
class ostrstream : public ostream {
public:
        ostrstream();
        ostrstream(char* s, int n, openmode mode = out);
        virtual ~ostrstream();
        strstreambuf* rdbuf() const;
        void freeze(int freezefl = 1);
        char* str();
        int pcount() const;
private:
//      strstreambuf sb;            exposition only
};
```

1 The class `ostrstream` is a derivative of `ostream` that assists in the writing of objects of class `strstreambuf`. It supplies a `strstreambuf` object to control the associated array object. For the sake of exposition, the maintained data is presented here as:

— *sb*, the `strstreambuf` object.

17.4.6.3.1 `ostrstream::ostrstream()` [lib.cons.ostrstream]

```
ostrstream();
```

1 Constructs an object of class `ostrstream`, initializing the base class with `ostream(&sb)`, and initializing *sb* with `sb()`.

17.4.6.3.2 `ostrstream::ostrstream(char*, int, openmode)` [lib.cons.ostrstream.str]

```
ostrstream(char* s, int n, openmode mode = out);
```

1 Constructs an object of class `ostrstream`, initializing the base class with `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(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*. The constructor is `sb(s, n, s + ::strlen(s))`.

2 The function signature `strlen(const char*)` is declared in `<cstring>` (17.2).

17.4.6.3.3 `ostrstream::~ostrstream()` [lib.des.ostrstream]

```
virtual ~ostrstream();
```

1 Destroys an object of class `ostrstream`.

17.4.6.3.4 `ostrstream::rdbuf()` [lib.ostrstream::rdbuf]

```
strstreambuf* rdbuf() const;
```

1 Returns `(strstreambuf*)&sb`.

17.4.6.3.5 `ostrstream::freeze(int)` [lib.ostrstream::freeze]

```
void freeze(int freezefl = 1);
```

1 Calls `sb.freeze(freezefl)`.

17.4.6.3.6 `ostrstream::str()` [lib.ostrstream::str]

```
char* str();
```

1 Returns `sb.str()`.

17.4.6.3.7 `ostrstream::pcount()` [lib.ostrstream::pcount]

```
int pcount() const;
```

1 Returns `sb.pcount()`.

17.4.7 Header `<sstream>` [lib.header.sstream]

1 The header `<sstream>` defines three types that associate stream buffers with objects of class `string`, as described in subclause _string_.

17.4.7.1 Class `stringbuf` [lib.stringbuf]

Box 117

Library WG issue: Jerry Schwarz, January 3, 1994

Formulating the ''as if'' rule is an interesting exercise. If the sequence is represented by `a` (i.e. the sequence is (a[0], ... a[max]) and the put pointer is at `px` and the get pointer is at `gx` then the rule requires the pointers to be such that.

a) `pbeg==NULL` or for all `i` such that
```
        px-(pnext-pbeg) <= i < px, a[i]==pbeg[i-px]
```

b) `gbeg==NULL` or for all `i`s such that
```
        gx-(gnext-gbeg) <= i < gx+(gend-gbeg), a[i]==gnext[i-px]
```

c) for any `i` such that both *
```
                px-(pnext-pbeg) <= i < px
```
and
```
                gx-(gnext-gbeg) <= i < gx+(gend-gbeg)
                pnext+(i-px) == gnext + (i-gx)
```

If my alternative protocols are accepted, essentially the same conditions are achieved by specializing so that the input and output streams are represented by
```
    Stream s ;
    size_t px;
    size_t gx;
```

I'll be happy to elaborate on any of the above.

```
class stringbuf : public streambuf {
public:
        stringbuf(ios::openmode which = ios::in | ios::out);
        stringbuf(const string& str,
                ios::openmode which = ios::in | ios::out);
        virtual ~stringbuf();
        string str() const;
        void str(const string& str_arg);
protected:
//      virtual int overflow(int c = EOF);       inherited
//      virtual int pbackfail(int c = EOF);      inherited
//      virtual int showmany(); inherited
//      virtual int underflow();         inherited
//      virtual int uflow();     inherited
//      virtual int xsgetn(char* s, int n);      inherited
//      virtual int xsputn(const char* s, int n);        inherited
//      virtual streampos seekoff(streamoff off, ios::seekdir way,
//              ios::openmode which = ios::in | ios::out);       inherited
//      virtual streampos seekpos(streampos sp,
//              ios::openmode which = ios::in | ios::out);       inherited
//      virtual streambuf* setbuf(char* s, int n);       inherited
//      virtual int sync();      inherited
private:
//      ios::openmode mode;      exposition only
};
```

1 The class `stringbuf` is derived from `streambuf` to associate possibly the input sequence and possibly the output sequence with a sequence of arbitrary (single-byte) characters. The sequence can be initialized from, or made available as, an object of class `string`.

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

— `ios::openmode` *mode*, has `ios::in` set if the input sequence can be read, and `ios::out` set if the output sequence can be written.

3 For the sake of exposition, the stored character sequence is described here as an array object.

17.4.7.1.1 `stringbuf::stringbuf(ios::openmode)` [lib.cons.stringbuf.m]

```
stringbuf(ios::openmode which = ios::in | ios::out);
```
*

1 Constructs an object of class `stringbuf`, initializing the base class with `streambuf()`, and initializing *mode* with *which*. The function allocates no array object.

17.4.7.1.2 `stringbuf::stringbuf(const string&, ios::openmode)` [lib.cons.stringbuf.sm]

```
stringbuf(const string& str, ios::openmode which = ios::in | ios::out);
```
*

1 Constructs an object of class `stringbuf`, initializing the base class with `streambuf()`, and initializing *mode* with *which*.

2 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* `& ios::in` is nonzero, the function executes:

```
setg(x, x, x + n);
```

3 If *which* `& ios::out` is nonzero, the function executes:

```
setp(x, x + n);
```

17.4.7.1.3 `stringbuf::~stringbuf()` [lib.des.stringbuf]

```
virtual ~stringbuf();
```

1 Destroys an object of class `stringbuf`.

17.4.7.1.4 `stringbuf::str()` [lib.stringbuf::str]

```
string str() const;
```

1 If *mode* `& ios::in` is nonzero and *gnext* is not a null pointer, returns `string(`*gbeg*`,` *gend* `-` *gbeg*`)`. Otherwise, if *mode* `& ios::out` is nonzero and *pnext* is not a null pointer, the function returns `string(`*pbeg*`,` *pptr* `-` *pbeg*`)`. Otherwise, the function returns `string()`. |

17.4.7.1.5 `stringbuf::str(const string&)` [lib.stringbuf::str.s]

```
void str(const string& 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* `& ios::in` is nonzero, the function executes:

```
setg(x, x, x + n);
```

3 If *which* & ios::out is nonzero, the function executes:

```
setp(x, x + n);
```

17.4.7.1.6 `stringbuf::overflow(int)` [lib.stringbuf::overflow]

```
//      virtual int overflow(int 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 (unsigned char)*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 The function returns EOF to indicate failure.

4 The function can make a write position available only if *mode* & ios::out is nonzero. To make a write position available, the function reallocates (or initially allocates) an array object with a sufficient number of elements to hold the current array object (if any), plus one additional write position. If *mode* & ios::in is nonzero, the function alters the read end pointer *gend* to point just past the new write position (as does the write end pointer *pend*).

17.4.7.1.7 `stringbuf::pbackfail(int)` [lib.stringbuf::pbackfail]

```
//      virtual int pbackfail(int c = EOF);
```
inherited

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 (unsigned char)*c* == (unsigned char)*gnext*[-1], the function assigns *gnext* - 1 to *gnext*. The function signals success by returning (unsigned char)*c*.

— If *c* != EOF, if the input sequence has a putback position available, and if *mode* & *ios::out* is nonzero, the function assigns *c* to *--*gnext*. The function signals success by returning (unsigned char)*c*.

— If *c* == EOF and if the input sequence has a putback position available, the function assigns *gnext* - 1 to *gnext*. The function signals success by returning (unsigned char)*c*.

2 If the function can succeed in more than one of these ways, it is unspecified which way is chosen.

3 The function returns EOF to indicate failure.

17.4.7.1.8 `stringbuf::showmany()` [lib.stringbuf::showmany]

```
//      virtual int showmany();
```
inherited

1 Behaves the same as streambuf::showmany(int).

17.4.7.1.9 `stringbuf::underflow()` [lib.stringbuf::underflow]

Box 118

Library WG issue: Jerry Schwarz, January 3, 1994

`Underflow` needs to consider that the sequence might have been extended with `overflows` from its initial state. *

```
//      virtual int underflow();          inherited
```

1 If the input sequence has a read position available, signals success by returning `(unsigned char)*gnext`. Otherwise, the function returns `EOF` to indicate failure.

17.4.7.1.10 `stringbuf::uflow()` [lib.stringbuf::uflow]

```
//      virtual int uflow();     inherited
```

1 Behaves the same as `streambuf::uflow(int)`.

17.4.7.1.11 `stringbuf::xsgetn(char*, int)` [lib.stringbuf::xsgetn]

```
//      virtual int xsgetn(char* s, int n);       inherited
```

1 Behaves the same as `streambuf::xsgetn(char*, int)`.

17.4.7.1.12 `stringbuf::xsputn(const char*, int)` [lib.stringbuf::xsputn]

```
//      virtual int xsputn(const char* s, int n);        inherited
```

1 Behaves the same as `streambuf::xsputn(char*, int)`.

17.4.7.1.13 `stringbuf::seekoff(streamoff, ios::seekdir, ios::openmode)` [lib.stringbuf::seekoff]

```
//      virtual streampos seekoff(streamoff off, ios::seekdir way,
//              ios::openmode which = ios::in | ios::out);       inherited
```

1 Alters the stream position within one of the controlled sequences, if possible, as described below. The function returns `streampos(`*newoff*`)`, constructed from the resultant offset *newoff* (of type `streamoff`), 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.

2 If *which* `& ios::in` is nonzero, the function positions the input sequence. Otherwise, if *which* `& ios::out` is nonzero, the function positions the output sequence. Otherwise, if *which* `& (ios::in | ios::out)` equals `ios::in | ios::out` and if *way* equals either `ios::beg` or `ios::end`, the function positions both the input and the output sequences. Otherwise, the positioning operation fails.

3 For a sequence to be positioned, if its next pointer is a null pointer, the positioning operation fails. Otherwise, the function determines *newoff* in one of three ways:

— If *way* `== ios::beg`, *newoff* is zero.

— If *way* `== ios::cur`, *newoff* is the next pointer minus the beginning pointer (*xnext* `-` *xbeg*).

— If *way* `== ios::end`, *newoff* is the end pointer minus the beginning pointer (*xend* `-` *xbeg*).

4 If *newoff* + *off* is less than zero, or if *xend* − *xbeg* is less than *newoff* + *off*, the positioning operation fails. Otherwise, the function assigns *xbeg* + *newoff* + *off* to the next pointer *xnext*.

17.4.7.1.14 `stringbuf::seekpos(streampos, ios::openmode)` [lib.stringbuf::seekpos]

Box 119

Library WG issue: Jerry Schwarz, January 3, 1994

Also it should be possible to seek the input stream anywhere in the sequence, even if it has been extended.

Box 120

Library WG issue: Jerry Schwarz, January 3, 1994

Seeking to position 0 should be allowed even when the sequence is empty.

```
//      virtual streampos seekpos(streampos sp,
//              ios::openmode which = ios::in | ios::out);       inherited
```

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). The function returns `streampos(newoff)`, constructed from the resultant offset *newoff* (of type `streamoff`), 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.

2 If *which* & `ios::in` is nonzero, the function positions the input sequence. If *which* & `ios::out` is nonzero, the function positions the output sequence. If the function positions neither sequence, the positioning operation fails.

3 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*.

17.4.7.1.15 `stringbuf::setbuf(char*, int)` [lib.stringbuf::setbuf]

```
//      virtual streambuf* setbuf(char* s, int n);       inherited
```

1 Performs an operation that is defined separately for each class derived from `stringbuf`.

2 The default behavior is the same as for `streambuf::setbuf(char*, int)`.

17.4.7.1.16 `stringbuf::sync()` [lib.stringbuf::sync]

```
//      virtual int sync();       inherited
```

1 Behaves the same as `streambuf::sync()`.

17.4.7.2 Class `istringstream` [lib.istringstream]

```
class istringstream : public istream {
public:
        istringstream(ios::openmode which = ios::in);
        istringstream(const string& str, ios::openmode which = ios::in);
        virtual ~istringstream();
        stringbuf* rdbuf() const;
        string str() const;
        void str(const string& str);
private:
//      stringbuf sb;     exposition only
};
```

1 The class `istringstream` is a derivative of `istream` that assists in the reading of objects of class `stringbuf`. It supplies a `stringbuf` object to control the associated array object. For the sake of exposition, the maintained data is presented here as:

— *sb*, the `stringbuf` object.

17.4.7.2.1 `istringstream::istringstream(ios::openmode)` [lib.cons.istringstream.m]

```
istringstream(ios::openmode which = ios::in);
```

1 Constructs an object of class `istringstream`, initializing the base class with `istream(&sb)`, and initializing *sb* with `sb(which)`.

17.4.7.2.2 `istringstream::istringstream(const string&, ios::openmode` [lib.cons.istringstream.sm]

```
istringstream(const string& str, ios::openmode which = ios::in);
```

1 Constructs an object of class `istringstream`, initializing the base class with `istream(&sb)`, and initializing *sb* with `sb(str, which)`.

17.4.7.2.3 `istringstream::~istringstream()` [lib.des.istringstream]

```
virtual ~istringstream();
```

1 Destroys an object of class `istringstream`.

17.4.7.2.4 `istringstream::rdbuf()` [lib.istringstream::rdbuf]

```
stringbuf* rdbuf() const;
```

1 Returns `(stringbuf*)&sb`.

17.4.7.2.5 `istringstream::str()` [lib.istringstream::str]

```
string str() const;
```

1 Returns `sb.str()`.

17.4.7.2.6 `istringstream::str(const string&)` [lib.istringstream::str.s]

```
void str(const string& str_arg);
```

1 Calls `sb.str(str_arg)`.

17.4.7.3 Class `ostringstream` [lib.ostringstream]

```
class ostringstream : public ostream {
public:
        ostringstream(ios::openmode which = ios::out);
        ostringstream(const string& str, ios::openmode which = ios::out);
        virtual ~ostringstream();
        stringbuf* rdbuf() const;
        string str() const;
        void str(const string& str);
private:
//      stringbuf sb;     exposition only
};
```

1 The class `ostringstream` is a derivative of `ostream` that assists in the writing of objects of class `stringbuf`. It supplies a `stringbuf` object to control the associated array object. For the sake of exposition, the maintained data is presented here as:

— *sb*, the `stringbuf` object.

17.4.7.3.1 `ostringstream::ostringstream(ios::openmode)` [lib.cons.ostringstream.m]

```
ostringstream(ios::openmode which = ios::out);
```

1 Constructs an object of class `ostringstream`, initializing the base class with `ostream(&sb)`, and initializing *sb* with `sb(which)`.

17.4.7.3.2 `ostringstream::ostringstream(const string&, ios::openmode` [lib.cons.ostringstream.sm]

```
ostringstream(const string& str, ios::openmode which = ios::out);
```

1 Constructs an object of class `ostringstream`, initializing the base class with `ostream(&sb)`, and initializing *sb* with `sb(str, which)`.

17.4.7.3.3 `ostringstream::~ostringstream()` [lib.des.ostringstream]

```
virtual ~ostringstream();
```

1 Destroys an object of class `ostringstream`.

17.4.7.3.4 `ostringstream::rdbuf()` [lib.ostringstream::rdbuf]

```
stringbuf* rdbuf() const;
```

1 Returns `(stringbuf*)&sb`.

17.4.7.3.5 `ostringstream::str()` [lib.ostringstream::str]

```
string str() const;
```

1 Returns `sb.str()`.

17.4.7.3.6 `ostringstream::str(const string&)` [lib.ostringstream::str.s]

```
void str(const string& str_arg);
```

1 Calls `sb.str(str_arg)`.

17.4.8 Header `<fstream>` [lib.header.fstream]

1 The header `<fstream>` defines six types that associate stream buffers with files and assist reading and writing files.

2 In this subclause, the type name *`FILE`* is a synonym for the type `FILE` defined in `<cstdio>` (17.2).

17.4.8.1 Class `filebuf` [lib.filebuf]

> **Box 121**
>
> **Library WG issue:** Jerry Schwarz, January 3, 1994
>
> Something needs to be said about setting of pointers. `pbeg, pend, pnext` should all be set to `NULL`.
>
> The `g` pointers are more delicate. The intention was that you throw away the get area and (if necessary) seek the file. Some implementor's haven't done the seek, or ignore failures. This gives you a way to throw away (some or all of) input from a terminal. We ought to say something about this. As the draft now reads it appears that the `g` pointers can't be modified.

```
class filebuf : public streambuf {
public:
        filebuf();
        virtual ~filebuf();
        bool is_open() const;
        filebuf* open(const char* s, ios::openmode mode);
        filebuf* close();
protected:
//      virtual int overflow(int c = EOF);       inherited
//      virtual int pbackfail(int c = EOF);      inherited
//      virtual int showmany(); inherited
//      virtual int underflow();          inherited
//      virtual int uflow();     inherited
//      virtual int xsgetn(char* s  int n);      inherited
//      virtual int xsputn(const char* s, int n);        inherited
//      virtual streampos seekoff(streamoff off, ios::seekdir way,
//              ios::openmode which = ios::in | ios::out);       inherited
//      virtual streampos seekpos(streampos sp,
//              ios::openmode which = ios::in | ios::out);       inherited
//      virtual streambuf* setbuf(char* s, int n);       inherited
//      virtual int sync();      inherited
private:
//      FILE* file;        exposition only
};
```

1 The class `filebuf` is derived from `streambuf` to associate both the input sequence and the output sequence with an object of type `FILE`. For the sake of exposition, the maintained data is presented here as:

— *`FILE *file`*, points to the `FILE` associated with the object of class `filebuf`.

2 The restrictions on reading and writing a sequence controlled by an object of class `filebuf` 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.

17.4.8.1.1 `filebuf::filebuf()` [lib.cons.filebuf]

```
filebuf();
```

1 Constructs an object of class `filebuf`, initializing the base class with `streambuf()`, and initializing *`file`* to a null pointer.

17.4.8.1.2 `filebuf::~filebuf()` [lib.des.filebuf]

```
virtual ~filebuf();
```

1 Destroys an object of class `filebuf`. The function calls `close()`.

17.4.8.1.3 `filebuf::is_open()` [lib.filebuf::is.open]

```
bool is_open() const;
```

1 Returns a nonzero value if *`file`* is not a null pointer.

17.4.8.1.4 `filebuf::open(const char*, ios::openmode)` [lib.filebuf::open]

```
filebuf* open(const char* s, ios::openmode mode);
```

1 If *`file`* is not a null pointer, returns a null pointer. Otherwise, the function opens a file, if possible, whose name is the NTBS *`s`*, by calling `fopen(`*`s`*`, `*`modstr`*`)` and assigning the return value to *`file`*. The NTBS *`modstr`* is determined from *`mode`*` & ~ios::ate` as follows:

— `ios::in` becomes `"r"`;

— `ios::out | ios::trunc` becomes `"w"`;

— `ios::out | ios::app` becomes `"a"`;

— `ios::in | ios::binary` becomes `"rb"`;

— `ios::out | ios::trunc | ios::binary` becomes `"wb"`;

— `ios::out | ios::app | ios::binary` becomes `"ab"`;

— `ios::in | ios::out` becomes `"r+"`;

— `ios::in | ios::out | ios::trunc` becomes `"w+"`;

— `ios::in | ios::out | ios::app` becomes `"a+"`;

— `ios::in | ios::out | ios::binary` becomes `"r+b"`;

— `ios::in | ios::out | ios::trunc | ios::binary` becomes `"w+b"`;

— `ios::in | ios::out | ios::app | ios::biaary` becomes `"a+b"`.

2 If the resulting *`file`* is not a null pointer and *`mode & ios::ate`* is nonzero, the function calls `fseek(`*`file`*`, 0, SEEK_END)`. If that function returns a null pointer, the function calls `close()` and returns a null pointer. Otherwise, the function returns `this`.

3 The macro `SEEK_END` is defined, and the function signatures `fopen(const char*, const char*)` and `fseek(FILE*, long, int)` are declared, in <cstdio> (17.2).

17.4.8.1.5 `filebuf::close()` [lib.filebuf::close]

> **Box 122**
> **Library WG issue:** Jerry Schwarz, January 3, 1994
>
> I think `close` should assign 0 to `file`.
>
> Not fixed.

```
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`. Otherwise, it returns a null pointer.

2 The function signature `fclose(FILE*)` is declared, in `<cstdio>` (17.2).

17.4.8.1.6 `filebuf::overflow(int)` [lib.filebuf::overflow]

```
//      virtual int overflow(int c = EOF);        inherited
```

1 Appends the character designated by *`c`* to the output sequence, if possible, in one of three ways:

— If *`c`* `!= EOF` and if either the output sequence has a write position available or the function makes a write position available (in an unspecified manner), the function assigns *`c`* to `*`*`pnext`*`++`. The function signals success by returning `(unsigned char)`*`c`*.

— If *`c`* `!= EOF`, the function appends *`c`* directly to the associated output sequence (as described below). If *`pbeg`* `<` *`pnext`*, the *`pnext`* `-` *`pbeg`* characters beginning at *`pbeg`* are first appended directly to the associated output sequence, beginning with the character at *`pbeg`*. The function signals success by returning `(unsigned char)`*`c`*.

— If *`c`* `== EOF`, there is no character to append. The function signals success by returning a value other than `EOF`.

2 If the function can succeed in more than one of these ways, it is unspecified which way is chosen. The function can alter the number of write positions available as a result of any call.

3 The function returns `EOF` to indicate failure. If *`file`* is a null pointer, the function always fails.

4 To append a character *`x`* directly to the associated output sequence, the function evaluates the expression:

```
fputc(x, file) == x
```

5 which must be nonzero. The function signature `fputc(int, FILE*)` is declared in `<cstdio>` (17.2).

17.4.8.1.7 `filebuf::pbackfail(int)` [lib.filebuf::pbackfail]

```
//      virtual int pbackfail(int c = EOF);       inherited
```

1 Puts back the character designated by *`c`* to the input sequence, if possible, in one of four ways:

— If *`c`* `!= EOF` and if either the input sequence has a putback position available or the function makes a putback position available (in an unspecified manner), the function assigns *`c`* to `*--`*`gnext`*. The function signals success by returning `(unsigned char)`*`c`*.

— If *`c`* `!= EOF` and if no putback position is available, the function puts back *`c`* directly to the associate input sequence (as described below). The function signals success by returning `(unsigned char)`*`c`*.

— If *`c`* `== EOF` and if either the input sequence has a putback position available or the function makes a putback position available, the function assigns *gnext* - 1 to *gnext*. The function signals success by returning (unsigned char)*c*.

— If *c* == EOF, if no putback position is available, and if the function can determine the character *x* immediately before the current position in the associated input sequence (in an unspecified manner), the function puts back *x* directly to the associated input sequence. The function signals success by returning a value other than EOF.

2 If the function can succeed in more than one of these ways, it is unspecified which way is chosen. The function can alter the number of putback positions available as a result of any call.

3 The function returns EOF to indicate failure. If *file* is a null pointer, the function always fails.

4 To put back a character *x* directly to the associated input sequence, the function evaluates the expression:

```
ungetc(x, file) == x
```

5 which must be nonzero. The function signature ungetc(int, FILE*) is declared in <cstdio> (17.2).

17.4.8.1.8 filebuf::showmany() [lib.filebuf::showmany]

```
//      virtual int showmany(); inherited
```

1 Behaves the same as streambuf::showmany().[115)]

17.4.8.1.9 filebuf::underflow() [lib.filebuf::underflow]

```
//      virtual int underflow();          inherited
```

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 (unsigned char)**gnext*.

— Otherwise, if the function can determine the character *x* at the current position in the associated input sequence (as described below), it signals success by returning (unsigned char)*x*. If the function makes a read position available, it also assigns *x* to **gnext*.

2 The function can alter the number of read positions available as a result of any call.

3 The function returns EOF to indicate failure. If *file* is a null pointer, the function always fails.

4 To determine the character *x* (of type int) at the current position in the associated input sequence, the function evaluates the expression:

```
(x = ungetc(fgetc(file), file)) != EOF
```

5 which must be nonzero. The function signatures fgetc(FILE*) and ungetc(int, FILE*) are declared in <cstdio> (17.2).

115) 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.

17.4.8.1.10 `filebuf::uflow()` [lib.filebuf::uflow]

`// virtual int uflow();` ***inherited***

1 Reads a character from the input sequence, if possible, and moves the stream position past it, as follows:

— If the input sequence has a read position available the function signals success by returning `(unsigned char)*`*`gnext`*`++`.

— Otherwise, if the function can read the character *`x`* directly from the associated input sequence (as described below), it signals success by returning `(unsigned char)`*`x`*. If the function makes a read position available (in an unspecified manner), it also assigns *`x`* to `*`*`gnext`*.

2 The function can alter the number of read positions available as a result of any call.

3 The function returns `EOF` to indicate failure. If *`file`* is a null pointer, the function always fails.

4 To read a character into an object *`x`* (of type `int`) directly from the associated input sequence, the function evaluates the expression:

```
(x = fgetc(file)) != EOF
```

5 which must be nonzero. The function signature `fgetc(FILE*)` is declared in <cstdio> (17.2).

17.4.8.1.11 `filebuf::xsgetn(char*, int)` [lib.filebuf::xsgetn]

`// virtual int xsgetn(char* s, int n);` ***inherited***

1 Behaves the same as `streambuf::xsgetn(char*, int)`.

17.4.8.1.12 `filebuf::xsputn(const char*, int)` [lib.filebuf::xsputn]

`// virtual int xsputn(const char* s, int n);` ***inherited***

1 Behaves the same as `streambuf::xsputn(char*, int)`.

17.4.8.1.13 `filebuf::seekoff(streamoff, ios::seekdir, ios::openmode)` [lib.filebuf::seekoff]

```
//      virtual streampos seekoff(streamoff off, ios::seekdir way,
//                    ios::openmode which = ios::in | ios::out);
```
inherited

1 Alters the stream position within the controlled sequences, if possible, as described below. The function returns a newly constructed `streampos` 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.

2 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`:

— If *`way`* `== ios::beg`, the argument is `SEEK_SET`;

— If *`way`* `== ios::cur`, the argument is `SEEK_CUR`;

— If *`way`* `== ios::end`, the argument is `SEEK_END`.

3 The function then calls `fseek(`*`file, off, whence`*`)` and, if that function returns nonzero, the positioning operation fails.

4 The macros `SEEK_SET`, `SEEK_CUR`, and `SEEK_END` are defined, and the function signature `fseek(FILE*, long, int)` is declared, in `<cstdio>` (17.2).

17.4.8.1.14 `filebuf::seekpos(streampos, ios::openmode)` [lib.filebuf::seekpos]

```
//      virtual streampos seekpos(streampos sp,
//              ios::openmode which = ios::in | ios::out);      inherited
```

1 Alters the stream position within the controlled sequences, if possible, to correspond to the stream position stored in *`sp.pos`* and *`sp.fp`*.[116] The function returns a newly constructed `streampos` 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. *

2 If *`file`* is a null pointer, the positioning operation fails.

17.4.8.1.15 `filebuf::setbuf(char*, int)` [lib.filebuf::setbuf]

```
//      virtual streambuf* setbuf(char* s, int n);      inherited
```

1 Makes the array of *n* (single-byte) characters, whose first element is designated by *s*, available for use as a buffer area for the controlled sequences, if possible. If *`file`* is a null pointer, the function returns a null pointer. Otherwise, if the call `setvbuf(`*`file`*`, `*`s`*`, _IOFBF, `*`n`*`)` is nonzero, the function returns a null pointer. Otherwise, the function returns `*this`.

2 The macro `_IOFBF` is defined, and the function signature `setvbuf(FILE*, char*, int, size_t)` is declared, in `<cstdio>` (17.2).

17.4.8.1.16 `filebuf::sync()` [lib.filebuf::sync]

```
//      virtual int sync();      inherited
```
*

1 Returns zero if *`file`* is a null pointer. Otherwise, the function returns `fflush(`*`file`*`)`.

2 The function signature `fflush(FILE*)` is declared in `<cstdio>` (17.2).

17.4.8.2 Class `ifstream` [lib.ifstream]

```
class ifstream : public istream {
public:
        ifstream();
        ifstream(const char* s, openmode mode = in);
        virtual ~ifstream();
        filebuf* rdbuf() const;
        bool is_open();
        void open(const char* s, openmode mode = in);
        void close();
private:
//      filebuf fb;      exposition only
};
```

1 The class `ifstream` is a derivative of `istream` that assists in the reading of named files. It supplies a `filebuf` object to control the associated sequence. For the sake of exposition, the maintained data is presented here as:

— `filebuf` *`fb`*, the `filebuf` object.

[116] The function may, for example, call `fsetpos(`*`file`*`, `*`&sp.fp`*`)` and/or `fseek(`*`file`*`, `*`sp.pos`*`, SEEK_SET)`, declared in `<cstdio>`.

17.4.8.2.1 `ifstream::ifstream()` [lib.cons.ifstream]

```
ifstream();
```

1 Constructs an object of class `ifstream`, initializing the base class with `istream(&fb)`.

17.4.8.2.2 `ifstream::ifstream(const char*, openmode)` [lib.cons.ifstream.fn]

```
ifstream(const char* s, openmode mode = in);
```

1 Constructs an object of class `ifstream`, initializing the base class with `istream(&fb)`, then calls `open(s, mode)`.

17.4.8.2.3 `ifstream::~ifstream()` [lib.des.ifstream]

```
virtual ~ifstream();
```

1 Destroys an object of class `ifstream`.

17.4.8.2.4 `ifstream::rdbuf()` [lib.ifstream::rdbuf]

```
filebuf* rdbuf() const;
```

1 Returns `(filebuf*)&fb`.

17.4.8.2.5 `ifstream::is_open()` [lib.ifstream::is.open]

```
bool is_open();
```

1 Returns `fb.is_open()`.

17.4.8.2.6 `ifstream::open(const char*, openmode)` [lib.ifstream::open]

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

1 Calls `fb.open(s, mode)`. If the call `is_open()` returns zero, calls `setstate(failbit)`. *

17.4.8.2.7 `ifstream::close()` [lib.ifstream::close]

```
void close();
```

1 Calls `fb.close()` and, if that function returns zero, calls `setstate(failbit)`.

17.4.8.3 Class `ofstream` [lib.ofstream]

```
class ofstream : public ostream {
public:
        ofstream();
        ofstream(const char* s, openmode mode = out);
        virtual ~ofstream();
        filebuf* rdbuf() const;
        bool is_open();
        void open(const char* s, openmode mode = out | trunc);
        void close();
private:
//      filebuf fb;       exposition only
};
```

1 The class `ofstream` is a derivative of `ostream` that assists in the writing of named files. It supplies a `filebuf` object to control the associated sequence. For the sake of exposition, the maintained data is presented here as:

— `filebuf` *fb*, the `filebuf` object.

17.4.8.3.1 `ofstream::ofstream()` [lib.cons.ofstream]

```
ofstream();
```

1 Constructs an object of class `ofstream`, initializing the base class with `ostream(&fb)`.

17.4.8.3.2 `ofstream::ofstream(const char*, openmode)` [lib.cons.ofstream.fn]

```
ofstream(const char* s, openmode mode = out);
```

1 Constructs an object of class `ofstream`, initializing the base class with `ostream(&fb)`, then calls `open(s, mode)`.

17.4.8.3.3 `ofstream::~ofstream()` [lib.des.ofstream]

```
virtual ~ofstream();
```

1 Destroys an object of class `ofstream`.

17.4.8.3.4 `ofstream::rdbuf()` [lib.ofstream::rdbuf]

```
filebuf* rdbuf() const;
```

1 Returns `(filebuf*)&fb`. |

17.4.8.3.5 `ofstream::is_open()` [lib.ofstream::is.open]

```
bool is_open();
```

1 Returns `fb.is_open()`.

17.4.8.3.6 `ofstream::open(const char*, openmode)` [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)`. *

17.4.8.3.7 `ofstream::close()` [lib.ofstream::close]

```
void close();
```

1 Calls `fb.close()` and, if that function returns zero, calls `setstate(failbit)`.

17.4.8.4 Class `stdiobuf` [lib.stdiobuf]

```
class stdiobuf : public streambuf {
public:
        stdiobuf(FILE* file_arg = 0);
        virtual ~stdiobuf();
        bool buffered() const;
        void buffered(bool buf_fl);
protected:
//      virtual int overflow(int c = EOF);       inherited
//      virtual int pbackfail(int c = EOF);      inherited
//      virtual int showmany(); inherited
//      virtual int underflow();          inherited
//      virtual int uflow();    inherited
//      virtual int xsgetn(char* s, int n);      inherited
//      virtual int xsputn(const char* s, int n);        inherited
//      virtual streampos seekoff(streamoff off, ios::seekdir way,
//              ios::openmode which = ios::in | ios::out);      inherited
//      virtual streampos seekpos(streampos sp,
//              ios::openmode which = ios::in | ios::out);      inherited
//      virtual streambuf* setbuf(char* s, int n);       inherited
//      virtual int sync();     inherited
private:
//      FILE* file;      exposition only
//      bool is_buffered;        exposition only
};
```

1 The class `stdiobuf` is derived from `streambuf` to associate both the input sequence and the output sequence with an externally supplied object of type `FILE`. Type `FILE` is defined in `<cstdio>` (17.2). 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 `stdiobuf` object is *buffered,* and hence need not be kept synchronized with the associated file (as described below).

2 The restrictions on reading and writing a sequence controlled by an object of class `stdiobuf` are the same as for an object of class `filebuf`.

3 If an `stdiobuf` object is not buffered and *`file`* is not a null pointer, it is kept synchronized with the associated file, as follows:

— the call `sputc(`*`c`*`)` is equivalent to the call `fputc(`*`c, file`*`);`

— the call `sputbackc(`*`c`*`)` is equivalent to the call `ungetc(`*`c, file`*`);`

— the call `sbumpc()` is equivalent to the call `fgetc(`*`file`*`).`

4 The functions `fgetc(FILE*)`, `fputc(int, FILE*)`, and `ungetc(int, FILE*)` are declared in `<cstdio>` (17.2).

17.4.8.4.1 `stdiobuf::stdiobuf(`*`FILE*`*`)` [lib.cons.stdiobuf.fi]

```
stdiobuf(FILE* file_arg = 0);
```

1 Constructs an object of class `stdiobuf`, initializing the base class with `streambuf()`, and initializing *`file`* to *`file_arg`* and *`is_buffered`* to zero.

17.4.8.4.2 `stdiobuf::~stdiobuf()` [lib.des.stdiobuf]

```
virtual ~stdiobuf();
```

1 Destroys an object of class `stdiobuf`.

17.4.8.4.3 `stdiobuf::buffered()` [lib.stdiobuf::buffered]

```
bool buffered() const;
```

1 Returns a nonzero value if *`is_buffered`* is nonzero.

17.4.8.4.4 `stdiobuf::buffered(bool)` [lib.stdiobuf::buffered.b]

```
void buffered(bool buf_fl);
```

1 Assigns *`buf_fl`* to *`is_buffered`*.

17.4.8.4.5 `stdiobuf::overflow(int)` [lib.stdiobuf::overflow]

```
//      virtual int overflow(int c = EOF);      inherited
```

1 Behaves the same as `filebuf::overflow(int)`, subject to the buffering requirements specified by *`is_buffered`*.

17.4.8.4.6 `stdiobuf::pbackfail(int)` [lib.stdiobuf::pbackfail]

```
//      virtual int pbackfail(int c = EOF);     inherited
```

1 Behaves the same as `filebuf::pbackfail(int)`, subject to the buffering requirements specified by *`is_buffered`*.

17.4.8.4.7 `stdiobuf::showmany()` [lib.stdiobuf::showmany]

```
//      virtual int showmany(); inherited
```

1 Behaves the same as `streambuf::showmany()`.[117]

17.4.8.4.8 `stdiobuf::underflow()` [lib.stdiobuf::underflow]

```
//      virtual int underflow();        inherited
```

1 Behaves the same as `filebuf::underflow()`, subject to the buffering requirements specified by *`is_buffered`*.

17.4.8.4.9 `stdiobuf::uflow()` [lib.stdiobuf::uflow]

```
//      virtual int uflow();    inherited
```

1 Behaves the same as `filebuf::uflow()`, subject to the buffering requirements specified by *`is_buffered`*.

[117] 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.

17.4.8.4.10 `stdiobuf::xsgetn(char*, int)` [lib.stdiobuf::xsgetn]

```
//      virtual int xsgetn(char* s, int n);        inherited
```

1 Behaves the same as `streambuf::xsgetn(char*, int)`.

17.4.8.4.11 `stdiobuf::xsputn(const char*, int)` [lib.stdiobuf::xsputn]

```
//      virtual int xsputn(const char* s, int n);        inherited
```

1 Behaves the same as `streambuf::xsputn(char*, int)`.

17.4.8.4.12 `stdiobuf::seekoff(streamoff, ios::seekdir, ios::openmode)` [lib.stdiobuf::seekoff]

```
//      virtual streampos seekoff(streamoff off, ios::seekdir way,
//              ios::openmode which = ios::in | ios::out);        inherited
```

1 Behaves the same as `filebuf::seekoff(streamoff, ios::seekdir, ios::openmode)`

17.4.8.4.13 `stdiobuf::seekpos(streampos, ios::openmode)` [lib.stdiobuf::seekpos]

```
//      virtual streampos seekpos(streampos sp,
//              ios::openmode which = ios::in | ios::out);        inherited
```

1 Behaves the same as `filebuf::seekpos(streampos, ios::openmode)`

17.4.8.4.14 `stdiobuf::setbuf(char*, int)` [lib.stdiobuf::setbuf]

```
//      virtual streambuf* setbuf(char* s, int n);        inherited
```

1 Behaves the same as `filebuf::setbuf(char*, int)`

17.4.8.4.15 `stdiobuf::sync()` [lib.stdiobuf::sync]

```
//      virtual int sync();        inherited
```

1 Behaves the same as `filebuf::sync()`

17.4.8.5 Class `istdiostream` [lib.istdiostream]

```
class istdiostream : public istream {
public:
        istdiostream(FILE* file_arg = 0);
        virtual ~istdiostream();
        stdiobuf* rdbuf() const;
        bool buffered() const;
        void buffered(bool buf_fl);
private:
//      stdiobuf fb;        exposition only
};
```

1 The class `istdiostream` is a derivative of `istream` that assists in the reading of files controlled by objects of type `FILE`. It supplies a `stdiobuf` object to control the associated sequence. For the sake of exposition, the maintained data is presented here as:

— `stdiobuf` *fb*, the `stdiobuf` object.

17.4.8.5.1 istdiostream::istdiostream(*FILE**) [lib.cons.istdiostream.fi]

```
istdiostream(FILE* file_arg = 0);
```

1 Constructs an object of class `istdiostream`, initializing the base class with `istream(&fb)` and initializing *fb* with `stdiobuf(file_arg)`.

17.4.8.5.2 istdiostream::~istdiostream() [lib.des.istdiostream]

```
virtual ~istdiostream();
```

1 Destroys an object of class `istdiostream`.

17.4.8.5.3 istdiostream::rdbuf() [lib.istdiostream::rdbuf]

```
stdiobuf* rdbuf() const;
```

1 Returns `(stdiobuf*)&fb`.

17.4.8.5.4 istdiostream::buffered() [lib.istdiostream::buffered]

```
bool buffered() const;
```

1 Returns a nonzero value if *is_buffered* is nonzero.

17.4.8.5.5 istdiostream::buffered(bool) [lib.istdiostream::buffered.b]

```
void buffered(bool buf_fl);
```

1 Assigns *buf_fl* to *is_buffered*.

17.4.8.6 Class ostdiostream [lib.ostdiostream]

```
class ostdiostream : public ostream {
public:
        ostdiostream(FILE* file_arg = 0);
        virtual ~ostdiostream();
        stdiobuf* rdbuf() const;
        bool buffered() const;
        void buffered(bool buf_fl);
private:
//      stdiobuf fb;      exposition only
};
```

1 The class `ostdiostream` is a derivative of `ostream` that assists in the writing of files controlled by objects of type `FILE`. It supplies a `stdiobuf` object to control the associated sequence. For the sake of exposition, the maintained data is presented here as:

— `stdiobuf` *fb*, the `stdiobuf` object.

17.4.8.6.1 ostdiostream::ostdiostream(*FILE**) [lib.cons.ostdiostream.fi]

```
ostdiostream(FILE* file_arg = 0);
```

1 Constructs an object of class `ostdiostream`, initializing the base class with `ostream(&fb)` and initializing *fb* with `stdiobuf(file_arg)`.

17.4.8.6.2 `ostdiostream::~ostdiostream()` [lib.des.ostdiostream]

```
virtual ~ostdiostream();
```

1 Destroys an object of class `ostdiostream`.

17.4.8.6.3 `ostdiostream::rdbuf()` [lib.ostdiostream::rdbuf]

```
stdiobuf* rdbuf() const;
```

1 Returns `(stdiobuf*)&`*`fb`*.

17.4.8.6.4 `ostdiostream::buffered()` [lib.ostdiostream::buffered]

```
bool buffered() const;
```

1 Returns a nonzero value if *`is_buffered`* is nonzero.

17.4.8.6.5 `ostdiostream::buffered(bool)` [lib.ostdiostream::buffered.b]

```
void buffered(bool buf_fl);
```

1 Assigns *`buf_fl`* to *`is_buffered`*.

17.4.9 Header `<iostream>` [lib.header.iostream]

1 The header `<iostream>` declares four objects that associate objects of class `stdiobuf` with the standard C streams provided for by the functions declared in `<cstdio>` (17.2). The four objects are constructed, and the associations are established, the first time an object of class `ios::Init` is constructed. The four objects are *not* destroyed during program execution.[118)]

17.4.9.1 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`.

17.4.9.2 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>` (17.2).

17.4.9.3 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>` (17.2).

2 After the object `cerr` is initialized, `cerr.flags() & unitbuf` is nonzero.

118) Constructors and destructors for static objects can access these objects to read input from `stdin` or write output to `stdout` or `stderr`.

17.4.9.4 Object **clog** [lib.clog]

```
extern ostream clog;
```

1 The object `clog` controls output to a stream buffer associated with the object `stderr`, declared in `<cstdio>` (17.2).

17.5 Support classes [lib.support.classes]

1 The Standard C++ library defines several types, and their supporting macros, constants, and function signatures, that support a variety of useful data structures.

17.5.1 Header **<string>** [lib.header.string]

Box 123
Library WG issue: Bjarne Stroustrup, November 10, 1993

The string components should be specified as templates.

1 The header `<string>` defines a type and several function signatures for manipulating varying-length sequences of (single-byte) characters.

17.5.1.1 Class **string** [lib.string]

Box 124
Library WG issue: Uwe Steinmüller, January 21, 1994

For all find operations (searching from the end) `rfind, fins_last_of` and `find_last_not_of` the clause

```
    Returns NPOS if pos > len.
```

should be removed. The functions should (as a convenience) calculate there starting position themselves. If you search forward it is for sure that you cannot find a string if `pos > len`.

... the conditions obtain: `xpos < pos` should be shanged to `xpos <= pos` as this behaviour is consistent with forward searches

```
        string s("1234");
        s.rfind("1", 0) should deliver 0
  and
        s.rfind("4", 3) should be 3
```

If the user wants to use the result for another search he has to decrement himself.

Box 125
Library WG issue: Uwe Steinmüller, January 4, 1994

```
*>M  string& operator=(const string& rsh);
*>M  string& operator=(const char* s);
*>M  string& operator=(char c);
```

Box 126

Library WG issue: Uwe Steinmüller, January 4, 1994

```
*>GENERAL
*>seperate different sections in the header constructors, assign,..

class string {
*>C  char *ptr;   // has this property, might be implemented different
*>C  size_t len; // has this property, might be implemented different
*>C  mutable size_t res; // does not change the string value !!
```

I dislike the approach to have these private members `ptr`, `len`, `res`, because we specify only the public interface. I understand, this only should help to get a better description.

Let me try a different way (a more ADT like approach):

A string can be thought of being a sequence of bytes (this does not imply it to be implemented this way) and has three properties:

len: number of bytes of this sequence

res (res >= len) hint to implementation to keep more byte than len to do some growth in place.

string content: sequence of bytes counted from 0 to len - 1

Now every function can be described to what it does to these properties and nothing is said how these properties are implemented.

Comment (Library WG meeting, San Diego, 3/8/94):

The general concern is that the text describes specifics of what happens to the ‘‘exposition only’’ member data, rather than behavior.

Example:

17.5.1.1.1 describes the action of the default constructor in terms of how the ‘‘exposition only’’ data should be initialized. It doesn’t say whether the string is the null string, an unitialized string of unspecified length, or what...

Recommend:

Generic behaviour should be specified, possibly with the aid of the exposition implementation.

Box 127

Library WG issue: Beman Dawes, December 19, 1993

String/wstring/dynarray/ptrdynarray/bitstring classes are all missing destructor and `operator=`. Bits is missing `operator=`.

Box 128

Library WG issue: Uwe Steinmüller, September 22, 1993

The `dynarray` and my former `string` class proposal followed this rule, we should get a consensus on this by the library WG.

Comment (Library WG meeting, San Diego, 3/8/94):

Uwe wants explicit destructor and (copy) assignment operator. Copy constructor is still there. This should be done for all classes.

Recommend:

All classes should explicitly list the copy constructor, assignment operator, and destructor in their description. But: 17.1.5.6 should state that an implementation can rely on the compiler to actually generate such functions.

```
class string {
public:
        string();
        string(size_t size, capacity cap);
        string(const string& str, size_t pos = 0, size_t n = NPOS);
        string(const char* s, size_t n = NPOS);
        string(char c, size_t rep = 1);
        string(unsigned char c, size_t rep = 1);
        string(signed char c, size_t rep = 1);
        string& operator=(const string& str);
        string& operator=(const char* s);
        string& operator=(char c);
        string& operator+=(const string& rhs);
        string& operator+=(const char* s);
        string& operator+=(char c);
        string& append(const string& str, size_t pos = 0,
                size_t n = NPOS);
        string& append(const char* s, size_t n = NPOS);
        string& append(char c, size_t rep = 1);
        string& assign(const string& str, size_t pos = 0,
                size_t n = NPOS);
        string& assign(const char* s, size_t n = NPOS);
        string& assign(char c, size_t rep = 1);
        string& insert(size_t pos1, const string& str, size_t pos2 = 0,
                size_t n = NPOS);
        string& insert(size_t pos, const char* s,
                size_t n = NPOS);
        string& insert(size_t pos, char c, size_t rep = 1);
        string& remove(size_t pos = 0, size_t n = NPOS);
        string& replace(size_t pos1, size_t n1, const string& str,
                size_t pos2 = 0, size_t n2 = NPOS);
        string& replace(size_t pos, size_t n1, const char* s,
                size_t n2 = NPOS);
        string& replace(size_t pos, size_t n, char c,
                size_t rep = 1);
        char get_at(size_t pos) const;
        void put_at(size_t pos, char c);
        char operator[](size_t pos) const;
        char& operator[](size_t pos);
        const char* data() const;
        size_t length() const:
        void resize(size_t n, char c = 0);
        size_t reserve() const;
        void reserve(size_t res_arg);
        size_t copy(char* s, size_t n, size_t pos = 0);
        size_t find(const string& str, size_t pos = 0) const;
        size_t find(const char* s, size_t pos = 0, size_t n = NPOS) const;
        size_t find(char c, size_t pos = 0) const;
        size_t rfind(const string& str, size_t pos = NPOS) const;
        size_t rfind(const char* s, size_t pos = NPOS,
                size_t n = NPOS) const;
        size_t rfind(char c, size_t pos = NPOS) const;
        size_t find_first_of(const string& str, size_t pos = 0) const;
        size_t find_first_of(const char* s, size_t pos = 0,
                size_t n = NPOS) const;
        size_t find_first_of(char c, size_t pos = 0) const;
        size_t find_last_of(const string& str, size_t pos = NPOS) const;
        size_t find_last_of(const char* s, size_t pos = NPOS,
                size_t n = NPOS) const;
        size_t find_last_of(char c, size_t pos = NPOS) const;
        size_t find_first_not_of(const string& str, size_t pos = 0) const;
        size_t find_first_not_of(const char* s, size_t pos = 0,
```

```
                size_t n = NPOS) const;
        size_t find_first_not_of(char c, size_t pos = 0) const;
        size_t find_last_not_of(const string& str, size_t pos = NPOS)
                const;
        size_t find_last_not_of(const char* s, size_t pos = NPOS,
                size_t n = NPOS) const;
        size_t find_last_not_of(char c, size_t pos = NPOS) const;
        string substr(size_t pos = 0, size_t n = NPOS) const;
        int compare(const string& str, size_t pos = 0,
                size_t n = NPOS) const;
        int compare(const char* s, size_t pos = 0, size_t n = NPOS) const;
        int compare(char c, size_t pos = 0, size_t rep = 1) const;
private:
//      char* ptr;          exposition only
//      size_t len, res;          exposition only
};
```

1 The class `string` describes objects that can store a sequence consisting of a varying number of arbitrary (single-byte) characters. The first element of the sequence is at position zero. Such a sequence is also called a *character 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 `string`. For the sake of exposition, the maintained data is presented here as:

— `char*` *`ptr`*, points to the initial character of the string;

— `size_t` *`len`*, counts the number of characters currently iU the string;

— `size_t` *`res`*, for an unallocated string, holds the recommended allocation size of the string, while for an allocated string, becomes the currently allocated size.

2 In all cases, *`len`* `<=` *`res`*.

3 The functions described in this subclause 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`.

4 To report one of these errors, the function evaluates the expression *`ex`*`.raise()`, where *`ex`* is an object of the associated exception type.

17.5.1.1.1 `string::string()` [lib.cons.string]

```
string();
```

1 Constructs an object of class `string` initializing:

— *`ptr`* to an unspecified value;

— *`len`* to zero;

— *`res`* to an unspecified value.

17.5.1.1.2 `string::string(size_t, capacity)` [lib.cons.string.cap]

```
string(size_t size, capacity cap);
```

1 Constructs an object of class `string`. If *cap* is *default_size*, the function either reports a length error if *size* equals `NPOS` or initializes:

— *ptr* to point at the first element of an allocated array of *size* elements, each of which is initialized to zero;

— *len* to *size*;

— *res* to a value at least as large as *len*.

2 Otherwise, *cap* shall be *reserve* and the function initializes:

— *ptr* to an unspecified value;

— *len* to zero;

— *res* to *size*.

17.5.1.1.3 `string::string(const string&, size_t, size_t)` [lib.cons.string.sub]

```
string(const string& str, size_t pos = 0, size_t n = NPOS);
```

1 Reports an out-of-range error if `pos > str.len`. Otherwise, the function constructs an object of class `string` and determines the effective length *rlen* of the initial string value as the smaller of *n* and `str.len - pos`. Thus, the function 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*;

— *res* to a value at least as large as *len*.

17.5.1.1.4 `string::string(const char*, size_t)` [lib.cons.string.str]

```
string(const char* s, size_t n = NPOS);
```

1 If *n* equals `NPOS`, stores `strlen(s)` in *n*. The function signature `strlen(const char*)` is declared in `<cstring>` (17.2).

2 In any case, the function constructs an object of class `string` and determines its initial string value from the array of `char` of length *n* whose first element is designated by *s*. *s* shall not be a null pointer. Thus, the function initializes:

— *ptr* to point at the first element of an allocated copy of the array whose first element is pointed at by *s*;

— *len* to *n*;

— *res* to a value at least as large as *len*.

17.5.1.1.5 `string::string(char, size_t)` [lib.cons.string.c]

```
string(char c, size_t rep = 1);
```

1 Reports a length error if *rep* equals `NPOS`. Otherwise, the function constructs an object of class `string` and determines its initial string value by repeating the character *c* for all *rep* elements. Thus, the function initializes:

— *ptr* to point at the first element of an allocated array of *rep* elements, each storing the initial value *c*;

— *len* to *rep*;

— *res* to a value at least as large as *len*.

17.5.1.1.6 `string::string(unsigned char, size_t)` [lib.cons.string.uc]

```
string(unsigned char c, size_t rep = 1);
```

1 Behaves the same as `string((char)c, rep)`.

17.5.1.1.7 `string::string(signed char, size_t)` [lib.cons.string.sc]

```
string(signed char c, size_t rep = 1);
```

1 Behaves the same as `string((char)c, rep)`.

17.5.1.1.8 `string::operator=(const string&)` [lib.string::op=.sub]

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

1 Returns `assign(str)`.

17.5.1.1.9 `string::operator=(const char*)` [lib.string::op=.str]

```
string& operator=(const char* s);
```

1 Returns `*this = string(s)`.

17.5.1.1.10 `string::operator=(char)` [lib.string::op=.c]

```
string& operator=(char c);
```

1 Returns `*this = string(c)`.

17.5.1.1.11 `string::operator+=(const string&)` [lib.string::op+=.sub]

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

1 Returns `append(rhs)`.

17.5.1.1.12 `string::operator+=(const char*)` [lib.string::op+=.str]

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

1 Returns `*this += string(s)`.

17.5.1.1.13 `string::operator+=(char)` [lib.string::op+=.c]

```
string& operator+=(char c);
```

1 Returns `*this += string(c)`.

17.5.1.1.14 `string::append(const string&, size_t, size_t)` [lib.string::append.sub]

```
string& append(const string& str, size_t pos = 0, size_t n = NPOS);
```

1 Reports an out-of-range error 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 reports a 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 The function returns `*this`.

17.5.1.1.15 `string::append(const char*, size_t)` [lib.string::append.str]

```
string& append(const char* s, size_t n = NPOS);
```
*

1 Returns `append(string(s, n))`.

17.5.1.1.16 `string::append(char, size_t)` [lib.string::append.c]

```
string& append(char c, size_t rep = 1);
```
*

1 Returns `append(string(c, rep))`.

17.5.1.1.17 `string::assign(const string&, size_t, size_t)` [lib.string::assign.sub]

```
string& assign(const string& str, size_t pos = 0, size_t n = NPOS);
```

1 Reports an out-of-range error 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 The function returns `*this`.

17.5.1.1.18 `string::assign(const char*, size_t)` [lib.string::assign.str]

```
string& assign(const char* s, size_t n = NPOS);
```
*

1 Returns `assign(string(s, n))`.

17.5.1.1.19 `string::assign(char, size_t)` [lib.string::assign.c]

```
string& assign(char c, size_t rep = 1);
```

1 Returns `assign(string(c, rep))`.

17.5.1.1.20 `string::insert(size_t, const string&, size_t, size_t)` [lib.string::insert.sub]

```
string& insert(size_t pos1, const string& str, size_t pos2 = 0,
        size_t n = NPOS);
```

1 Reports an out-of-range error 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*. The function then reports a 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 The function returns `*this`.

17.5.1.1.21 string::insert(size_t, const char*, size_t) [lib.string::insert.str]

```
string& insert(size_t pos, const char* s, size_t n = NPOS);
```
*

1 Returns `insert(pos, string(s, n))`.

17.5.1.1.22 string::insert(size_t, char, size_t) [lib.string::insert.c]

```
string& insert(size_t pos, char c, size_t rep = 1);
```

1 Returns `insert(pos, string(c, rep))`.

17.5.1.1.23 string::remove(size_t, size_t) [lib.string::remove]

```
string& remove(size_t pos = 0, size_t n = NPOS);
```
*

1 Reports an out-of-range error 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 The function returns `*this`.

17.5.1.1.24 string::replace(size_t, size_t, const string&, size_t, size_t) [lib.string::replace.sub]

```
string& replace(size_t pos1, size_t n1, const string& str,
        size_t pos2 = 0, size_t n2 = NPOS);
```

1 Reports an out-of-range error 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*. The function then reports a 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 The function returns `*this`.

17.5.1.1.25 string::replace(size_t, size_t, const char*, size_t) [lib.string::replace.str]

```
string& replace(size_t pos, size_t n1, const char* s,
        size_t n2 = NPOS);
```

1 Returns `replace(pos, n1, string(s, n2))`.

17.5.1.1.26 string::replace(size_t, size_t, char, size_t) [lib.string::replace.c]

```
string& replace(size_t pos, size_t n, char c, size_t rep = 1);
```

1 Returns `replace(pos, n, string(c, rep))`.

17.5.1.1.27 `string::get_at(size_t)` [lib.string::get.at]

Box 129

Library WG issue: Uwe Steinmüller, January 4, 1994

```
*>C  const char get_at(size_t pos) const;
```

Comment (Library WG meeting, San Diego, 3/8/94):

Should member functions that return a value return a `const` value? This issue arises by the decision in San Jose no to return `const` from return values.

Recommend:

All value return types should be returned as a `const` value.

```
char get_at(size_t pos) const;
```

1 Reports an out-of-range error if *pos* >= *len*. Otherwise, the function returns *ptr*[*pos*].

17.5.1.1.28 `string::put_at(size_t, char)` [lib.string::put.at]

```
void put_at(size_t pos, char c);
```

1 Reports an out-of-range error if *pos* >= *len*. Otherwise, the function assigns *c* to *ptr*[*pos*].

17.5.1.1.29 `string::operator[](size_t)` [lib.string::op.array]

Box 130

Library WG issue: Uwe Steinmüller, January 4, 1994

```
*>C const char operator[](size_t pos) const;
        char& operator[](size_t pos);
```

Box 131

Library WG issue: Uwe Steinmüller, January 4, 1994

```
*>C  const char operator[](size_t pos) const;
```

```
char operator[](size_t pos) const;
char& 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 a subsequent call to any member function for the object.

17.5.1.1.30 `string::data()` [lib.string::data]

```
const char* data() 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. 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 `string` that designates the same object as `*this`.

17.5.1.1.31 `string::length()` [lib.string::length]

```
size_t length() const:
```

1 Returns *len*.

17.5.1.1.32 `string::resize(size_t, char)` [lib.string::resize]

```
void resize(size_t n, char c = 0);
```

1 Reports a length error if *n* equals `NPOS`. Otherwise, the function alters the length of the string designated by `*this` as follows:

— 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`.

— 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*.

17.5.1.1.33 `string::reserve()` [lib.string::reserve]

Box 132

Library WG issue: Uwe Steinmüller, January 4, 1994

```
    size_t reserve() const;

*>C  void reserve(size_t res_arg) const;  //res is mutable
```

Comment (Library WG meeting, San Diego, 3/8/94):

Should this be a `const` member function?

Reccomend:

It should not be a `const` member function. (If it were a `const` member function an implementation would have to use a `mutable` member data which would then not be ROMable). However, the description should state that the string should be semantically const.

```
size_t reserve() const;
```

1 Returns *res*.

17.5.1.1.34 `string::reserve(size_t)` [lib.string::reserve.cap]

```
void reserve(size_t res_arg);
```

1 If no string is allocated, the function assigns *res_arg* to *res*. Otherwise, whether or how the function alters *res* is unspecified.

17.5.1.1.35 `string::copy(char*, size_t, size_t)` [lib.string::copy]

```
size_t copy(char* s, size_t n, size_t pos = 0);
```

1 Reports an out-of-range error 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*.[119)]

3 The function returns *rlen*.

17.5.1.1.36 `string::find(const string&, size_t)` [lib.string::find.sub]

```
size_t find(const string& 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`;

— `ptr[xpos + I] == str.ptr[I]` for all elements *I* of the string controlled by *str*.

2 If the function can determine such a value for *xpos*, it returns *xpos*. Otherwise, it returns `NPOS`.

17.5.1.1.37 `string::find(const char*, size_t, size_t)` [lib.string::find.str]

```
size_t find(const char* s, size_t pos = 0, size_t n = NPOS) const;
```

1 Returns `find(string(s, n), pos)`.

17.5.1.1.38 `string::find(char, size_t)` [lib.string::find.c]

```
size_t find(char c, size_t pos = 0) const;
```

1 Returns `find(string(c), pos)`.

17.5.1.1.39 `string::rfind(const string&, size_t)` [lib.string::rfind.sub]

```
size_t rfind(const string& 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`;

— `ptr[xpos + I] == str.ptr[I]` for all elements *I* of the string controlled by *str*.

2 If the function can determine such a value for *xpos*, it returns *xpos*. Otherwise, it returns `NPOS`.

119) The function does not append a null character to the string.

17.5.1.1.40 string::rfind(const char*, size_t, size_t) [lib.string::rfind.str]

```
size_t rfind(const char* s, size_t pos = NPOS,
        size_t n = NPOS) const;
```
*

1 Returns `rfind(string(s, n), pos)`.

17.5.1.1.41 string::rfind(char, size_t) [lib.string::rfind.c]

```
size_t rfind(char c, size_t pos = NPOS) const;
```

1 Returns `rfind(string(c, n), pos)`.

17.5.1.1.42 string::find_first_of(const string&, size_t) [lib.string::find.first.of.sub]

```
size_t find_first_of(const string& 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*;

— *ptr*`[`*xpos*`] ==` *str*`.`*ptr*`[`*I*`]` for some element *I* of the string controlled by *str*.

2 If the function can determine such a value for *xpos*, it returns *xpos*. Otherwise, it returns `NPOS`. |

17.5.1.1.43 string::find_first_of(const char*, size_t, size_t) [lib.string::find.first.of.str]

```
size_t find_first_of(const char* s, size_t pos = 0,
        size_t n = NPOS) const;
```

1 Returns `find_first_of(string(s, n), pos)`.

17.5.1.1.44 string::find_first_of(char, size_t) [lib.string::find.first.of.c]

```
size_t find_first_of(char c, size_t pos = 0) const;
```

1 Returns `find_first_of(string(c), pos)`.

17.5.1.1.45 string::find_last_of(const string&, size_t) [lib.string::find.last.of.sub]

```
size_t find_last_of(const string& 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*`;`

— *ptr*`[`*xpos*`] ==` *str*`.`*ptr*`[`*I*`]` for some element *I* of the string controlled by *str*.

2 If the function can determine such a value for *xpos*, it returns *xpos*. Otherwise, it returns `NPOS`. |

17.5.1.1.46 string::find_last_of(const char*, size_t, size_t) [lib.string::find.last.of.str]

```
size_t find_last_of(const char* s, size_t pos = NPOS,
        size_t n = NPOS) const;
```

1 Returns `find_last_of(string(s, n), pos)`.

17.5.1.1.47 string::find_last_of(char, size_t) [lib.string::find.last.of.c]

```
size_t find_last_of(char c, size_t pos = NPOS) const;
```

1 Returns `find_last_of(string(c, n), pos)`.

17.5.1.1.48 string::find_first_not_of(const string&, size_t) [lib.string::find.first.not.of.sub]

```
size_t find_first_not_of(const string& 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*;

— *ptr*`[`*xpos*`] ==` *str*`.`*ptr*`[`*I*`]` for no element *I* of the string controlled by *str*.

2 If the function can determine such a value for *xpos*, it returns *xpos*. Otherwise, it returns `NPOS`.

17.5.1.1.49 string::find_first_not_of(const char*, size_t, size_t) [lib.string::find.first.not.of.str]

```
size_t find_first_not_of(const char* s, size_t pos = 0,
        size_t n = NPOS) const;
```

1 Returns `find_first_not_of(string(s, n), pos)`.

17.5.1.1.50 string::find_first_not_of(char, size_t) [lib.string::find.first.not.of.c]

```
size_t find_first_not_of(char c, size_t pos = 0) const;
```

1 Returns `find_first_not_of(string(c), pos)`.

17.5.1.1.51 string::find_last_not_of(const string&, size_t) [lib.string::find.last.not.of.sub]

```
size_t find_last_not_of(const string& 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*;

— *ptr*`[`*xpos*`] ==` *str*`.`*ptr*`[`*I*`]` for no element *I* of the string controlled by *str*.

2 If the function can determine such a value for *xpos*, it returns *xpos*. Otherwise, it returns `NPOS`.

17.5.1.1.52 string::find_last_not_of(const char*, size_t, size_t) [lib.string::find.last.not.of.str]

```
size_t find_last_not_of(const char* s, size_t pos = NPOS,
        size_t n = NPOS) const;
```
*

1 Returns `find_last_not_of(string(s, n), pos)`.

17.5.1.1.53 `string::find_last_not_of(char, size_t)` [lib.string::find.last.not.of.c]

```
size_t find_last_not_of(char c, size_t pos = NPOS) const;
```

1 Returns `find_last_not_of(string(c, n), pos)`.

17.5.1.1.54 `string::substr(size_t, size_t)` [lib.string::substr]

```
string substr(size_t pos = 0, size_t n = NPOS) const;
```

1 Returns `string(*this, pos, n)`.

17.5.1.1.55 `string::compare(const string&, size_t, size_t)` [lib.string::compare.sub]

```
int compare(const string& str, size_t pos = 0, size_t n = NPOS) const;
```

1 Reports an out-of-range error if `pos > len`. Otherwise, if `str.len < n`, the function stores `str.len` in `n`. The function then determines the effective length `rlen` of the strings to compare as the smaller of `n` and `len - pos`. The function then compares the two strings by calling `memcmp(ptr + pos, str.ptr, rlen)`. The function signature `memcmp(const void*, const void*, size_t)` is declared in `<cstring>` (<$RS*,[lib.standard.c.library]xxx>).[120]

2 If the result of that comparison is nonzero, the function returns the nonzero result. Otherwise, the function returns:

— if `len - pos < n`, a value less than zero;

— if `len - pos == n`, the value zero;

— if `len - pos > n`, a value greater than zero.

17.5.1.1.56 `string::compare(const char*, size_t, size_t)` [lib.string::compare.str]

```
size_t compare(const char* s, size_t pos = 0, size_t n = NPOS) const;
```

1 Returns `compare(string(s, n), pos)`.

17.5.1.1.57 `string::compare(char, size_t, size_t)` [lib.string::compare.c]

```
size_t compare(char c, size_t pos = 0, size_t rep = 1) const;
```

1 Returns `compare(string(c, rep), pos)`.

17.5.1.2 `operator+(const string&, const string&)` [lib.op+.sub.sub]

```
string operator+(const string& lhs, const string& rhs);
```

1 Returns `string(lhs).append(rhs)`.

17.5.1.3 `operator+(const char*, const string&)` [lib.op+.str.sub]

```
string operator+(const char* lhs, const string& rhs);
```

[120] The elements are compared as if they had type `unsigned char`.

1 Returns `string(lhs) + rhs`.

17.5.1.4 operator+(char, const string&) [lib.op+.c.sub]

```
string operator+(char lhs, const string& rhs);
```

1 Returns `string(lhs) + rhs`.

17.5.1.5 operator+(const string&, const char*) [lib.op+.sub.str]

```
string operator+(const string& lhs, const char* rhs);
```

1 Returns `lhs + string(rhs)`.

17.5.1.6 operator+(const string&, char) [lib.op+.str.c]

```
string operator+(const string& lhs, char rhs);
```

1 Returns `lhs + string(rhs)`.

17.5.1.7 operator==(const string&, const string&) [lib.op==.sub.sub]

```
bool operator==(const string& lhs, const string& rhs);
```

1 Returns a nonzero value if `!(lhs == rhs)` is nonzero.

17.5.1.8 operator==(const char*, const string&) [lib.op==.str.sub]

```
bool operator==(const char* lhs, const string& rhs);
```

1 Returns `string(lhs) == rhs`.

17.5.1.9 operator==(char, const string&) [lib.op==.c.sub]

```
bool operator==(char lhs, const string& rhs);
```

1 Returns `string(lhs) == rhs`.

17.5.1.10 operator==(const string&, const char*) [lib.op==.sub.str]

```
bool operator==(const string& lhs, const char* rhs);
```

1 Returns `lhs == string(rhs)`.

17.5.1.11 operator==(const string&, char) [lib.op==.sub.c]

```
bool operator==(const string& lhs, char rhs);
```

1 Returns `lhs == string(rhs)`.

17.5.1.12 operator!=(const string&, const string&) [lib.op!=.sub.sub]

```
bool operator!=(const string& lhs, const string& rhs);
```

1 Returns a nonzero value if `lhs.compare(rhs)` is nonzero.

17.5.1.13 `operator!=(const char*, const string&)` [lib.op!=.str.sub]

```
bool operator!=(const char* lhs, const string& rhs);
```

1 Returns `string(lhs) != rhs`.

17.5.1.14 `operator!=(char, const string&)` [lib.op!=.c.sub]

```
bool operator!=(char lhs, const string& rhs);
```

1 Returns `string(lhs) != rhs`.

17.5.1.15 `operator!=(const string&, const char*)` [lib.op!=.sub.str]

```
bool operator!=(const string& lhs, const char* rhs);
```

1 Returns `lhs != string(rhs)`.

17.5.1.16 `operator!=(const string&, char)` [lib.op!=.sub.c]

```
bool operator!=(const string& lhs, char rhs);
```

1 Returns `lhs != string(rhs)`.

17.5.1.17 `operator>>(istream&, string&)` [lib.ext.sub]

```
istream& operator>>(istream& is, string& str);
```

1 A formatted input function, extracts characters and appends them to the string controlled by *str*. The string is initially made empty by calling `str.remove()`. Each extracted character *c* is appended as if by calling `str.append(c)`. If `width()` is greater than zero, the maximum number of characters stored *n* is `width()`; otherwise it is `INT_MAX`, defined in `<climits>` (17.2).

2 Characters are extracted and appended until any of the following occurs:

— *n* characters are appended;

— `NPOS - 1` characters are appended;

— end-of-file occurs on the input sequence;

— `isspace(c)` is nonzero for the next available input character *c* (in which case the input character is not extracted).

3 The function signature `isspace(int)` is declared in `<cctype>`.

4 If the function appends no characters, it calls `setstate(failbit)`. In any case, it calls `width(0)`. The function returns *is*.

17.5.1.18 `getline(istream&, string&, char)` [lib.getline.sub]

```
istream& getline(istream& is, string& str, char delim = '\n');
```

1 An unformatted input function, extracts characters and appends them to the string controlled by *str*. The string is initially made empty by calling `str.remove()`. Each extracted character *c* is appended as if by calling `str.append(c)`. Characters are extracted and appended until any of the following occurs:

— `NPOS - 1` characters are appended (in which case the function calls `setstate(failbit)`);

— 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 the input character is extracted but not appended).

2 If the function appends no characters, it calls `setstate(failbit)`. The function returns *is*.

17.5.1.19 `operator<<(ostream&, const string&)` [lib.ins.sub]

```
ostream& operator<<(ostream& os, const string& str);
```

1 A formatted output function, behaves the same as `os.write(str.data(), str.length())`.

2 The function returns *os*.

17.5.2 Header `<wstring>` [lib.header.wstring]

1 The header `<wstring>` defines a type and several function signatures for manipulating varying-length sequences of wide characters.

17.5.2.1 Class `wstring` [lib.wstring]

Box 133

Library WG issue: Ichiro Koshida, January 10, 1994

`Wstring` class lacks I/O functions:

```
   istream& operator>>( istream&, wstring& )
   istream& getline( istream&, wstring&, wchar_t )
  ostream& operator<<( ostream&, wstring& )
```

*

```
class wstring {
public:
        wstring();
        wstring(size_t size, capacity cap);
        wstring(const wstring& str, size_t pos = 0, size_t n = NPOS);
        wstring(const wchar_t* s, size_t n = NPOS);
        wstring(wchar_t c, size_t rep = 1);
        wstring& operator=(const wstring& str);
        wstring& operator=(const wchar_t* s);
        wstring& operator=(wchar_t c);
        wstring& operator+=(const wstring& rhs);
        wstring& operator+=(const wchar_t* s);
        wstring& operator+=(wchar_t c);
        wstring& append(const wstring& str, size_t pos = 0,
                size_t n = NPOS);
        wstring& append(const wchar_t* s, size_t n = NPOS);
        wstring& append(wchar_t c, size_t rep = 1);
        wstring& assign(const wstring& str, size_t pos = 0,
                size_t n = NPOS);
        wstring& assign(const wchar_t*  s, size_t n = NPOS);
        wstring& assign(wchar_t c, size_t rep = 1);
        wstring& insert(size_t pos1, const wstring& str, size_t pos2 = 0,
                size_t n = NPOS);
        wstring& insert(size_t pos, const wchar_t* s,
                size_t n = NPOS);
        wstring& insert(size_t pos, wchar_t c, size_t rep = 1);
        wstring& remove(size_t pos = 0, size_t n = NPOS);
        wstring& replace(size_t pos1, size_t n1, const wstring& str,
                size_t pos2 = 0, size_t n2 = NPOS);
        wstring& replace(size_t pos, size_t n1, const wchar_t* s,
                size_t n2 = NPOS);
        wstring& replace(size_t pos, size_t n, wchar_t c,
                size_t rep = 1);
        wchar_t get_at(size_t pos) const;
        void put_at(size_t pos, wchar_t c);
        wchar_t operator[](size_t pos) const;
        wchar_t& operator[](size_t pos);
        const wchar_t* data() const;
        size_t length() const:
        void resize(size_t n, wchar_t c = 0);
        size_t reserve() const;
        void reserve(size_t res_arg);
        size_t copy(wchar_t* s, size_t n, size_t pos = 0);
        size_t find(const wstring& str, size_t pos = 0) const;
        size_t find(const wchar_t* s, size_t pos = 0, size_t n = NPOS)
                const;
        size_t find(wchar_t c, size_t pos = 0) const;
        size_t rfind(const wstring& str, size_t pos = NPOS) const;
        size_t rfind(const wchar_t* s, size_t pos = NPOS,
                size_t n = NPOS) const;
        size_t rfind(wchar_t c, size_t pos = NPOS) const;
        size_t find_first_of(const wstring& str, size_t pos = 0) const;
        size_t find_first_of(const wchar_t* s, size_t pos = 0,
                size_t n = NPOS) const;
        size_t find_first_of(wchar_t c, size_t pos = 0) const;
        size_t find_last_of(const wstring& str, size_t pos = NPOS) const;
        size_t find_last_of(const wchar_t* s, size_t pos = NPOS,
                size_t n = NPOS) const;
        size_t find_last_of(wchar_t c, size_t pos = NPOS) const;
        size_t find_first_not_of(const wstring& str, size_t pos = 0)
                const;
        size_t find_first_not_of(const wchar_t* s, size_t pos = 0,
```

```
                size_t n = NPOS) const;
        size_t find_first_not_of(wchar_t c, size_t pos = 0) const;
        size_t find_last_not_of(const wstring& str, size_t pos = NPOS)
                const;
        size_t find_last_not_of(const wchar_t* s, size_t pos = NPOS,
                size_t n = NPOS) const;
        size_t find_last_not_of(wchar_t c, size_t pos = NPOS) const;
        wstring substr(size_t pos = 0, size_t n = NPOS) const;
        int compare(const wstring& str, size_t pos = 0,
                size_t n = NPOS) const;
        int compare(const wchar_t* s, size_t pos = 0, size_t n = NPOS) const;
        int compare(wchar_t c, size_t pos = 0, size_t rep = 1) const;
private:
//      wchar_t* ptr;     exposition only
//      size_t len, res;          exposition only
};
```

1 The class `wstring` describes objects that can store a sequence consisting of a varying number of arbitrary wide characters. The first element of the sequence is at position zero. Such a sequence is also called a *wide-character 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 `wstring`. For the sake of exposition, the maintained data is presented here as:

— `wchar_t* ptr`, points to the initial character of the string;

— `size_t len`, counts the number of characters currently in the string;

— `size_t res`, for an unallocated string, holds the recommended allocation size of the string, while for an allocated string, becomes the currently allocated size.

2 In all cases, `len <= res`.

3 The functions described in this subclause 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`.

4 To report one of these errors, the function evaluates the expression `ex.raise()`, where `ex` is an object of the associated exception type.

17.5.2.1.1 `wstring::wstring()` [lib.cons.wstring]

```
wstring();
```

1 Constructs an object of class `wstring` initializing:

— `ptr` to an unspecified value;

— `len` to zero;

— `res` to an unspecified value.

17.5.2.1.2 `wstring::wstring(size_t, capacity)` [lib.cons.wstring.cap]

```
wstring(size_t size, capacity cap);
```

wstring::wstring(size_t, capacity)

1 Constructs an object of class `wstring`. If *cap* is *default_size*, the function either reports a length error if *size* equals `NPOS` or initializes:

— *ptr* to point at the first element of an allocated array of *size* elements, each of which is initialized to zero;

— *len* to *size*;

— *res* to a value at least as large as *len*.

2 Otherwise, *cap* shall be *reserve* and the function initializes:

— *ptr* to an unspecified value;

— *len* to zero;

— *res* to *size*.

17.5.2.1.3 wstring::wstring(const wstring&, size_t, size_t) [lib.cons.wstring.wsub]

```
wstring(const wstring& str, size_t pos = 0, size_t n = NPOS);
```

1 Reports an out-of-range error if *pos > str.len*. Otherwise, the function constructs an object of class `wstring` and determines the effective length *rlen* of the initial wstring value as the smaller of *n* and *str.len - pos*. Thus, the function initializes:

— *ptr* to point at the first element of an allocated copy of *rlen* elements of the wstring controlled by *str* beginning at position *pos*;

— *len* to *rlen*;

— *res* to a value at least as large as *len*.

17.5.2.1.4 wstring::wstring(const wchar_t*, size_t) [lib..cons.wstring.wstr]

```
wstring(const wchar_t* s, size_t n);
```

1 If *n* equals `NPOS`, stores `wcslen(s)` in *n*. The function signature `wcslen(const wchar_T*)` is declared in `<cwchar>` (17.2).

2 In any case, the function constructs an object of class `wstring` and determines its initial string value from the array of `wchar_t` of length *n* whose first element is designated by *s*. *s* shall not be a null pointer. hhus, the function initializes:

— *ptr* to point at the first element of an allocated copy of the array whose first element is pointed at by *s*;

— *len* to *n*;

— *res* to a value at least as large as *len*.

17.5.2.1.5 wstring::wstring(wchar_t, size_t) [lib..cons.wstring.wc]

```
wstring(wchar_t c, size_t rep = 1);
```

1 Reports a length error if *rep* equals `NPOS`. Otherwise, the function constructs an object of class `wstring` and determines its initial string value by repeating the character *c* for all *rep* elements. Thus, the function initializes:

— *ptr* to point at the first element of an allocated array of *rep* elements, each storing the initial value *c*;

— *len* to *rep*;

— *res* to a value at least as large as *len*.

17.5.2.1.6 `wstring::operator=(const wchar_t*)` [lib.wstring::op=.sub]

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

1 Returns `assign(str)`.

17.5.2.1.7 `wstring::operator=(const wchar_t*)` [lib.wstring::op=.wstr]

```
wstring& operator=(const wchar_t* s);
```

1 Returns `*this = string(s)`.

17.5.2.1.8 `wstring::operator=(wchar_t)` [lib.wstring::op=.wc]

```
wstring& operator=(wchar_t c);
```

1 Returns `*this = string(c)`.

17.5.2.1.9 `wstring::operator+=(const wstring&)` [lib.wstring::op+=.wsub]

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

1 Returns `append(rhs)`.

17.5.2.1.10 `wstring::operator+=(const wchar_t*)` [lib.wstring::op+=.wstr]

```
wstring& operator+=(const wchar_t* s);
```

1 Returns `*this += string(s)`.

17.5.2.1.11 `wstring::operator+=(wchar_t)` [lib.wstring::op+=.wc]

```
wstring& operator+=(wchar_t c);
```

1 Returns `*this += string(c)`.

17.5.2.1.12 `wstring::append(const wstring&, size_t, size_t)` [lib.wstring::append.wsub]

```
wstring& append(const wstring& str, size_t pos = 0, size_t n = NPOS);
```

1 Reports an out-of-range error 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 reports a 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 The function returns `*this`.

17.5.2.1.13 wstring::append(const wchar_t*, size_t) [lib.wstring::append.wstr]

```
wstring& append(const wchar_t* s, size_t n = NPOS);
```

1 Returns `append(wstring(s, n))`.

17.5.2.1.14 wstring::append(wchar_t, size_t) [lib.wstring::append.wc]

```
wstring& append(wchar_t c, size_t rep = 1);
```

1 Returns `append(wstring(c, rep))`.

17.5.2.1.15 wstring::assign(const wstring&, size_t, size_t) [lib.wstring::assign.wsub]

```
wstring& assign(const wstring& str, size_t pos = 0, size_t n = NPOS);
```

1 Reports an out-of-range error 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 The function returns `*this`.

17.5.2.1.16 wstring::assign(const wchar_t*, size_t) [lib.wstring::assign.wstr]

```
wstring& assign(const wchar_t* s, size_t n = NPOS);
```

1 Returns `assign(wstring(s, n))`.

17.5.2.1.17 wstring::assign(wchar_t, size_t) [lib.wstring::assign.wc]

```
wstring& assign(wchar_t c, size_t rep = 1);
```

1 Returns `assign(wstring(c, rep))`.

17.5.2.1.18 wstring::insert(size_t, const wstring&, size_t, size_t) [lib.wstring::insert.wsub]

```
wstring& insert(size_t pos1, const wstring& str, size_t pos2 = 0,
        size_t n = NPOS);
```

1 Reports an out-of-range error 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*. The function then reports a 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 The function returns `*this`.

17.5.2.1.19 `wstring::insert(size_t, const wchar_t*, size_t)` [lib.wstring::insert.wstr]

```
wstring& insert(size_t pos, const wchar_t* s, size_t n = NPOS);
```

1 Returns `insert(pos, wstring(s, n))`.

17.5.2.1.20 `wstring::insert(size_t, wchar_t, size_t)` [lib.wstring::insert.wc]

```
wstring& insert(size_t pos, wchar_t c, size_t rep = 1);
```

1 Returns `insert(pos, wstring(c, rep))`.

17.5.2.1.21 `wstring::remove(size_t, size_t)` [lib.wstring::remove]

```
wstring& remove(size_t pos = 0, size_t n = NPOS);
```

1 Reports an out-of-range error 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 The function returns `*this`.

17.5.2.1.22 `wstring::replace(size_t, size_t, const wstring&, size_t, size_t)` [lib.wstring::replace.wsub]

```
wstring& replace(size_t pos1, size_t n1, const wstring& str,
        size_t pos2 = 0, size_t n2 = NPOS);
```

1 Reports an out-of-range error 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`. The function then reports a 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 The function returns `*this`.

17.5.2.1.23 `wstring::replace(size_t, size_t, const wchar_t*, size_t)` [lib.wstring::replace.wstr]

```
wstring& replace(size_t pos, size_t n1, const wchar_t* s,
        size_t n2 = NPOS);
```

1 Returns `replace(pos, n1, wstring(s, n2))`.

17.5.2.1.24 `wstring::replace(size_t, size_t, wchar_t, size_t)` [lib.wstring::replace.wc]

```
wstring& replace(size_t pos, size_t n, wchar_t c, size_t rep = 1);
```

1 Returns `replace(`*pos*`,` *n*`, wstring(`*c*`,` *rep*`))`.

17.5.2.1.25 `wstring::get_at(size_t)` [lib.wstring::get.at]

```
wchar_t get_at(size_t pos) const;
```

1 Reports an out-of-range error if *pos* `>=` *len*. Otherwise, the function returns *ptr*`[`*pos*`]`.

17.5.2.1.26 `wstring::put_at(size_t, wchar_t)` [lib.wstring::put.at]

```
void put_at(size_t pos, wchar_t c);
```

1 Reports an out-of-range error if *pos* `>=` *len*. Otherwise, the function assigns *c* to *ptr*`[`*pos*`]`.

17.5.2.1.27 `wstring::operator[](size_t)` [lib.wstring::op.array]

```
wchar_t operator[](size_t pos) const;
wchar_t& 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 a subsequent call to any member function for the object.

17.5.2.1.28 `wstring::data()` [lib.wstring::data]

```
const wchar_t* data() 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. 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 `wstring` that designates the same object as `*this`.

17.5.2.1.29 `wstring::length()` [lib.wstring::length]

```
size_t length() const:
```

1 Returns *len*.

17.5.2.1.30 `wstring::resize(size_t, wchar_t)` [lib.wstring::resize]

```
void resize(size_t n, wchar_t c = 0);
```

1 Reports a length error if *n* equals `NPOS`. Otherwise, the function alters the length of the string designated by `*this` as follows:

— 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`.

— 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*.

17.5.2.1.31 `wstring::reserve()` [lib.wstring::reserve]

```
size_t reserve() const;
```

1 Returns *res*.

17.5.2.1.32 `wstring::reserve(size_t)` [lib.wstring::reserve.cap]

```
void reserve(size_t res_arg);
```

1 If no string is allocated, the function assigns *res_arg* to *res*. Otherwise, whether or how the function alters *res* is unspecified.

17.5.2.1.33 `wstring::copy(wchar_t*, size_t, size_t)` [lib.wstring::copy.wstr]

```
size_t copy(wchar_t* s, size_t n, size_t pos = 0);
```

1 Reports an out-of-range error 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*.[121)]

3 The function returns *rlen*.

17.5.2.1.34 `wstring::find(const wstring&, size_t)` [lib.wstring::find.wsub]

```
size_t find(const wstring& 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`;

— `ptr[xpos + I] == str.ptr[I]` for all elements *I* of the string controlled by *str*.

2 If the function can determine such a value for *xpos*, it returns *xpos*. Otherwise, it returns `NPOS`.

17.5.2.1.35 `wstring::find(const wchar_t*, size_t, size_t)` [lib.wstring::find.wstr]

```
size_t find(const wchar_t* s, size_t pos = 0, size_t n = NPOS) const;
```

1 Returns `find(wstring(s, n), pos)`.

17.5.2.1.36 `wstring::find(wchar_t, size_t)` [lib.wstring::find.wc]

```
size_t find(wchar_t c, size_t pos = 0) const;
```

1 Returns `find(wstring(c), pos)`.

17.5.2.1.37 `wstring::rfind(const wstring&, size_t)` [lib.wstring::rfind.wsub]

```
size_t rfind(const wstring& 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`;

121) The function does not append a null wide character to the string.

— *ptr*[*xpos* + *I*] == *str*.*ptr*[*I*] for all elements *I* of the string controlled by *str*.

2 If the function can determine such a value for *xpos*, it returns *xpos*. Otherwise, it returns NPOS.

17.5.2.1.38 wstring::rfind(const wchar_t*, size_t, size_t) [lib.wstring::rfind.wstr]

```
size_t rfind(const wchar_t* s, size_t pos = NPOS, size_t n = NPOS)
        const;
```

1 Returns rfind(wstring(*s*, *n*), *pos*).

17.5.2.1.39 wstring::rfind(wchar_t, size_t) [lib.wstring::rfind.wc]

```
size_t rfind(wchar_t c, size_t pos = NPOS) const;
```

1 Returns rfind(wstring(*c*, *n*), *pos*).

17.5.2.1.40 wstring::find_first_of(const wstring&, size_t) [lib.wstring::find.first.of.wsub]

```
size_t find_first_of(const wstring& 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*;

— *ptr*[*xpos*] == *str*.*ptr*[*I*] for some element *I* of the string controlled by *str*.

2 If the function can determine such a value for *xpos*, it returns *xpos*. Otherwise, it returns NPOS.

17.5.2.1.41 wstring::find_first_of(const wchar_t*, size_t, size_t) [lib.wstring::find.first.of.wstr]

```
size_t find_first_of(const wchar_t* s, size_t pos = 0,
        size_t n = NPOS) const;
```

1 Returns find_first_of(wstring(*s*, *n*), *pos*).

17.5.2.1.42 wstring::find_first_of(wchar_t, size_t) [lib.wstring::find.first.of.wc]

```
size_t find_first_of(wchar_t c, size_t pos = 0) const;
```

1 Returns find_first_of(wstring(*c*), *pos*).

17.5.2.1.43 wstring::find_last_of(const wstring&, size_t) [lib.wstring::find.last.of.wsub]

```
size_t find_last_of(const wstring& 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*;

— *ptr*[*xpos*] == *str*.*ptr*[*I*] for some element *I* of the string controlled by *str*.

2 If the function can determine such a value for *xpos*, it returns *xpos*. Otherwise, it returns NPOS.

17.5.2.1.44 `wstring::find_last_of(const wchar_t*, size_t, size_t)` [lib.wstring::find.last.of.wstr]

```
size_t find_last_of(const wchar_t* s, size_t pos = NPOS,
        size_t n = NPOS) const;
```

1 Returns `find_last_of(wstring(s, n), pos)`.

17.5.2.1.45 `wstring::find_last_of(wchar_t, size_t)` [lib.wstring::find.last.of.wc]

```
size_t find_last_of(wchar_t c, size_t pos = NPOS) const;
```

1 Returns `find_last_of(wstring(c, n), pos)`.

17.5.2.1.46 `wstring::find_first_not_of(const wstring&, size_t)` [lib.wstring::find.first.not.of.wsub]

```
size_t find_first_not_of(const wstring& 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`;

— `ptr[xpos] == str.ptr[I]` for no element *I* of the string controlled by *str*.

2 If the function can determine such a value for *xpos*, it returns *xpos*. Otherwise, it returns `NPOS`.

17.5.2.1.47 `wstring::find_first_not_of(const wchar_t*, size_t, size_t)` [lib.wstring::find.first.not.of.wstr]

```
size_t find_first_not_of(const wchar_t* s, size_t pos = 0,
        size_t n = NPOS) const;
```

1 Returns `find_first_not_of(wstring(s, n), pos)`.

17.5.2.1.48 `wstring::find_first_not_of(wchar_t, size_t)` [lib.wstring::find.first.not.of.wc]

```
size_t find_first_not_of(wchar_t c, size_t pos = 0) const;
```

1 Returns `find_first_not_of(wstring(c), pos)`.

17.5.2.1.49 `wstring::find_last_not_of(const wstring&, size_t)` [lib.wstring::find.last.not.of.wsub]

```
size_t find_last_not_of(const wstring& 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;`

— `ptr[xpos] == str.ptr[I]` for no element *I* of the string controlled by *str*.

2 If the function can determine such a value for *xpos*, it returns *xpos*. Otherwise, it returns `NPOS`.

17.5.2.1.50 wstring::find_last_not_of(const wchar_t*, size_t, size_t) [lib.wstring::find.last.not.of.wstr]

```
size_t find_last_not_of(const wchar_t* s, size_t pos = NPOS,
        size_t n = NPOS) const;
```

1 Returns find_last_not_of(wstring(*s*, *n*), *pos*).

17.5.2.1.51 wstring::find_last_not_of(wchar_t, size_t) [lib.wstring::find.last.not.of.wc]

```
size_t find_last_not_of(wchar_t c, size_t pos = NPOS) const;
```

1 Returns find_last_not_of(wstring(*c*, *n*), *pos*).

17.5.2.1.52 wstring::substr(size_t, size_t) [lib.wstring::substr]

```
wstring substr(size_t pos = 0, size_t n = NPOS) const;
```

1 Returns wstring(*this, *pos*, *n*).

17.5.2.1.53 wstring::compare(const wstring&, size_t, size_t) [lib.wstring::compare.wsub]

```
int compare(const wstring& str, size_t pos, size_t n = NPOS) const;
```

1 Reports an out-of-range error if *pos* > *len*. Otherwise, if *str.len* < *n*, the function stores *str.len* in *n*. The function then determines the effective length *rlen* of the strings to compare as the smaller of *n* and *len* − *pos*. The function then compares the two strings by calling wcscmp(*ptr* + *pos*, *str.ptr*, *rlen*). The function signature wmemcmp(const wchar_t*, const wchar_t*, size_t) is declared in <cwchar>.

2 If the result of that comparison is nonzero, the function returns the nonzero result. Otherwise, the function returns:

— if *len* < *rlen*, a value less than zero;

— if *len* == *rlen*, the value zero;

— if *len* > *rlen*, a value greater than zero.

17.5.2.1.54 wstring::compare(const wchar_t*, size_t) [lib.wstring::compare.wstr]

```
size_t compare(const wchar_t* s, size_t n = NPOS) const;
```

1 Returns compare(wstring(*s*, *n*), *pos*).

17.5.2.1.55 wstring::compare(wchar_t, size_t) [lib.wstring::compare.wc]

```
size_t compare(wchar_t c, size_t rep = 1) const;
```

1 Returns compare(wstring(*c*, *rep*), *pos*).

17.5.2.2 `operator+(const wstring&, const wstring&)` [lib.op+.wsub.wsub]

```
wstring operator+(const wstring& lhs, const wstring& rhs);
```

1 Returns `wstring(`*`lhs`*`).append(`*`rhs`*`)`.

17.5.2.3 `operator+(const wchar_t*, const wstring&)` [lib.op+.wstr.wsub]

```
wstring operator+(const wchar_t* lhs, const wstring& rhs);
```

1 Returns `wstring(`*`lhs`*`) + `*`rhs`*.

17.5.2.4 `operator+(wchar_t, const wstring&)` [lib.op+.wc.wsub]

```
wstring operator+(wchar_t lhs, const wstring& rhs);
```

1 Returns `wstring(`*`lhs`*`) + `*`rhs`*.

17.5.2.5 `operator+(const wstring&, const wchar_t*)` [lib.op+.wsub.wstr]

```
wstring operator+(const wstring& lhs, const wchar_t* rhs);
```

1 Returns *`lhs`*` + wstring(`*`rhs`*`)`.

17.5.2.6 `operator+(const wstring&, wchar_t)` [lib.op+.wsub.wc]

```
wstring operator+(const wstring& lhs, wchar_t rhs);
```

1 Returns *`lhs`*` + wstring(`*`rhs`*`)`.

17.5.2.7 `operator==(const wstring&, const wstring&)` [lib.op==.wsub.wsub]

```
bool operator==(const wstring& lhs, const wstring& rhs);
```

1 Returns a nonzero value if *`lhs`*`.compare(`*`rhs`*`)` is zero.

17.5.2.8 `operator==(const wchar_t*, const wstring&)` [lib.op==.wstr.wsub]

```
bool operator==(const wchar_t* lhs, const wstring& rhs);
```

1 Returns `wstring(`*`lhs`*`) == `*`rhs`*.

17.5.2.9 `operator==(wchar_t, const wstring&)` [lib.op==.wc.wsub]

```
bool operator==(wchar_t lhs, const wstring& rhs);
```

1 Returns `wstring(`*`lhs`*`) == `*`rhs`*.

17.5.2.10 `operator==(const wstring&, const wchar_t*)` [lib.op==.wsub.wstr]

```
bool operator==(const wstring& lhs, const wchar_t* rhs);
```

1 Returns *`lhs`*` == wstring(`*`rhs`*`)`.

17.5.2.11 `operator==(const wstring&, wchar_t)` [lib.op==.wsub.wc]

```
bool operator==(const wstring& lhs, wchar_t rhs);
```

1 Returns *`lhs`*` == wstring(`*`rhs`*`)`.

17.5.2.12 `operator!=(const wstring&, const wstring&)` [lib.op!=.wsub.wsub]

```
bool operator!=(const wstring& lhs, const wstring& rhs);
```

1 Returns a nonzero value if `!(lhs == rhs)` is nonzero.

17.5.2.13 `operator!=(const wchar_t*, const wstring&)` [lib.op!=.wstr.wsub]

```
bool operator!=(const wchar_t* lhs, const wstring& rhs);
```

1 Returns `wstring(lhs) != rhs`.

17.5.2.14 `operator!=(wchar_t, const wstring&)` [lib.op!=.wc.wsub]

```
bool operator!=(wchar_t lhs, const wstring& rhs);
```

1 Returns `wstring(lhs) != rhs`.

17.5.2.15 `operator!=(const wstring&, const wchar_t*)` [lib.op!=.wsub.wstr]

```
bool operator!=(const wstring& lhs, const wchar_t* rhs);
```

1 Returns `lhs != wstring(rhs)`.

17.5.2.16 `operator!=(const wstring&, wchar_t)` [lib.op!=.wsub.wc]

```
bool operator!=(const wstring& lhs, wchar_t rhs);
```

1 Returns `lhs != wstring(rhs)`.

17.5.3 Header `<bits>` [lib.header.bits]

1 The header `<bits>` defines a template class and several related functions for representing and manipulating fixed-size sequences of bits.

17.5.3.1 Template class `bits<N>` [lib.template.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
};
```

1 The template class `bits<N>` describes an object that can store a sequence consisting of a fixed number of bits, *N*.

2 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.

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

— `char array[N]`, the sequence of bits, stored one bit per element.[122)]

4 The functions described in this subclause can report three kinds of errors, each associated with a distinct exception:

— an *invalid-argument* error is associated with exceptions of type `invalid_argument`;

— an *out-of-range* error is associated with exceptions of type `out_of_range`;

— an *overflow* error is associated with exceptions of type `overflow`.

5 To report one of these errors, the function evaluates the expression `ex.raise()`, where *ex* is an object of the associated exception type.

122) An implementation is free to store the bit sequence more efficiently.

17.5.3.1.1 `bits<N>::bits()` [lib.cons.bits]

```
bits();
```

1 Constructs an object of class `bits<N>`, initializing all bits to zero.

17.5.3.1.2 `bits<N>::bits(unsigned long)` [lib.cons.bits.ul]

```
bits(unsigned long val);
```
*

1 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)`. The macro `CHAR_BIT` is defined in `<climits>` (17.2).

2 If *M* < *N*, remaining bit positions are initialized to zero.

17.5.3.1.3 `bits<N>::bits(const string&, size_t, size_t)` [lib.cons.bits.subt]

```
bits(const string& str, size_t pos = 0, size_t n = NPOS);
```

1 Reports an out-of-range error 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*. The function then reports an invalid-argument error if any of the *rlen* characters in *str* beginning at position *pos* is other than `0` or `1`.

2 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*. 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.

3 If *M* < *N*, remaining bit positions are initialized to zero.

17.5.3.1.4 `bits<N>::operator&=(const bits<N>&)` [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. The function returns `*this`.

17.5.3.1.5 `bits<N>::operator|=(const bits<N>&)` [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. The function returns `*this`.

17.5.3.1.6 `bits<N>::operator^=(const bits<N>&)` [lib.bits::op^=.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. The function returns `*this`.

17.5.3.1.7 `bits<N>::operator<<=(size_t)` [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 The function returns `*this`.

17.5.3.1.8 `bits<N>::operator>>=(size_t)` [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 The function returns `*this`.

17.5.3.1.9 `bits<N>::set()` [lib.bits::set]

```
bits<N>& set();
```

1 Sets all bits in `*this`. The function returns `*this`.

17.5.3.1.10 `bits<N>::set(size_t, int)` [lib.bits::set.n]

```
bits<N>& set(size_t pos, int val = 1);
```

1 Reports an out-of-range error 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. The function returns `*this`.

17.5.3.1.11 `bits<N>::reset()` [lib.bits::reset]

```
bits<N>& reset();
```

1 Resets all bits in `*this`. The function returns `*this`.

17.5.3.1.12 `bits<N>::reset(size_t)` [lib.bits::reset.n]

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

1 Reports an out-of-range error if *pos* does not correspond to a valid bit position. Otherwise, the function resets the bit at position *pos* in `*this`. The function returns `*this`.

17.5.3.1.13 `bits<N>::operator~()` [lib.bits::op~]

```
bits<N> operator~() const;
```

1 Constructs an object *x* of class `bits<N>` and initializes it with `*this`. The function then returns `x.toggle()`.

17.5.3.1.14 `bits<N>::toggle()` [lib.bits::toggle]

```
bits<N>& toggle();
```

1 Toggles all bits in `*this`. The function returns `*this`.

17.5.3.1.15 `bits<N>::toggle(size_t)` [lib.bits::toggle.n]

```
bits<N>& toggle(size_t pos);
```

1 Reports an out-of-range error if *pos* does not correspond to a valid bit position. Otherwise, the function toggles the bit at position *pos* in `*this`. The function returns `*this`.

17.5.3.1.16 `bits<N>::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`, reports an overflow error. Otherwise, the function returns *x*.

17.5.3.1.17 `bits<N>::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`, reports an overflow error. Otherwise, the function returns *x*.

17.5.3.1.18 `bits<N>::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 The function returns the created object.

17.5.3.1.19 `bits<N>::count()` [lib.bits::count]

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

1 Returns a count of the number of bits set in `*this`.

17.5.3.1.20 `bits<N>::length()` [lib.bits::length]

```
size_t length() const;
```

1 Returns *N*.

17.5.3.1.21 `bits<N>::operator==(const bits<N>&)` [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*.

17.5.3.1.22 `bits<N>::operator!=(const bits<N>&)` [lib.bits::op!=.bt]

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

1 Returns a nonzero value if `!(*this == rhs)`.

17.5.3.1.23 `bits<N>::test(size_t)` [lib.bits::test]

```
bool test(size_t pos) const;
```

1 Reports an out-of-range error if *pos* does not correspond to a valid bit position. Otherwise, the function returns a nonzero value if the bit at position *pos* in `*this` has the value one.

17.5.3.1.24 `bits<N>::any()` [lib.bits::any]

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

1 Returns a nonzero value if any bit in `*this` is one.

17.5.3.1.25 `bits<N>::none()` [lib.bits::none]

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

1 Returns a nonzero value if no bit in `*this` is one.

17.5.3.1.26 `bits<N>::operator<<(size_t)` [lib.bits::op.lsh]

```
bits<N> operator<<(size_t pos) const;
```

1 Returns `bits<N>(*this) <<= pos`.

17.5.3.1.27 `bits<N>::operator>>(size_t)` [lib.bits::op.rsh]

```
bits<N> operator>>(size_t pos) const;
```

1 Returns `bits<N>(*this) >>= pos`.

17.5.3.2 `operator&(const bits<N>&, const bits<N>&)` [lib.op&.bt.bt]

```
bits<N> operator&(const bits<N>& lhs, const bits<N>& rhs);
```

1 Returns `bits<N>(lhs) &= pos`.

17.5.3.3 `operator|(const bits<N>&, const bits<N>&)` [lib.op|.bt.bt]

```
bits<N> operator|(const bits<N>& lhs, const bits<N>& rhs);
```

1 Returns `bits<N>(lhs) |= pos`.

17.5.3.4 `operator^(const bits<N>&, const bits<N>&)` [lib.op^.bt.bt]

```
bits<N> operator^(const bits<N>& lhs, const bits<N>& rhs);
```

1 Returns `bits<N>(lhs) ^= pos`.

17.5.3.5 `operator>>(istream&, bits<N>&)` [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 The function returns *is*.

17.5.3.6 `operator<<(ostream&, const bits<N>&)` [lib.ins.bt]

```
ostream& operator<<(ostream& os, const bits<N>& x);
```

1 Returns *os* `<<` *x*`.to_string()`.

17.5.4 Header `<bitstring>` [lib.header.bitstring]

1 The header `<bitstring>` defines a class and several function signatures for representing and manipulating varying-length sequences of bits. |

17.5.4.1 Class `bit_string` | [lib.bit.string]

Box 134

Library WG issue: Charles Allison, August 26, 1993

I don't appreciate the need for a `reserve()` function. I need someone to convince me. |

Recommend (Library WG meeting, San Diego, 3/8/94): |

For symmetry with strings: |
Get rid of `bitstring::trim()`. |
Add `bitstring::reserve()`.

```
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
};
```

1 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`.

2 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)`.[123] The integral value corresponding to two or more bits is the sum of their bit values.

[123] 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>`.

3 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;[124)]

— `size_t len`, the length of the bit sequence.

4 The functions described in this subclause can report three kinds of errors, each associated with a distinct exception:

— an *invalid-argument* error is associated with exceptions of type `invalid_argument`;

— 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`.

5 To report one of these errors, the function evaluates the expression *ex*`.raise()`, where *ex* is an object of the associated exception type.

17.5.4.1.1 bit_string::bit_string() [lib.cons.bit.string]

```
bit_string();
```

1 Constructs an object of class `bit_string`, initializing:

— *ptr* to an unspecified value;

— *len* to zero.

17.5.4.1.2 bit_string::bit_string(unsigned long, size_t) [lib.cons.bit.string.ul]

```
bit_string(unsigned long val, size_t n);
```

1 Reports a 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.

17.5.4.1.3 bit_string::bit_string(const bit_string&, size_t, size_t) [lib.cons.bit.string.bs]

```
bit_string(const bit_string& str, size_t pos = 0, size_t n = NPOS);
```

1 Reports an out-of-range error 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*.

124) An implementation is, of course, free to store the bit sequence more efficiently.

17.5.4.1.4 bit_string::bit_string(const string&, size_t, size_t) [lib.cons.bit.string.sub]

```
bit_string(const string& str, size_t pos = 0, size_t n = NPOS);
```

1 Reports an out-of-range error 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`. The function then reports an invalid-argument error if any of the `rlen` characters in `str` beginning at position `pos` is other than `0` or `1`.

2 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.

3 Thus, the function initializes:

— `ptr` to point at the first element of the string;

— `len` to `rlen`.

17.5.4.1.5 bit_string::operator+=(const bit_string&) [lib.bit.string::op+=.bs]

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

1 Reports a 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 The function returns `*this`.

17.5.4.1.6 bit_string::operator&=(const bit_string&) [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 The function returns `*this`.

17.5.4.1.7 bit_string::operator|=(const bit_string&) [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 The function returns `*this`.

17.5.4.1.8 `bit_string::operator^=(const bit_string&)` [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 the corresponding elements of `*this` and `rhs` have different values.

2 The function returns `*this`.

17.5.4.1.9 `bit_string::operator<<=(size_t)` [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 The function returns `*this`.

17.5.4.1.10 `bit_string::operator>>=(size_t)` [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`.

17.5.4.1.11 `bit_string::append(const bit_string&, size_t, size_t)` [lib.bit.string::append]

```
bit_string& append(const bit_string& str, size_t pos = 0,
        size_t n = NPOS);
```

1 Reports an out-of-range error 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 reports a 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 The function returns `*this`.

17.5.4.1.12 `bit_string::assign(const bit_string&, size_t, size_t)` [lib.bit.string::assign]

```
bit_string& assign(const bit_string& str, size_t pos = 0,
        size_t n = NPOS);
```

1 Reports an out-of-range error 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 The function returns `*this`.

17.5.4.1.13 `bit_string::insert(size_t, const bit_string&, size_t, size_t)` [lib.bit.string::insert]

```
bit_string& insert(size_t pos1, const bit_string& str, size_t pos2 = 0,
        size_t n = NPOS);
```

1 Reports an out-of-range error 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`. The function then reports a 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 The function returns `*this`.

17.5.4.1.14 `bit_string::remove(size_t, size_t)` [lib.bit.string::remove]

```
bit_string& remove(size_t pos = 0, size_t n = NPOS);
```

1 Reports an out-of-range error 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 The function returns `*this`.

17.5.4.1.15 `bit_string::replace(size_t, size_t, const bit_string&, size_t, size_t)` [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 Reports an out-of-range error 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`. The function then reports a 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 The function returns `*this`.

17.5.4.1.16 bit_string::set() [lib.bit.string::set]

```
bit_string& set();
```

1 Sets all elements of the string controlled by *this. The function returns *this.

17.5.4.1.17 bit_string::set(size_t, bool) [lib.bit.string::set.n]

```
bit_string& set(size_t pos, bool val = 1);
```

1 Reports an out-of-range error 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*. 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. The function returns *this.

17.5.4.1.18 bit_string::reset() [lib.bit.string::reset]

```
bit_string& reset();
```

1 Resets all elements of the string controlled by *this. The function returns *this.

17.5.4.1.19 bit_string::reset(size_t) [lib.bit.string::reset.n]

```
bit_string& reset(size_t pos);
```

1 Reports an out-of-range error 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.

17.5.4.1.20 bit_string::toggle() [lib.bit.string::toggle]

```
bit_string& toggle();
```

1 Toggles all elements of the string controlled by *this. The function returns *this.

17.5.4.1.21 bit_string::toggle(size_t) [lib.bit.string::toggle.n]

```
bit_string& toggle(size_t pos);
```

1 Reports an out-of-range error if *pos* >= *len*. Otherwise, the function toggles the element at position *pos* in *this.

17.5.4.1.22 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 The function returns the created object.

17.5.4.1.23 `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`.

17.5.4.1.24 `bit_string::length()` | [lib.bit.string::length]

```
size_t length() const;
```

1 Returns *len*.

17.5.4.1.25 `bit_string::resize(size_t, bool)` | [lib.bit.string::resize]

```
size_t resize(size_t n, bool val = 0);
```

1 Reports a 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 The function returns the previous value of *len*.

17.5.4.1.26 `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 The function returns the new value of *len*.

17.5.4.1.27 `bit_string::find(bool, size_t, size_t)` | [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 If the function can determine such a value for *xpos*, it returns *xpos*. Otherwise, it returns `NPOS`.

17.5.4.1.28 bit_string::rfind(bool, size_t, size_t) [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 If the function can determine such a value for *xpos*, it returns *xpos*. Otherwise, it returns NPOS.

17.5.4.1.29 bit_string::substr(size_t, size_t) [lib.bit.string::substr]

```
bit_string substr(size_t pos, size_t n = NPOS) const;
```

1 Returns bit_string(*this, *pos*, *n*).

17.5.4.1.30 bit_string::operator==(const bit_string&) [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*.

17.5.4.1.31 bit_string::operator!=(const bit_string&) [lib.bit.string::op!=.bs]

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

1 Returns a nonzero value if !(*this == *rhs*).

17.5.4.1.32 bit_string::test(size_t) [lib.bit.string::test]

```
bool test(size_t pos) const;
```

1 Reports an out-of-range error if *pos* >= *len*. Otherwise, the function returns a nonzero value if the element at position *pos* in the string controlled by *this is one.

17.5.4.1.33 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.

17.5.4.1.34 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.

17.5.4.1.35 bit_string::operator<<(size_t) [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. The function then returns *x* <<= *pos*.

17.5.4.1.36 bit_string::operator>>(size_t) [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. The function then returns *x* >>= *pos*.

17.5.4.1.37 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. The function then returns *x*.toggle().

17.5.4.2 operator+(const bit_string&, const bit_string&) [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*. The function then returns *x* += *rhs*.

17.5.4.3 operator&(const bit_string&, const bit_string&) [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*. The function then returns *x* &= *rhs*.

17.5.4.4 operator|(const bit_string&, const bit_string&) [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*. The function then returns *x* |= *rhs*.

17.5.4.5 operator^(const bit_string&, const bit_string&) [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*. The function then returns *x* ^= *rhs*.

17.5.4.6 operator>>(istream&, bit_string&) [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 The function returns *is*.

17.5.4.7 operator<<(ostream&, const bit_string&) [lib.ins.bs]

```
ostream& operator<<(ostream& os, const bit_string& x);
```

1 Returns `os << x.to_string()`.

17.5.5 Header <dynarray> [lib.header.dynarray]

1 The header `<dynarray>` defines a template class and several related functions for representing and manipulating varying-size sequences of some object type *T*.

17.5.5.1 Template class dyn_array<*T*> [lib.template.dyn.array]

Box 135

Library WG issue: Uwe Steinmüller, January 21, 1994

missing

```
~dynarray()
dynarray<T>& operator=(const dynarray<T>&);
```

Box 136

Library WG issue: Dag Brück, December 12, 1993

The introduction (17.5.5.1) should have a summary of all operations that resize the array and possibly move its elements.

```
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
};
```

1 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*().

2 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.[125)]

[125)] 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*>.

3 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.

4 In all cases, *`len`* `<=` *`res`*.

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`;

6 To report one of these errors, the function evaluates the expression *`ex`*`.raise()`, where *`ex`* is an object of the associated exception type.

17.5.5.1.1 `dyn_array<T>::dyn_array()` [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.

17.5.5.1.2 `dyn_array<T>::dyn_array(size_t, capacity)` [lib.cons.dyn.array.cap]

```
dyn_array(size_t size, capacity cap);
```

1 Reports a 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`*.

2 Otherwise, *`cap`* shall be *`reserve`* and the function initializes:

— *`ptr`* to an unspecified value;

— *`len`* to zero;

— *`res`* to *`size`*.

17.5.5.1.3 dyn_array<T>::dyn_array(const dyn_array<T>&) [lib.cons.dyn.array.da]

```
dyn_array(const dyn_array<T>& arr);
```

1 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*.

17.5.5.1.4 dyn_array<T>::dyn_array(const T&, size_t) [lib.cons.dyn.array.t]

```
dyn_array(const T& obj, size_t rep = 1);
```

1 Reports a 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*.

17.5.5.1.5 dyn_array<T>::dyn_array(const T*, size_t) [lib.cons.dyn.array.pt]

```
dyn_array(const T* parr, size_t n);
```

1 Reports a length error if *n* equals NPOS. Otherwise, the function reports an invalid-argument error if *parr* is a null pointer. Otherwise, *parr* shall designate the first element of an array of at least *n* elements of type *T*.

2 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*.

17.5.5.1.6 dyn_array<T>::operator+=(const dyn_array<T>&) [lib.dyn.array::op+=.da]

Box 137

Library WG issue: Dag Brück, December 12, 1993

I find it very questionable that dynarray is allowed to do initialization as a sequence of default constructor + assignment. We know how to get around that problem (new with placement syntax). However, I understand that the library WG has been through all this before, but I really don't like it.

Comment (Library WG meeting, San Diego, 3/8/94):

What do we say about whether the default constructor is used, followed by assignment; versus using the copy constructor?

```
dyn_array<T>& operator+=(const dyn_array<T>& rhs);
```

1 Reports a 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 The function returns *this.

17.5.5.1.7 dyn_array<*T*>::operator+=(const *T*&) [lib.dyn.array::op+=.t]

```
dyn_array<T>& operator+=(const T& obj);
```

1 Returns append(*obj*).

17.5.5.1.8 dyn_array<*T*>::append(const *T*&, size_t) [lib.dyn.array::append.t]

```
dyn_array<T>& append(const T& obj, size_t rep = 1);
```

1 Reports a 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 The function returns *this.

17.5.5.1.9 dyn_array<*T*>::append(const *T**, size_t) [lib.dyn.array::append.pt]

```
dyn_array<T>& append(const T* parr, size_t n = 1);
```

1 Reports a length error if *len* >= NPOS - *n*. Otherwise, the function reports an invalid-argument error 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*.

2 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*.

3 The function returns *this.

17.5.5.1.10 dyn_array<*T*>::assign(const *T*&, size_t) [lib.dyn.array::assign.t]

```
dyn_array<T>& assign(const T& obj, size_t rep = 1);
```

1 Reports a 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 The function returns *this.

17.5.5.1.11 dyn_array<T>::assign(const T*, size_t) [lib.dyn.array::assign.pt]

```
dyn_array<T>& assign(const T* parr, size_t n = 1);
```

1 Reports a length erroriif *n* == NPOS. Otherwise, the function reports an invalid-argument error 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*.

2 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*.

3 The function returns *this.

17.5.5.1.12 dyn_array<T>::insert(size_t, const dyn_array<T>&) [lib.dyn.array::insert.da]

```
dyn_array<T>& insert(size_t pos, const dyn_array<T>& arr);
```

1 Reports an out-of-range error if *pos* > *len*. Otherwise, the function reports a 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 The function returns *this.

17.5.5.1.13 dyn_array<T>::insert(size_t, const T&, size_t) [lib.dyn.array::insert.t]

```
dyn_array<T>& insert(size_t pos, const T& obj, size_t rep = 1);
```

1 Reports an out-of-range error if *pos* > *len*. Otherwise, the function reports a length error if *len* >= NPOS - *rep*.

2 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.

3 The function returns *this.

17.5.5.1.14 dyn_array<T>::insert(size_t, const T*, size_t) [lib.dyn.array::insert.pt]

```
dyn_array<T>& insert(size_t pos, const T* parr, size_t n = 1);
```

1 Reports an out-of-range error if *pos* > *len*. Otherwise, the function reports a length error if *len* >= NPOS - *n*. Otherwise, the function reports an invalid-argument error 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*.

2 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.

3 The function returns `*this`.

17.5.5.1.15 `dyn_array<T>::remove(size_t, size_t)` [lib.dyn.array::remove]

> **Box 138**
>
> **Library WG issue:** Dag Brück, December 12, 1993
>
> I find it unintuitive that `da.remove(4);` removes all the elements starting at postion 4. I.e., I think the default value for `n` should be 1 instead of `NPOS`.
>
> Recommend (Library WG meeting, San Diego, 3/8/94):
>
> Should `dynarray<T>::remove(4)` remove to end of string, or just element 4?

```
dyn_array<T>& remove(size_t pos = 0, size_t n = NPOS);
```

1 Reports an out-of-range error 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 The function returns `*this`.

17.5.5.1.16 `dyn_array<T>::sub_array(dyn_array<T>&, size_t, size_t)` [lib.dyn.array::sub.array]

```
dyn_array<T>& sub_array(dyn_array<T>& arr, size_t pos, size_t n = NPOS);
```

1 Reports an out-of-range error 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 The function returns *arr*.

17.5.5.1.17 `dyn_array<T>::swap(dyn_array<T>&)` [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`.[126)]

17.5.5.1.18 `dyn_array<T>::get_at(size_t)` [lib.dyn.array::get.at]

```
const T& get_at(size_t pos) const;
```

1 Reports an out-of-range error if *pos >= len*. Otherwise, the function returns `ptr[pos]`.

126) Presumably, this operation occurs with no actual copying of array elements.

2 The reference returned is invalid after a subsequent call to any member function for the object.

17.5.5.1.19 dyn_array<*T*>::put_at(size_t, const *T*&) [lib.dyn.array::put.at]

```
void put_at(size_t pos, const T& obj);
```

1 Reports an out-of-range error if *pos* >= *len*. Otherwise, the function assigns *obj* to the element at position *pos* in the dynamic array designated by *this.

17.5.5.1.20 dyn_array<*T*>::operator[](size_t) [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.

17.5.5.1.21 dyn_array<*T*>::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.

17.5.5.1.22 dyn_array<*T*>::length() [lib.dyn.array::length]

```
size_t length() const;
```

1 Returns *len*.

17.5.5.1.23 dyn_array<*T*>::resize(size_t) [lib.dyn.array::resize]

```
void resize(size_t n);
```

1 Reports a 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*.

17.5.5.1.24 dyn_array<*T*>::resize(size_t, const *T*&) [lib.dyn.array::resize.t]

```
void resize(size_t n, const T& obj);
```

1 Reports a 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*.

17.5.5.1.25 `dyn_array<T>::reserve()` [lib.dyn.array::reserve]

```
size_t reserve() const;
```

1 Returns *res*.

17.5.5.1.26 `dyn_array<T>::reserve(size_t)` [lib.dyn.array::reserve.cap]

```
void reserve(size_t res_arg);
```

1 If no dynamic array is allocated, assigns *res_arg* to *res*. Otherwise, whether or how the function alters *res* is unspecified.

17.5.5.2 `operator+(const dyn_array<T>&, const dyn_array<T>&)` [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`.

17.5.5.3 `operator+(const dyn_array<T>&, const T&)` [lib.op+.da.t]

```
dyn_array<T> operator+(const dyn_array<T>& lhs, const T& obj);
```

1 Returns `dyn_array<T>(lhs) += rhs`.

17.5.5.4 `operator+(const T&, const dyn_array<T>&)` [lib.op+.t.da]

```
dyn_array<T> operator+(const T& obj, const dyn_array<T>& rhs);
```

1 Returns `dyn_array<T>(lhs) += rhs`.

17.5.6 Header `<ptrdynarray>` [lib.header.ptrdynarray]

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*. *

17.5.6.1 Template class `ptrdyn_array<T>` [lib.template.ptr.dyn.array]

```
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 ptrdyn_array<T>& arr);
        ptr_dyn_array(T* obj, size_t rep = 1);
        ptr_dyn_array(T** parr, size_t n = 1);
        ptrdyn_array<T>& operator+=(const ptrdyn_array<T>& rhs);
        ptrdyn_array<T>& operator+=(T* obj);
        ptrdyn_array<T>& append(T* obj, size_t rep = 1);
        ptrdyn_array<T>& append(T** parr, size_t n = 1);
        ptrdyn_array<T>& assign(T* obj, size_t rep = 1);
        ptrdyn_array<T>& assign(T** parr, size_t n = 1);
        ptrdyn_array<T>& insert(size_t pos, const ptrdyn_array<T>& arr);
        ptrdyn_array<T>& insert(size_t pos, T* obj, size_t rep = 1);
        ptrdyn_array<T>& insert(size_t pos, T** parr, size_t n = 1);
        ptrdyn_array<T>& remove(size_t pos = 0, size_t n = NPOS);
        ptrdyn_array<T>& sub_array(ptrdyn_array<T>& arr, size_t pos,
                size_t n = NPOS);
        void swap(ptrdyn_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);
};
```

1 The template class `ptrdyn_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.

17.5.6.1.1 `ptrdyn_array<T>::ptr_dyn_array()` [lib.cons.ptr.dyn.array]

```
ptr_dyn_array();
```

1 Constructs an object of class `ptrdyn_array<T>`, initializing the base class with `dyn_array<void*>()`.

17.5.6.1.2 `ptrdyn_array<T>::ptr_dyn_array(size_t, capacity)` [lib.cons.ptr.dyn.array.cap]

```
ptr_dyn_array(size_t size, capacity cap);
```

1 Constructs an object of class `ptrdyn_array<T>`, initializing the base class with `dyn_array<void*>(size, cap)`.

17.5.6.1.3 `ptrdyn_array<T>::ptr_dyn_array(const ptrdyn_array<T>&)` [lib.cons.ptr.dyn.array.pda]

```
ptr_dyn_array(const ptrdyn_array<T>& arr);
```

ptrdyn_array<*T*>::ptr_dyn_array(const ptrdyn_array<*T*>&)

1 Constructs an object of class `ptrdyn_array<T>`, initializing the base class with `dyn_array<void*>(arr)`.

17.5.6.1.4 ptrdyn_array<*T*>::ptr_dyn_array(*T**) [lib.cons.ptr.dyn.array.pt]

```
ptr_dyn_array(T* obj, size_t rep = 1);
```

1 Constructs an object of class `ptrdyn_array<T>`, initializing the base class with `dyn_array<void*>((void*)obj, rep)`.

17.5.6.1.5 ptrdyn_array<*T*>::ptr_dyn_array(const *T***, size_t) [lib.cons.ptr.dyn.array.ppt]

```
ptr_dyn_array(const T** parr, size_t n);
```

1 Constructs an object of class `ptrdyn_array<T>`, initializing the base class with `dyn_array<void*>((void**)parr, n)`.

17.5.6.1.6 ptrdyn_array<*T*>::operator+=(const ptrdyn_array<*T*>&) [lib.ptr.dyn.array::op+=.pda]

```
ptrdyn_array<T>& operator+=(const ptrdyn_array<T>& rhs);
```

1 Returns `(ptrdyn_array<T>&)dyn_array<void*>::operator+=((const dyn_array<void*>&)rhs)`.

17.5.6.1.7 ptrdyn_array<*T*>::operator+=(*T**) [lib.ptr.dyn.array::op+=.pt]

```
ptrdyn_array<T>& operator+=(T* obj);
```

1 Returns `(ptrdyn_array<T>&)dyn_array<void*>:: operator+=((void*)obj)`.

17.5.6.1.8 ptrdyn_array<*T*>::append(*T**, size_t) [lib.ptr.dyn.array::append.pt]

```
ptrdyn_array<T>& append(T* obj, size_t rep = 1);
```

1 Returns `(ptrdyn_array<T>&)dyn_array<void*>::append((void*)obj, rep)`.

17.5.6.1.9 ptrdyn_array<*T*>::append(*T***, size_t) [lib.ptr.dyn.array::append.ppt]

```
ptrdyn_array<T>& append(T** parr, size_t n = 1);
```

1 Returns `(ptrdyn_array<T>&)dyn_array<void*>::append((void**)parr, n)`.

17.5.6.1.10 ptrdyn_array<*T*>::assign(*T**, size_t) [lib.ptr.dyn.array::assign.pt]

```
ptrdyn_array<T>& assign(T* obj, size_t rep = 1);
```

1 Returns `(ptrdyn_array<T>&)dyn_array<void*>::assign((void*)obj, rep)`.

17.5.6.1.11 ptrdyn_array<*T*>::assign(*T***, size_t) [lib.ptr.dyn.array::assign.ppt]

```
ptrdyn_array<T>& assign(T** parr, size_t n = 1);
```

1 Returns `(ptrdyn_array<T>&)dyn_array<void*>::assign((void**)parr, n)`.

17.5.6.1.12 ptrdyn_array<*T*>::insert(size_t, const ptrdyn_array<*T*>&, size_t) [lib.ptr.dyn.array::insert.pda]

```
ptrdyn_array<T>& insert(size_t pos, const ptrdyn_array<T>& arr);
```

1 Returns `(ptrdyn_array<T>&)dyn_array<void*>::insert(pos, (const dyn_array<void*>&)arr)`.

17.5.6.1.13 ptrdyn_array<*T*>::insert(size_t, *T**, size_t) [lib.ptr.dyn.array::insert.pt]

```
ptrdyn_array<T>& insert(size_t pos, T*obj, size_t rep = 1);
```

1 Returns `(ptrdyn_array<T>&)dyn_array<void*>::insert(pos, (void*)obj, rep)`.

17.5.6.1.14 ptrdyn_array<*T*>::insert(size_t, *T***, size_t) [lib.ptr.dyn.array::insert.ppt]

```
ptrdyn_array<T>& insert(size_t pos, T**parr, size_t n = 1);
```

1 Returns `(ptrdyn_array<T>&)dyn_array<void*>::insert(pos, (void**)parr, n)`.

17.5.6.1.15 ptrdyn_array<*T*>::remove(size_t, size_t) [lib.ptr.dyn.array::remove]

```
ptrdyn_array<T>& remove(size_t pos = 0, size_t n = NPOS);
```

1 Returns `(ptrdyn_array<T>&)dyn_array<void*>::remove(pos, n)`.

17.5.6.1.16 ptrdyn_array<*T*>::sub_array(ptrdyn_array<*T*>&, size_t, size_t) [lib.ptr.dyn.array::sub.array]

```
ptrdyn_array<T>& sub_array(ptrdyn_array<T>& arr, size_t pos,
        size_t n = NPOS);
```

1 Returns `(ptrdyn_array<T>&)dyn_array<void*>::sub_array(arr, pos, n)`.

17.5.6.1.17 ptrdyn_array<*T*>::swap(ptrdyn_array<*T*>&) [lib.ptr.dyn.array::swap]

```
void swap(ptrdyn_array<T>& arr);
```

1 Calls `dyn_array<void*>::swap(arr)`.

17.5.6.1.18 ptrdyn_array<*T*>::get_at(size_t) [lib.ptr.dyn.array::get.at]

```
T* get_at(size_t pos) const;
```

1 Returns `(T*)dyn_array<void*>::get_at(pos)`.

17.5.6.1.19 ptrdyn_array<*T*>::put_at(size_t, const *T*&) [lib.ptr.dyn.array::put.at]

```
void put_at(size_t pos, T* obj);
```

1 Calls `dyn_array<void*>::put_at(pos, (void*)obj)`.

17.5.6.1.20 `ptrdyn_array<T>::operator[](size_t)` [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)`.

17.5.6.1.21 `ptrdyn_array<T>::data()` [lib.ptr.dyn.array::data]

```
T** data();
const T** data() const;
```

1 Returns `(T*)dyn_array<void*>::data()`.

17.5.6.1.22 `ptrdyn_array<T>::length()` [lib.ptr.dyn.array::length]

```
size_t length() const;
```

1 Returns `dyn_array<void*>::length()`.

17.5.6.1.23 `ptrdyn_array<T>::resize(size_t)` [lib.ptr.dyn.array::resize]

```
void resize(size_t n);
```

1 Calls `dyn_array<void*>::resize(n)`.

17.5.6.1.24 `ptrdyn_array<T>::resize(size_t, T*)` [lib.ptr.dyn.array::resize.pt]

```
void resize(size_t n, T* obj);
```

1 Calls `dyn_array<void*>::resize(n, (void*)obj)`.

17.5.6.1.25 `ptrdyn_array<T>::reserve()` [lib.ptr.dyn.array::reserve]

```
size_t reserve() const;
```

1 Returns `dyn_array<void*>::reserve()`.

17.5.6.1.26 `ptrdyn_array<T>::reserve(size_t)` [lib.ptr.dyn.array::reserve.cap]

```
void reserve(size_t res_arg);
```

1 Returns `dyn_array<void*>::reserve(res_arg)`.

17.5.6.2 `operator+(const ptrdyn_array<T>&, const ptrdyn_array<T>&)` [lib.op+.pda.pda]

```
ptrdyn_array<T> operator+(const ptrdyn_array<T>& lhs,
        const ptrdyn_array<T>& rhs);
```

1 Returns `ptrdyn_array<T><T>(lhs) += rhs)`.

17.5.6.3 `operator+(const ptrdyn_array<T>&, T*)` [lib.op+.pda.pt]

```
ptrdyn_array<T> operator+(const ptrdyn_array<T>& lhs, T* obj);
```

1 Returns `ptrdyn_array<T><T>(lhs) += rhs)`.

17.5.6.4 `operator+(T*, const ptrdyn_array<T>&)` [lib.op+.pt.pda]

```
ptrdyn_array<T> operator+(T* obj, const ptrdyn_array<T>& rhs);
```

1 Returns `ptrdyn_array<T><T>(lhs) += rhs)`.

17.5.7 Header `<complex>` [lib.header.complex]

Box 139
Library WG issue: Bjarne Stroustrup, November 10, 1993

The complex components should be specified as templates.

Box 140
Library WG issue: Al Vermeulen , September 28, 1993

The complex classes need to be reviewed and verified.

1 The header `<complex>` defines a macro, three types, and numerous functions for representing and manipulating complex numbers.

2 The macro is:

```
__STD_COMPLEX
```

3 whose definition is unspecified.

17.5.7.1 Complex numbers with `float` precision [lib.complex.with.float]

17.5.7.1.1 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.

17.5.7.1.1.1 `float_complex::float_complex(float, float)` [lib.cons.float.complex.f.f]

```
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*.

17.5.7.1.1.2 `operator+=(float_complex)` [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`.

17.5.7.1.1.3 `operator-=(float_complex)` [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`.

17.5.7.1.1.4 `operator*=(float_complex)` [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`.

17.5.7.1.1.5 `operator/=(float_complex)` [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`.

17.5.7.1.2 `_float_complex(const double_complex&)` [lib..float.complex.dc]

```
float_complex _float_complex(const double_complex& rhs);
```

1 Returns `float_complex((float)real(rhs), (float)imag(rhs))`.

17.5.7.1.3 `_float_complex(const long_double_complex&)` [lib..float.complex.ldc]

```
float_complex _float_complex(const long_double_complex& rhs);
```

1 Returns `float_complex((float)real(rhs), (float)imag(rhs))`.

17.5.7.1.4 `operator+(float_complex, float_complex)` [lib.op+.fc.fc]

```
float_complex operator+(float_complex lhs, float_complex rhs);
```

1 Returns `float_complex(lhs) += rhs`.

17.5.7.1.5 `operator+(float_complex, float)` [lib.op+.fc.f]

```
float_complex operator+(float_complex lhs, float rhs);
```

1 Returns `float_complex(lhs) += float_complex(rhs)`.

17.5.7.1.6 `operator+(float, float_complex)` [lib.op+.f.fc]

```
float_complex operator+(float lhs, float_complex rhs);
```

1 Returns `float_complex(lhs) += rhs`.

17.5.7.1.7 `operator-(float_complex, float_complex)` [lib.op-.fc.fc]

```
float_complex operator-(float_complex lhs, float_complex rhs);
```

1 Returns `float_complex(lhs) -= rhs`.

17.5.7.1.8 `operator-(float_complex, float)` [lib.op-.fc.f]

```
float_complex operator-(float_complex lhs, float rhs);
```

1 Returns `float_complex(lhs) -= float_complex(rhs)`.

17.5.7.1.9 `operator-(float, float_complex)` [lib.op-.f.fc]

```
float_complex operator-(float lhs, float_complex rhs);
```

1 Returns `float_complex(lhs) -= rhs`.

17.5.7.1.10 `operator*(float_complex, float_complex)` [lib.op*.fc.fc]

```
float_complex operator*(float_complex lhs, float_complex rhs);
```

1 Returns `float_complex(lhs) *= rhs`.

17.5.7.1.11 `operator*(float_complex, float)` [lib.op*.fc.f]

```
float_complex operator*(float_complex lhs, float rhs);
```

1 Returns `float_complex(lhs) *= float_complex(rhs)`.

17.5.7.1.12 `operator*(float, float_complex)` [lib.op*.f.fc]

```
float_complex operator*(float lhs, float_complex rhs);
```

1 Returns `float_complex(lhs) *= rhs`.

17.5.7.1.13 `operator/(float_complex, float_complex)` [lib.op/.fc.fc]

```
float_complex operator/(float_complex lhs, float_complex rhs);
```

1 Returns `float_complex(lhs) /= rhs`.

17.5.7.1.14 `operator/(float_complex, float)` [lib.op/.fc.f]

```
float_complex operator/(float_complex lhs, float rhs);
```

1 Returns `float_complex(lhs) /= float_complex(rhs)`.

17.5.7.1.15 `operator/(float, float_complex)` [lib.op/.f.fc]

```
float_complex operator/(float lhs, float_complex rhs);
```

1 Returns `float_complex(lhs) /= rhs`.

17.5.7.1.16 operator+(float_complex) [lib.op+.fc]

```
float_complex operator+(float_complex lhs);
```

1 Returns `float_complex(lhs)`.

17.5.7.1.17 operator-(float_complex) [lib.op-.fc]

```
float_complex operator-(float_complex lhs);
```

1 Returns `float_complex(-real(lhs), -imag(lhs))`.

17.5.7.1.18 operator==(float_complex, float_complex) [lib.op==.fc.fc]

```
bool operator==(float_complex lhs, float_complex >rhs);
```

1 Returns `real(lhs) == real(rhs) && imag(lhs) == imag(rhs)`.

17.5.7.1.19 operator==(float_complex, float) [lib.op==.fc.f]

```
bool operator==(float_complex lhs, float rhs);
```

1 Returns `real(lhs) == rhs && imag(lhs) == 0`.

17.5.7.1.20 operator==(float, float_complex) [lib.op==.f.fc]

```
bool operator==(float lhs, float_complex rhs);
```

1 Returns `lhs == real(rhs) && imag(rhs) == 0`.

17.5.7.1.21 operator!=(float_complex, float_complex) [lib.op!=.fc.fc]

```
bool operator!=(float_complex lhs, float_complex rhs);
```

1 Returns `real(lhs) != real(rhs) || imag(lhs) != imag(rhs)`.

17.5.7.1.22 operator!=(float_complex, float) [lib.op!=.fc.f]

```
bool operator!=(float_complex lhs, float rhs);
```

1 Returns `real(lhs) != rhs || imag(lhs) != 0`.

17.5.7.1.23 operator!=(float, float_complex) [lib.op!=.f.fc]

```
bool operator!=(float lhs, float_complex rhs);
```

1 Returns `lhs != real(rhs) || imag(rhs) != 0`.

17.5.7.1.24 operator>>(istream&, float_complex&) [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 The function returns *is*.

17.5.7.1.25 `operator<<(ostream&, float_complex)` [lib.ins.fc]

```
ostream& operator<<(ostream& os, float_complex x);
```

1 Returns `os << '(' << real(x) << ',' << imag(x) << ')'`.

17.5.7.1.26 `abs(float_complex)` [lib.abs.fc]

```
float abs(float_complex x);
```

1 Returns the magnitude of *x*.

17.5.7.1.27 `arg(float_complex)` [lib.arg.fc]

```
float arg(float_complex x);
```

1 Returns the phase angle of *x*.

17.5.7.1.28 `conj(float_complex)` [lib.conj.fc]

```
float_complex conj(float_complex x);
```

1 Returns the conjugate of *x*.

17.5.7.1.29 `cos(float_complex)` [lib.cos.fc]

```
float_complex cos(float_complex x);
```

1 Returns the cosine of *x*.

17.5.7.1.30 `cosh(float_complex)` [lib.cosh.fc]

```
float_complex cosh(float_complex x);
```

1 Returns the hyperbolic cosine of *x*.

17.5.7.1.31 `exp(float_complex)` [lib.exp.fc]

```
float_complex exp(float_complex x);
```

1 Returns the exponential of *x*.

17.5.7.1.32 `imag(float_complex)` [lib.imag.fc]

```
float imag(float_complex x);
```

1 Returns the imaginary part of *x*.

17.5.7.1.33 `log(float_complex)` [lib.log.fc]

```
float_complex log(float_complex x);
```

1 Returns the logarithm of *x*.

17.5.7.1.34 norm(float_complex) [lib.norm.fc]

```
float norm(float_complex x);
```

1 Returns the squared magnitude of *x*.

17.5.7.1.35 polar(float, float) [lib.polar.f.f]

```
float_complex polar(float rho, float theta);
```

1 Returns the float_complex value corresponding to a complex number whose magnitude is *rho* and whose phase angle is *theta*.

17.5.7.1.36 pow(float_complex, float_complex) [lib.pow.fc.fc]

```
float_complex pow(float_complex x, float_complex y);
```

1 Returns *x* raised to the power *y*.

17.5.7.1.37 pow(float_complex, float) [lib.pow.fc.f]

```
float_complex pow(float_complex x, float y);
```

1 Returns *x* raised to the power *y*.

17.5.7.1.38 pow(float_complex, int) [lib.pow.fc.i]

```
float_complex pow(float_complex x, int y);
```

1 Returns *x* raised to the power *y*.

17.5.7.1.39 pow(float, float_complex) [lib.pow.f.fc]

```
float_complex pow(float x, float_complex y);
```

1 Returns *x* raised to the power *y*.

17.5.7.1.40 real(float_complex) [lib.real.fc]

```
float real(float_complex x);
```

1 Returns the real part of *x*.

17.5.7.1.41 sin(float_complex) [lib.sin.fc]

```
float_complex sin(float_complex x);
```

1 Returns the sine of *x*.

17.5.7.1.42 sinh(float_complex) [lib.sinh.fc]

```
float_complex sinh(float_complex x);
```

1 Returns the hyperbolic sine of *x*.

17.5.7.1.43 sqrt(float_complex) [lib.sqrt.fc]

```
float_complex sqrt(float_complex x);
```

1 Returns the square root of *x*.

17.5.7.2 Complex numbers with double precision [lib.complex.with.d]

17.5.7.2.1 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.

17.5.7.2.1.1 double_complex::double_complex(double, double) [lib.cons.double.complex.d.d]

```
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*.

17.5.7.2.1.2 double_complex::double_complex(float_complex&) [lib.cons.double.complex.fc]

```
double_complex(float_complex& rhs);
```

1 Constructs an object of class `double_complex`, initializing *re* to `(double)real(`*rhs*`)` and *im* to `(double)imag(`*rhs*`)`.

17.5.7.2.1.3 operator+=(double_complex) [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`. The function returns `*this`.

17.5.7.2.1.4 operator-=(double_complex) [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`. The function returns `*this`.

17.5.7.2.1.5 operator*=(double_complex) [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`. The function returns `*this`.

17.5.7.2.1.6 operator/=(double_complex) [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`. The function returns `*this`.

17.5.7.2.2 _double_complex(const long_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)).`

17.5.7.2.3 operator+(double_complex, double_complex) [lib.op+.dc.dc]

```
double_complex operator+(double_complex lhs, double_complex rhs);
```

1 Returns `double_complex(lhs) += rhs`.

17.5.7.2.4 operator+(double_complex, double) [lib.op+.dc.d]

```
double_complex operator+(double_complex lhs, double rhs);
```

1 Returns `double_complex(lhs) += double_complex(rhs)`.

17.5.7.2.5 operator+(double, double_complex) [lib.op+.d.dc]

```
double_complex operator+(double lhs, double_complex rhs);
```

1 Returns `double_complex(lhs) += rhs`.

17.5.7.2.6 operator-(double_complex, double_complex) [lib.op-.dc.dc]

```
double_complex operator-(double_complex lhs, double_complex rhs);
```

1 Returns `double_complex(lhs) -= rhs`.

17.5.7.2.7 operator-(double_complex, double) [lib.op-.dc.d]

```
double_complex operator-(double_complex lhs, double rhs);
```

1 Returns `double_complex(lhs) -= double_complex(rhs)`.

17.5.7.2.8 operator-(double, double_complex) [lib.op-.d.dc]

```
double_complex operator-(double lhs, double_complex rhs);
```

1 Returns `double_complex(lhs) -= rhs`.

17.5.7.2.9 operator*(double_complex, double_complex) [lib.op*.dc.dc]

```
double_complex operator*(double_complex lhs, double_complex rhs);
```

1 Returns `double_complex(lhs) *= rhs`.

17.5.7.2.10 operator*(double_complex, double) [lib.op*.dc.d]

```
double_complex operator*(double_complex lhs, double rhs);
```

1 Returns `double_complex(lhs) *= double_complex(rhs)`.

17.5.7.2.11 operator*(double, double_complex) [lib.op*.d.dc]

```
double_complex operator*(double lhs, double_complex rhs);
```

1 Returns `double_complex(lhs) *= rhs`.

17.5.7.2.12 operator/(double_complex, double_complex) [lib.op/.dc.dc]

```
double_complex operator/(double_complex lhs, double_complex rhs);
```

1 Returns `double_complex(lhs) /= rhs`.

17.5.7.2.13 operator/(double_complex, double) [lib.op/.dc.d]

```
double_complex operator/(double_complex lhs, double rhs);
```

1 Returns `double_complex(lhs) /= double_complex(rhs)`.

17.5.7.2.14 operator/(double, double_complex) [lib.op/.d.dc]

```
double_complex operator/(double lhs, double_complex rhs);
```

1 Returns `double_complex(lhs) /= rhs`.

17.5.7.2.15 operator+(double_complex) [lib.op+.dc]

```
double_complex operator+(double_complex lhs);
```

1 Returns `double_complex(lhs)`.

17.5.7.2.16 operator-(double_complex) [lib.op-.dc]

```
double_complex operator-(double_complex lhs);
```

1 Returns `double_complex(-real(lhs), -imag(lhs))`.

17.5.7.2.17 operator==(double_complex, double_complex) [lib.op==.dc.dc]

```
bool operator==(double_complex lhs, double_complex rhs);
```

1 Returns `real(lhs) == real(rhs) && imag(lhs) == imag(rhs)`.

17.5.7.2.18 operator==(double_complex, double) [lib.op==.dc.d]

```
bool operator==(double_complex lhs, double rhs);
```

1 Returns `real(lhs) == rhs && imag(lhs) == 0`.

17.5.7.2.19 operator==(double, double_complex) [lib.op==.d.dc]

```
bool operator==(double lhs, double_complex rhs);
```

1 Returns `lhs == real(rhs) && imag(rhs) == 0`.

17.5.7.2.20 operator!=(double_complex, double_complex) [lib.op!=.dc.dc]

```
bool operator!=(double_complex lhs, double_complex rhs);
```

1 Returns `real(lhs) != real(rhs) || imag(lhs) != imag(rhs)`.

17.5.7.2.21 operator!=(double_complex, double) [lib.op!=.dc.d]

```
bool operator!=(double_complex lhs, double rhs);
```

1 Returns `real(lhs) != rhs || imag(lhs) != 0`.

17.5.7.2.22 operator!=(double, double_complex) [lib.op!=.d.dc]

```
bool operator!=(double lhs, double_complex rhs);
```

1 Returns `lhs != real(rhs) || imag(rhs) != 0`.

17.5.7.2.23 operator>>(istream&, double_complex&) [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*.

17.5.7.2.24 operator<<(ostream&, double_complex) [lib.ins.dc]

```
ostream& operator<<(ostream& os, double_complex x);
```

1 Returns `os << '(' << real(x) << ',' << imag(x) << ')'`.

17.5.7.2.25 abs(double_complex) [lib.abs.dc]

```
double abs(double_complex x);
```

1 Returns the magnitude of *x*.

17.5.7.2.26 arg(double_complex) [lib.arg.dc]

```
double arg(double_complex x);
```

1 Returns the phase angle of *x*.

17.5.7.2.27 conj(double_complex) [lib.conj.dc]

```
double_complex conj(double_complex x);
```

1 Returns the conjugate of *x*.

17.5.7.2.28 cos(double_complex) [lib.cos.dc]

```
double_complex cos(double_complex x);
```

1 Returns the cosine of *x*.

17.5.7.2.29 cosh(double_complex) [lib.cosh.dc]

```
double_complex cosh(double_complex x);
```

1 Returns the hyperbolic cosine of *x*.

17.5.7.2.30 exp(double_complex) [lib.exp.dc]

```
double_complex exp(double_complex x);
```

1 Returns the exponential of *x*.

17.5.7.2.31 imag(double_complex) [lib.imag.dc]

```
double imag(double_complex x);
```

1 Returns the imaginary part of *x*.

17.5.7.2.32 log(double_complex) [lib.log.dc]

```
double_complex log(double_complex x);
```

1 Returns the logarithm of *x*.

17.5.7.2.33 norm(double_complex) [lib.norm.dc]

```
double norm(double_complex x);
```

1 Returns the squared magnitude of *x*.

17.5.7.2.34 polar(double, double) [lib.polar.d.d]

```
double_complex polar(double rho, double theta);
```

1 Returns the `double_complex` value corresponding to a complex number whose magnitude is *rho* and whose phase angle is *theta*.

17.5.7.2.35 pow(double_complex, double_complex) [lib.pow.dc.dc]

```
double_complex pow(double_complex x, double_complex y);
```

1 Returns *x* raised to the power *y*.

17.5.7.2.36 `pow(double_complex, double)` [lib.pow.dc.d]

```
double_complex pow(double_complex x, double y);
```

1 Returns *x* raised to the power *y*.

17.5.7.2.37 `pow(double_complex, int)` [lib.pow.dc.i]

```
double_complex pow(double_complex x, int y);
```

1 Returns *x* raised to the power *y*.

17.5.7.2.38 `pow(double, double_complex)` [lib.pow.d.dc]

```
double_complex pow(double x, double_complex y);
```

1 Returns *x* raised to the power *y*.

17.5.7.2.39 `real(double_complex)` [lib.real.dc]

```
double real(double_complex x);
```

1 Returns the real part of *x*.

17.5.7.2.40 `sin(double_complex)` [lib.sin.dc]

```
double_complex sin(double_complex x);
```

1 Returns the sine of *x*.

17.5.7.2.41 `sinh(double_complex)` [lib.sinh.dc]

```
double_complex sinh(double_complex x);
```

1 Returns the hyperbolic sine of *x*.

17.5.7.2.42 `sqrt(double_complex)` [lib.sqrt.dc]

```
double_complex sqrt(double_complex x);
```

1 Returns the square root of *x*.

17.5.7.3 Complex numbers with `long double` precision [lib.complex.with.ld]

17.5.7.3.1 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.

17.5.7.3.1.1 **long_double_complex::long_double_complex(long double, long double)** [lib.cons.long.double.complex.ld.ld]

```
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*.

17.5.7.3.1.2 **long_double_complex::long_double_complex(float_complex&)** [lib.cons.long.double.complex.fc]

```
long_double_complex(float_complex& rhs);
```

1 Constructs an object of class `long_double_complex`, initializing *re* to `(long double)real(`*rhs*`)` and *im* to `(long double)imag(`*rhs*`)`.

17.5.7.3.1.3 **long_double_complex::long_double_complex(double_complex&)** [lib.cons.long.double.complex.dc]

```
long_double_complex(double_complex& rhs);
```

1 Constructs an object of class `long_double_complex`, initializing *re* to `(long double)real(`*rhs*`)` and *im* to `(long double)imag(`*rhs*`)`.

17.5.7.3.1.4 **operator+=(long_double_complex)** [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`. The function returns `*this`.

17.5.7.3.1.5 **operator-=(long_double_complex)** [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`. The function returns `*this`.

17.5.7.3.1.6 **operator*=(long_double_complex)** [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`. The function returns `*this`.

17.5.7.3.1.7 operator/=(long_double_complex) [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`. The function returns `*this`.

17.5.7.3.2 operator+(long_double_complex, long_double_complex) [lib.op+.ldc.ldc]

```
long_double_complex operator+(long_double_complex lhs,
        long_double_complex rhs);
```

1 Returns `long_double_complex(lhs) += rhs`.

17.5.7.3.3 operator+(long_double_complex, long double) [lib.op+.ldc.ld]

```
long_double_complex operator+(long_double_complex lhs,
        long double rhs);
```

1 Returns `long_double_complex(lhs) += long_double_complex(rhs)`.

17.5.7.3.4 operator+(long double, long_double_complex) [lib.op+.ld.ldc]

```
long_double_complex operator+(long double lhs,
        long_double_complex rhs);
```

1 Returns `long_double_complex(lhs) += rhs`.

17.5.7.3.5 operator-(long_double_complex, long_double_complex) [lib.op-.ldc.ldc]

```
long_double_complex operator-(long_double_complex lhs,
        long_double_complex rhs);
```

1 Returns `long_double_complex(lhs) -= rhs`.

17.5.7.3.6 operator-(long_double_complex, long double) [lib.op-.ldc.ld]

```
long_double_complex operator-(long_double_complex lhs,
        long double rhs);
```

1 Returns `long_double_complex(lhs) -= long_double_complex(rhs)`.

17.5.7.3.7 operator-(long double, long_double_complex) [lib.op-.ld.ldc]

```
long_double_complex operator-(long double lhs,
        long_double_complex rhs);
```

1 Returns `long_double_complex(lhs) -= rhs`.

17.5.7.3.8 operator*(long_double_complex, long_double_complex) [lib.op*.ldc.ldc]

```
long_double_complex operator*(long_double_complex lhs,
        long_double_complex rhs);
```

1 Returns `long_double_complex(lhs) *= rhs`.

17.5.7.3.9 operator*(long_double_complex, long double) [lib.op*.ldc.ld]

```
long_double_complex operator*(long_double_complex lhs,
        long double rhs);
```

1 Returns long_double_complex(*lhs*) *= long_double_complex(*rhs*).

17.5.7.3.10 operator*(long double, long_double_complex) [lib.op*.ld.ldc]

```
long_double_complex operator*(long double lhs,
        long_double_complex rhs);
```

1 Returns long_double_complex(*lhs*) *= *rhs*.

17.5.7.3.11 operator/(long_double_complex, long_double_complex) [lib.op/.ldc.ldc]

```
long_double_complex operator/(long_double_complex lhs,
        long_double_complex rhs);
```

1 Returns long_double_complex(*lhs*) /= *rhs*.

17.5.7.3.12 operator/(long_double_complex, long double) [lib.op/.ldc.ld]

```
long_double_complex operator/(long_double_complex lhs,
        long double rhs);
```

1 Returns long_double_complex(*lhs*) /= long_double_complex(*rhs*).

17.5.7.3.13 operator/(long double, long_double_complex) [lib.op/.ld.ldc]

```
long_double_complex operator/(long double lhs,
        long_double_complex rhs);
```

1 Returns long_double_complex(*lhs*) /= *rhs*.

17.5.7.3.14 operator+(long_double_complex) [lib.op+.ldc]

```
long_double_complex operator+(long_double_complex lhs);
```

1 Returns long_double_complex(*lhs*).

17.5.7.3.15 operator-(long_double_complex) [lib.op-.ldc]

```
long_double_complex operator-(long_double_complex lhs);
```

1 Returns long_double_complex(-real(*lhs*), -imag(*lhs*)).

17.5.7.3.16 operator==(long_double_complex, long_double_complex) [lib.op==.ldc.ldc]

```
bool operator==(long_double_complex lhs, long_double_complex rhs);
```

1 Returns real(*lhs*) == real(*rhs*) && imag(*lhs*) == imag(*rhs*).

17.5.7.3.17 `operator==(long_double_complex, long double)` [lib.op==.ldc.ld]

```
bool operator==(long_double_complex lhs, long double rhs);
```

1 Returns `real(lhs) == rhs && imag(lhs) == 0`.

17.5.7.3.18 `operator==(long double, long_double_complex)` [lib.op==.ld.ldc]

```
bool operator==(long double lhs, long_double_complex rhs);
```

1 Returns `lhs == real(rhs) && imag(rhs) == 0`.

17.5.7.3.19 `operator!=(long_double_complex, long_double_complex)` [lib.op!=.ldc.ldc]

```
bool operator!=(long_double_complex lhs, long_double_complex rhs);
```

1 Returns `real(lhs) != real(rhs) || imag(lhs) != imag(rhs)`.

17.5.7.3.20 `operator!=(long_double_complex, long double)` [lib.op!=.ldc.ld]

```
bool operator!=(long_double_complex lhs, long double rhs);
```

1 Returns `real(lhs) != rhs || imag(lhs) != 0`.

17.5.7.3.21 `operator!=(long double, long_double_complex)` [lib.op!=.ld.ldc]

```
bool operator!=(long double lhs, long_double_complex rhs);
```

1 Returns `lhs != real(rhs) || imag(rhs) != 0`.

17.5.7.3.22 `operator>>(istream&, long_double_complex&)` [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*.

17.5.7.3.23 `operator<<(ostream&, long_double_complex)` [lib.ins.ldc]

```
ostream& operator<<(ostream& os, long_double_complex x);
```

1 Returns `os << '(' << real(x) << ',' << imag(x) << ')'`.

17.5.7.3.24 `abs(long_double_complex)` [lib.abs.ldc]

```
long double abs(long_double_complex x);
```

1 Returns the magnitude of *x*.

17.5.7.3.25 `arg(long_double_complex)` [lib.arg.ldc]

```
long double arg(long_double_complex x);
```

1 Returns the phase angle of *x*.

17.5.7.3.26 `conj(long_double_complex)` [lib.conj.ldc]

```
long_double_complex conj(long_double_complex x);
```

1 Returns the conjugate of *x*.

17.5.7.3.27 `cos(long_double_complex)` [lib.cos.ldc]

```
long_double_complex cos(long_double_complex x);
```

1 Returns the cosine of *x*.

17.5.7.3.28 `cosh(long_double_complex)` [lib.cosh.ldc]

```
long_double_complex cosh(long_double_complex x);
```

1 Returns the hyperbolic cosine of *x*.

17.5.7.3.29 `exp(long_double_complex)` [lib.exp.ldc]

```
long_double_complex exp(long_double_complex x);
```

1 Returns the exponential of *x*.

17.5.7.3.30 `imag(long_double_complex)` [lib.imag.ldc]

```
long double imag(long_double_complex x);
```

1 Returns the imaginary part of *x*.

17.5.7.3.31 `log(long_double_complex)` [lib.log.ldc]

```
long_double_complex log(long_double_complex x);
```

1 Returns the logarithm of *x*.

17.5.7.3.32 `norm(long_double_complex)` [lib.norm.ldc]

```
long double norm(long_double_complex x);
```

1 Returns the squared magnitude of *x*.

17.5.7.3.33 `polar(long double, long double)` [lib.polar.ld.ld]

```
long_double_complex polar(long double rho, long double theta);
```

1 Returns the `long_double_complex` value corresponding to a complex number whose magnitude is *rho* and whose phase angle is *theta*.

17.5.7.3.34 pow(long_double_complex, long_double_complex) [lib.pow.ldc.ldc]

```
long_double_complex pow(long_double_complex x, long_double_complex y);
```

1 Returns *x* raised to the power *y*.

17.5.7.3.35 pow(long_double_complex, long double) [lib.pow.ldc.ld]

```
long_double_complex pow(long_double_complex x, long double y);
```

1 Returns *x* raised to the power *y*.

17.5.7.3.36 pow(long_double_complex, int) [lib.pow.ldc.i]

```
long_double_complex pow(long_double_complex x, int y);
```

1 Returns *x* raised to the power *y*.

17.5.7.3.37 pow(long double, long_double_complex) [lib.pow.ld.ldc]

```
long_double_complex pow(long double x, long_double_complex y);
```

1 Returns *x* raised to the power *y*.

17.5.7.3.38 real(long_double_complex) [lib.real.ldc]

```
long double real(long_double_complex x);
```

1 Returns the real part of *x*.

17.5.7.3.39 sin(long_double_complex) [lib.sin.ldc]

```
long_double_complex sin(long_double_complex x);
```

1 Returns the sine of *x*.

17.5.7.3.40 sinh(long_double_complex) [lib.sinh.ldc]

```
long_double_complex sinh(long_double_complex x);
```

1 Returns the hyperbolic sine of *x*.

17.5.7.3.41 sqrt(long_double_complex) [lib.sqrt.ldc]

```
long_double_complex sqrt(long_double_complex x);
```

1 Returns the square root of *x*.

17.5.8 Header <objcpy> [lib.header.objcpy]

1 The header <objcpy> defines several template functions that copy, construct, and destroy arrays of objects.

17.5.8.1 Template function objcpy<*T*>(*T**, const *T**, size_t) [lib.template.objcpy.t]

```
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 The function returns *dest*.

17.5.8.2 Template function `objmove<T>(T*, T*, size_t)` [lib.template.objmove.t]

```
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 The function returns *dest*.

17.5.8.3 Template function `objcpy<T>(void*, const T*, size_t)` [lib.template.objcpy.v]

```
template<class T> T* objcpy(void* dest, const T* src, size_t n);
```

1 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.

2 The function returns `(T*)dest`.

17.5.8.4 Template function `objmove<T>(void*, T*, size_t)` [lib.template.objmove.v]

```
template<class T> T* objmove(void* dest, T* src, size_t n);
```

1 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.

2 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.

3 The order in which elements are constructed or destroyed is otherwise unspecified.

4 The function returns `(T*)dest`.

17.5.8.5 Template function `objconstruct<T>(void*, size_t)` [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 The function returns `(T*)dest`.

17.5.8.6 Template function `objdestroy<T>(T*, size_t)` [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 The function returns `(void*)dest`.

17.5.9 Header `<locale>` [lib.header.locale]

1 The header `<locale>` defines two classes and several functions that encapsulate and manipulate the information peculiar to a locale.

2 In this subclause, the type name `struct tm` is an incomplete type that is defined in `<ctime>`.

17.5.9.1 Class `locale` [lib.locale]

```
class locale {
public:
        typedef T1 category;
        static const category COLLATE;
        static const category CTYPE;
        static const category MESSAGES;
        static const category MONETARY;
        static const category NUMERIC;
        static const category TIME;
        static const category ALL;
        typedef T2 ctype;
        static const ctype ALPHA;
        static const ctype CNTRL;
        static const ctype DIGIT;
        static const ctype LOWER;
        static const ctype PRINT;
        static const ctype PUNCT;
        static const ctype SPACE;
        static const ctype UPPER;
        static const ctype XDIGIT;
        static const ctype ALNUM;
        static const ctype GRAPH;
        static const ctype NO_MATCH;
        typedef T3 dateorder;
        static const dateorder DMY;
        static const dateorder MDY;
        static const dateorder NO_ORDER;
        static const dateorder YDM;
        static const dateorder YMD;
        typedef T4 moneysymbol;
        static const moneysymbol LOCAL;
        static const moneysymbol INTL;
        static const moneysymbol NONE;
        typedef T5 totype;
        static const totype DOWN;
        static const totype NO_CHANGE;
        static const totype UP;
```

```
class virtuals {
protected:
        virtuals(size_t refs_arg);
        virtual ~virtuals();
        virtual virtuals* copybut(const char*name, category cat)
                const;
        virtual void name(ostream& os) const = 0;
        virtual bool equal(const virtuals* vir_arg, category cat)
                const;
        virtual void insert(ostream& os, bool n) const;
        virtual void insert(ostream& os, long n) const;
        virtual void insert(ostream& os, unsigned long n) const;
        virtual void insert(ostream& os, double n) const;
        virtual void insert(ostream& os, long double n) const;
        virtual void extract(istream& is, bool& n) const;
        virtual void extract(istream& is, long& n) const;
        virtual void extract(istream& is, unsigned long& n) const;
        virtual void extract(istream& is, double& n) const;
        virtual void extract(istream& is, long double& n) const;
        virtual int narrow(wchar_t wc, char& c) const;
        virtual int widen(char c, wchar_t& wc) const;
        virtual bool is(ctype mask, wchar_t wc) const;
        virtual size_t is(const wchar_t* src, size_t n, ctype* dest)
                const;
        virtual ctype namedctype(const char *name) const;
        virtual char to(totype way, char c) const;
        virtual char to(totype way, wchar_t c) const;
        virtual size_t to(totype way, char* s, size_t n) const;
        virtual size_t to(totype way, wchar_t* s, size_t n) const;
        virtual totype namedto(const char *name) const;
        virtual int collate(const char* s1, size_t n1,
                const char* s2, size_t n2) const;
        virtual int collate(const wchar_t* s1, size_t n1,
                const wchar_t* s2, size_t n2) const;
        virtual size_t transform(ostream& os, const char* s,
                size_t n) const;
        virtual size_t transform(ostream& os, const wchar_t* s,
                size_t n) const;
        virtual long hash(const char* s, size_t n) const;
        virtual long hash(const wchar_t* s, size_t n) const;
        virtual void insert(ostream& os, const struct tm* t,
                char code); const
        virtual void extracttime(istream& is, struct tm* t) const;
        virtual void extractdate(istream& is, struct tm* t) const;
        virtual void extractweekday(istream& is, struct tm* t) const;
        virtual void extractmonthname(istream& is, struct tm* t)
                const;
        virtual dateorder date_order() const;
        virtual void insert(ostream& os, double units,
                moneysymbol sym) const;
        virtual void insert(ostream& os, char* digits,
                moneysymbol sym) const;
        virtual void extractmoney(istream& is, double& units,
                moneysymbol sym) const;
        virtual void extractmoney(istream& is, ostream& digits,
                moneysymbol sym) const;
        virtual int moneyfracdigits(moneysymbol sym) const;
        const ctype* ctypetable;
private:
        virtuals(const virtuals&);       // not defined
        const virtuals& operator=(const virtuals&);     // not defined
        void add_reference();
```

```
                void remove_reference();
//              size_t refs;      exposition only
        };
```

```
locale(const char* name);
locale(virtuals* vir_arg);
locale(const locale& loc, const char* name, category cat);
~locale();
bool ok() const;
bool operator==(const locale& rhs) const;
bool operator!=(const locale& rhs) const;
bool equal(const locale& rhs, category cat = ALL) const;
void insert(ostream& os, bool n) const;
void insert(ostream& os, long n) const;
void insert(ostream& os, unsigned long n) const;
void insert(ostream& os, double n) const;
void insert(ostream& os, long double n) const;
void extract(istream& is, bool& n) const;
void extract(istream& is, long& n) const;
void extract(istream& is, unsigned long& n) const;
void extract(istream& is, double& n) const;
void extract(istream& is, long double& n) const;
int narrow(wchar_t wc, char& c) const;
int widen(char c, wchar_t& wc) const;
bool is(ctype mask, char c) const;
bool is(ctype mask, unsigned char c) const;
bool is(ctype mask, signed char c) const;
bool is(ctype mask, int c) const;
bool is(ctype mask, wchar_t wc) const;
size_t is(const char* src, size_t n, ctype* dest) const;
size_t is(const wchar_t* src, size_t n, ctype* dest) const;
ctype namedctype(const char *name) const;
char to(totype way, char c) const;
char to(totype way, unsigned char c) const;
char to(totype way, signed char c) const;
char to(totype way, wchar_t c) const;
size_t to(totype way, char* s, size_t n) const;
size_t to(totype way, wchar_t* s, size_t n) const;
totype namedto(const char *name) const;
int collate(const char* s1, size_t n1,
                const char* s2, size_t n2) const;
int collate(const wchar_t* s1, size_t n1,
                const wchar_t* s2, size_t n2) const;
size_t transform(ostream& os, const char* s, size_t n) const;
size_t transform(ostream& os, const wchar_t* s, size_t n) const;
long hash(const char* s, size_t n) const;
long hash(const wchar_t* s, size_t n) const;
void insert(ostream& os, const struct tm* t, const char* fmt)
        const;
void insert(ostream& os, const struct tm* t, char code); const
void extracttime(istream& is, struct tm* t) const;
void extractdate(istream& is, struct tm* t) const;
void extractweekday(istream& is, struct tm* t) const;
void extractmonthname(istream& is, struct tm* t) const;
dateorder date_order() const;
void insert(ostream& os, double units, moneysymbol sym) const;
void insert(ostream& os, char* digits, moneysymbol sym) const;
void extractmoney(istream& is, double& units, moneysymbol sym)
                const;
void extractmoney(istream& is, ostream& digits, moneysymbol sym)
                const;
int moneyfracdigits(moneysymbol sym) const;
static locale global();
static locale global(const locale& loc);
static const locale& classic();
static const locale& transparent();
```

```
private:
        void name(ostream& os) const;
//      virtuals* vir;   exposition only
};
```

1 The class `locale` encapsulates the information peculiar to a locale. It defines several member types:

— the bitmask types `category` and `ctype`;

— the enumerated types `dateorder`, `moneysymbol`, and `totype`;

— the class `virtuals`.

2 The macro `UCHAR_MAX` is defined in `<ciimits>`.

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

— `virtuals* vir`, points to the object of class `virtuals` that describes a specific locale.

17.5.9.1.1 Type `locale::category` [lib.locale::category]

```
typedef T1 category;
```

1 The type `category` is a bitmask type (indicated here as *T1*) with the elements (corresponding to macros defined in `<clocale>`:

— `COLLATE`, set to select the category `LC_COLLATE`;

— `CTYPE`, set to select the category `LC_CTYPE`;

— `MESSAGES`, set to select the category `LC_MESSAGES`;

— `MONETARY`, set to select the category `LC_MONETARY`;

— `NUMERIC`, set to select the category `LC_NUMERIC`;

— `TIME`, set to select the category `LC_TIME`.

2 Type `category` also defines the constant:

— `ALL`, the union of all elements of the type `category` (corresponds to `LC_ALL`).

17.5.9.1.2 Type `locale::ctype` [lib.locale::ctype]

```
typedef T2 ctype;
```

1 The type `ctype` is a bitmask type (indicated here as *T2*) with the elements (corresponding to functions declared in `<cctype>`:

— `ALPHA`, set to match characters for which `isalpha(int)` returns a nonzero value;

— `CNTRL`, set to match characters for which `iscntrl(int)` returns a nonzero value;

— `DIGIT`, set to match characters for which `isdigit(int)` returns a nonzero value;

— `LOWER`, set to match characters for which `islower(int)` returns a nonzero value;

— `PRINT`, set to match characters for which `isprint(int)` returns a nonzero value;

— `PUNCT`, set to match characters for which `ispunct(int)` returns a nonzero value;

— `SPACE`, set to match characters for which `isspace(int)` returns a nonzero value;

— `UPPER`, set to match characters for which `isupper(int)` returns a nonzero value;

— `XDIGIT`, set to match characters for which `isxdigit(int)` returns a nonzero value;

2 Type `ctype` also defines the constants:

— `ALNUM`, the value `ALPHA | DIGIT` (corresponds to `isalnum(int)`).

— `GRAPH`, the value `ALPHA | DIGIT | PUNCT` (corresponds to `isgraph(int)`).

— `NO_MATCH`, the value zero.

17.5.9.1.3 Type `locale::dateorder` [lib.locale::dateorder]

```
typedef T3 dateorder;
```

1 The type `dateorder` is an enumerated type (indicated here as *T3*) with the elements:

— `DMY`, to specify that date components appear in the order date, month, and year;

— `MDY`, to specify that date components appear in the order month, date, and year;

— `NO_ORDER`, to specify that the order of appearance of date components is not meaningful;

— `YDM`, to specify that date components appear in the order year, date, and month;

— `YMD`, to specify that date components appear in the order year, month, and date.

17.5.9.1.4 Type `locale::moneysymbol` [lib.locale::moneysymbol]

```
typedef T4 moneysymbol;
```

1 The type `moneysymbol` is an enumerated type (indicated here as *T4*) with the elements (corresponding to members of `struct lconv` defined in `<clocale>`:

— `LOCAL`, to specify that the currency symbol should be specified by the member `currency_symbol`;

— `INTL`, to specify that the currency symbol should be specified by the member `int_curr_symbol`;

— `NONE`, to specify no currency symbol.

17.5.9.1.5 Type `locale::totype` [lib.locale::totype]

```
typedef T5 totype;
```

1 The type `totype` is an enumerated type (indicated here as *T5*) with the elements (corresponding to functions declared in `<cctype>`:

— `DOWN`, to specify translation of upper case characters to lower case (corresponds to `tolower(int)`);

— `NO_CHANGE`, to specify no translation;

— `UP`, to specify translation of lower case characters to upper case (corresponds to `toupper(int)`);

17.5.9.1.6 Class `locale::virtuals` | [lib.locale::virtuals]

```
class virtuals {
protected:
        virtuals(size_t refs_arg);
        virtual ~virtuals();
        virtual virtuals* copybut(const char*name, category cat) const;
        virtual void name(ostream& os) const = 0;
        virtual bool equal(const virtuals* vir_arg, category cat) const;
        virtual void insert(ostream& os, bool n) const;
        virtual void insert(ostream& os, long n) const;
        virtual void insert(ostream& os, unsigned long n) const;
        virtual void insert(ostream& os, double n) const;
        virtual void insert(ostream& os, long double n) const;
        virtual void extract(istream& is, bool& n) const;
        virtual void extract(istream& is, long& n) const;
        virtual void extract(istream& is, unsigned long& n) const;
        virtual void extract(istream& is, double& n) const;
        virtual void extract(istream& is, long double& n) const;
        virtual int narrow(wchar_t wc, char& c) const;
        virtual int widen(char c, wchar_t& wc) const;
        virtual bool is(ctype mask, wchar_t wc) const;
        virtual size_t is(const wchar_t* src, size_t n, ctype* dest)
                const;
        virtual ctype namedctype(const char *name) const;
        virtual char to(totype way, char c) const;
        virtual char to(totype way, wchar_t c) const;
        virtual size_t to(totype way, char* s, size_t n) const;
        virtual size_t to(totype way, wchar_t* s, size_t n) const;
        virtual totype namedto(const char *name) const;
        virtual int collate(const char* s1, size_t n1,
                const char* s2, size_t n2) const;
        virtual int collate(const wchar_t* s1, size_t n1,
                const wchar_t* s2, size_t n2) const;
        virtual size_t transform(ostream& os, const char* s, size_t n)
                const;
        virtual size_t transform(ostream& os, const wchar_t* s, size_t n)
                const;
        virtual long hash(const char* s, size_t n) const;
        virtual long hash(const wchar_t* s, size_t n) const;
        virtual void insert(ostream& os, const struct tm* t, char code)
                const;
        virtual void extracttime(istream& is, struct tm* t) const;
        virtual void extractdate(istream& is, struct tm* t) const;
        virtual void extractweekday(istream& is, struct tm* t) const;
        virtual void extractmonthname(istream& is, struct tm* t) const;
        virtual dateorder date_order() const;
        virtual void insert(ostream& os, double units, moneysymbol sym)
                const;
        virtual void insert(ostream& os, char* digits, moneysymbol sym)
                const;
        virtual void extractmoney(istream& is, double& units,
                moneysymbol sym) const;
        virtual void extractmoney(istream& is, ostream& digits,
                moneysymbol sym) const;
        virtual int moneyfracdigits(moneysymbol sym) const;
        const ctype* ctypetable;
private:
        virtuals(const virtuals&);      // not defined
        const virtuals& operator=(const virtuals&);     // not defined
        void add_reference();
        void remove_reference();
//      size_t refs;     exposition only
};
```

1 The class `virtuals` describes a specific locale. The default behavior of all virtual member functions is to perform any locale-specific behavior consistent with that required for the `"C"` locale.

2 The maintained data is:

— `const ctype*` *`ctypetable`*, points to the initial element of an array of `UCHAR_MAX + 1` object of type `const ctype` that describes the properties of all values of type `unsigned char`.

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

— `size_t` *`refs`*, counts the number of references from other objects to the object of class `virtuals`.

4 Objects of class `locale` alter *`refs`* to match the number of `locale::`*`vir`* pointers that designate an object of class `virtuals`.

17.5.9.1.6.1 `locale::virtuals::virtuals(size_t)` [lib.cons.locale::virtuals.refs]

```
virtuals(size_t refs_arg);
```

1 Construct an object of class `virtuals`, initializing *`ctypetable`* to values suitable for the `"C"` locale and *`refs`* to *`refs_arg`*.

17.5.9.1.6.2 `locale::virtuals::~virtuals()` [lib.des.locale::virtuals]

```
virtual ~virtuals();
```

1 Destroys an object of class `virtuals`.

17.5.9.1.6.3 `locale::virtuals::copybut(const char*, category)` [lib.locale::copybut]

```
virtual virtuals* copybut(const char*name, category cat) const;
```

1 Creates an object of class `virtuals` by evaluating the expression *`vir`* `= new (virtuals)`, where *`vir`* is an object of type pointer to `virtuals`. The function copies from `*this` the locale-specific behavior for categories not selected by *`cat`*. Otherwise, the locale-specific behavior is the same as for the global locale established by the call `setlocale(`*`name`*`, `*`cat`*`)`.

2 The function returns *`vir`*.

17.5.9.1.6.4 `locale::virtuals::name(ostream&)` [lib.locale::virtuals::name]

```
virtual void name(ostream& os) const = 0;
```

1 A pure virtual function, inserts in *`os`* the name of the locale described by `*this`.

17.5.9.1.6.5 `locale::virtuals::equal(const virtuals*, category)` [lib.locale::virtuals::equal]

```
virtual bool equal(const virtuals* vir_arg, category cat) const;
```

1 Returns a nonzero value if the locale described by `*`*`vir_arg`* is the same as the locale described by `*this` for the categories selected by *`cat`*. In particular, the call

```
locale("C").equal(locale::classic())
```

2 is nonzero.

17.5.9.1.6.6 `locale::virtuals::insert(ostream&, bool)` [lib.locale::virtuals::insert.bool]

```
virtual void insert(ostream& os, bool n) const;
```

1 Behaves the same as `os << n`, with the locale-specific behavior described by `*this`.

17.5.9.1.6.7 `locale::virtuals::insert(ostream&, long)` [lib.locale::virtuals::insert.li]

```
virtual void insert(ostream& os, long n) const;
```

1 Behaves the same as `os << n`, with the locale-specific behavior described by `*this`.

17.5.9.1.6.8 `locale::virtuals::insert(ostream&, unsigned long)` [lib.locale::virtuals::insert.uli]

```
virtual void insert(ostream& os, unsigned long n) const;
```

1 Behaves the same as `os << n`, with the locale-specific behavior described by `*this`.

17.5.9.1.6.9 `locale::virtuals::insert(ostream&, double)` [lib.locale::virtuals::insert.d]

```
virtual void insert(ostream& os, double n) const;
```

1 Behaves the same as `os << n`, with the locale-specific behavior described by `*this`.

17.5.9.1.6.10 `locale::virtuals::insert(ostream&, long double)` [lib.locale::virtuals::insert.ld]

```
virtual void insert(ostream& os, long double n) const;
```

1 Behaves the same as `os << n`, with the locale-specific behavior described by `*this`.

17.5.9.1.6.11 `locale::virtuals::extract(instream&, bool&)` [lib.locale::virtuals::extract.bool]

```
virtual void extract(istream& is, bool& n) const;
```

1 Behaves the same as `os >> n`, with the locale-specific behavior described by `*this`.

17.5.9.1.6.12 `locale::virtuals::extract(istream&, long&)` [lib.locale::virtuals::extract.li]

```
virtual void extract(istream& is, long& n) const;
```

1 Behaves the same as `os >> n`, with the locale-specific behavior described by `*this`.

17.5.9.1.6.13 `locale::virtuals::extract(istream&, unsigned long&)` [lib.locale::virtuals::extract.uli]

```
virtual void extract(istream& is, unsigned long& n) const;
```

1 Behaves the same as `os >> n`, with the locale-specific behavior described by `*this`.

17.5.9.1.6.14 `locale::virtuals::extract(istream&, double&)` [lib.locale::virtuals::extract.d]

```
virtual void extract(istream& is, double& n) const;
```

1 Behaves the same as `os >> n`, with the locale-specific behavior described by `*this`.

17.5.9.1.6.15 `locale::virtuals::extract(istream&, long double&)` [lib.locale::virtuals::extract.ld]

```
virtual void extract(istream& is, long double& n) const;
```

1 Behaves the same as `os >> n`, with the locale-specific behavior described by `*this`.

17.5.9.1.6.16 `locale::virtuals::narrow(wchar_t, char&)` [lib.locale::virtuals::narrow]

```
virtual int narrow(wchar_t wc, char& c) const;
```

1 If `wctob(wc) == EOF`, returns zero. Otherwise, the function stores `wctob(wc)` in *c* and returns a nonzero value. The function signature `wctob(wchar_t)` is declared in `<cwchar>`.

17.5.9.1.6.17 `locale::virtuals::widen(char, wchar_t&)` [lib.locale::virtuals::widen]

```
virtual int widen(char c, wchar_t& wc) const;
```

1 If `btowc(c) == WEOF`, returns zero. Otherwise, the function stores `btowc(c)` in *wc* and returns a nonzero value. The function signature `btowc(wchar_t)` is declared, and the macro `WEOF` is defined, in `<cwchar>`.

17.5.9.1.6.18 `locale::virtuals::is(ctype, wchar_t)` [lib.locale::virtuals::is.wc]

```
virtual bool is(ctype mask, wchar_t wc) const;
```

1 Determines the `ctype` value *x* that characterizes *wc* and returns a nonzero value if `x & mask` is nonzero.

17.5.9.1.6.19 `locale::virtuals::is(const wchar_t*, size_t, ctype*)` [lib.locale::virtuals::is.wcs]

```
virtual size_t is(const wchar_t* src, size_t n, ctype* dest) const;
```

1 Assigns to `dest[I]` the `ctype` value *x* that characterizes `src[I]`, for successive non-negative values of *I* starting with zero. Assignment proceeds until either:

— *n* values have been stored;

— The value *x* equals `NO_MATCH`, in which case the value is not stored.

2 The pointer `src` shall designate the initial element of an array of *n* objects of type `wchar_t`. The pointer `dest` shall designate the initial element of an array of *n* objects of type `ctype`.

3 The function returns a count of the number of values stored.

17.5.9.1.6.20 `locale::virtuals::namedctype(const char*)` [lib.locale::virtuals::namedctype]

```
virtual ctype namedctype(const char *name) const;
```

1 Returns the `ctype` value corresponding to the NTBS *name*, or `NO_MATCH` if no corresponding value exists. The function shall return the corresponding `ctype` value for each of the NTBS arguments in the minimum set accepted by the function signature `wctype(const char*)`, declared in `<cwctype>`.

17.5.9.1.6.21 `locale::virtuals::to(totype, char)` [lib.locale::virtuals::to.c]

```
virtual char to(totype way, char c) const;
```

1 Returns the `char` value that corresponds to the mapping of *c* specified by *way*.

17.5.9.1.6.22 `locale::virtuals::to(totype, wchar_t)` [lib.locale::virtuals::to.wc]

```
virtual char to(totype way, wchar_t c) const;
```

1 Returns the `wchar_t` value that corresponds to the mapping of *c* specified by *way*.

17.5.9.1.6.23 `locale::virtuals::to(totype, char*, size_t)` [lib.locale::virtuals::to.str]

```
virtual size_t to(totype way, char* s, size_t n) const;
```

1 Assigns `to(way, s[I])` to `s[I]` for all non-negative values of *I* less than *n*. The pointer *s* shall designate the initial element of an array of *n* objects of type `char`.

2 The function returns *n*.

17.5.9.1.6.24 `locale::virtuals::to(totype, wchar_t*, size_t)` [lib.locale::virtuals::to.wcs]

```
virtual size_t to(totype way, wchar_t* s, size_t n) const;
```

1 Assigns `to(way, s[I])` to `s[I]` for all non-negative values of *I* less than *n*. The pointer *s* shall designate the initial element of an array of *n* objects of type `wchar_t`.

17.5.9.1.6.25 `locale::virtuals::namedto(const char*)` [lib.locale::virtuals::namedto]

```
virtual totype namedto(const char *name) const;
```

1 Returns the `totype` value corresponding to the NTBS *name*, or a unique if no corresponding value has been previously determined for that name. The function shall return the corresponding `totype` value for each of the NTBS arguments in the minimum set accepted by the function signature `wctrans(const char*)`, declared in `<cwctype>`.

17.5.9.1.6.26 `locale::virtuals::collate(const char*, size_t, const char*, size_t)` [lib.locale::virtuals::collate.str]

```
virtual int collate(const char* s1, size_t n1,
        const char* s2, size_t n2) const;
```

1 Returns the same value as `strcoll(s1, s2)` with null characters (conceptually) stored in `s1[n1]` and `s2[n2]`. The function signature `strcoll(const char*, const char*)` is declared in `<cstring>`. The pointer `s1` shall designate the initial element of an array of *n1* objects of type `char`. The pointer `s2` shall designate the initial element of an array of *n2* objects of type `char`.

17.5.9.1.6.27 locale::virtuals::collate(const wchar_t*, size_t, const wchar_t*, size_t) [lib.locale::virtuals::collate.wcs]

```
virtual int collate(const wchar_t* s1, size_t n1,
        const wchar_t* s2, size_t n2) const;
```

1 Returns the same value as `wcscoll(s1, s2)` with null wide characters (conceptually) stored in `s1[n1]` and `s2[n2]`. The function signature `wcscoll(const wchar_t*, const wchar_t*)` is declared in `<cwchar>`. The pointer `s1` shall designate the initial element of an array of *n1* objects of type `wchar_t`. The pointer `s2` shall designate the initial element of an array of *n2* objects of type `wchar_t`.

17.5.9.1.6.28 locale::virtuals::transform(ostream&, const char*, size_t) [lib.locale::virtuals::transform.str]

```
virtual int transform(ostream& os, const char* s, size_t n) const;
```

1 Behaves the same as `os.write(x, m = strxfrm(x, s, M))` with a null character (conceptually) stored in `s[n]`. Here, *m* is an object of type `size_t`, *x* is an array of *M* objects of type `char`, and *M* is larger than *m*. The function signature `strxfrm(char*, const char*, size_t)` is declared in `<cstring>`. The pointer `s` shall designate the initial element of an array of *n* objects of type `char`.

2 The function returns *m*.

17.5.9.1.6.29 locale::virtuals::transform(ostream&, const wchar_t*, size_t) [lib.locale::virtuals::transform.wcs]

```
virtual size_t transform(ostream& os, const wchar_t* s, size_t n)
        const;
```

1 Behaves the same as `os.write((char*)x, m = wcsxfrm(x, s, M) * sizeof (wchar_t))` with a null wide character (conceptually) stored in `s[n]`. Here, *m* is an object of type `size_t`, *x* is an array of *M* objects of type `wchar_t`, and *M* is larger than *m*. The function signature `wcsxfrm(wchar_t*, const wchar_t*, size_t)` is declared in `<cwchar>`. The pointer `s` shall designate the initial element of an array of *n* objects of type `wchar_t`.

2 The function returns *m*.

17.5.9.1.6.30 locale::virtuals::hash(const char*, size_t) [lib.locale::virtuals::hash.str]

```
virtual long hash(const char* s, size_t n) const;
```

1 Returns a value that is a function of the elements `s[I]`, for all non-negative values of *I* less than *n*. The pointer `s` shall designate the initial element of an array of *n* objects of type `char`.

17.5.9.1.6.31 locale::virtuals::hash(const wchar_t*, size_t) [lib.locale::virtuals::hash.wcs]

```
virtual long hash(const wchar_t* s, size_t n) const;
```

1 Returns a value that is a function of the elements `s[I]`, for all non-negative values of *I* less than *n*. The pointer `s` shall designate the initial element of an array of *n* objects of type `wchar_t`.

17.5.9.1.6.32 `locale::virtuals::insert(ostream&, const struct tm*, char)` [lib.locale::virtuals::insert.tm]

```
virtual void insert(ostream& os, const struct tm* t, char code) const;
```

1 Behaves the same as `os.write(x, strftime(x, fmt, M, t))`. Here, *fmt* is an array of `char` with `fmt[0] == '%'`, `fmt[1] == code`, and `fmt[2] == '\0'`; *x* is an array of *M* objects of type `char`; and *M* is large enough that the value returned by `strftime` is nonzero. The function signature `strftime(char*, size_t, const char*, struct tm*)` is declared in <ctime>.

17.5.9.1.6.33 `locale::virtuals::extracttime(istream&, struct tm*)` [lib.locale::virtuals::extracttime]

```
virtual void extracttime(istream& is, struct tm* t) const;
```

1 Extracts characters from *is* to determine the encoded time values to store in `t->tm_hour`, `t->tm_min`, and `t->tm_sec`. The character sequence inserted by `insert(os, t1, 'X')` shall be extracted by `extracttime(is, t2)` such that `t1->tm_hour == t2->tm_hour && t1->tm_min == t2->tm_min && t1->tm_hour == t2->tm_hour` is nonzero. The character sequences recognized are otherwise locale specific.

17.5.9.1.6.34 `locale::virtuals::extractdate(istream&, struct tm*)` [lib.locale::virtuals::extractdate]

```
virtual void extractdate(istream& is, struct tm* t) const;
```

1 Extracts characters from *is* to determine the encoded time values to store in `t->tm_year`, `t->tm_mon`,`t->tm_mday`, `t->tm_yday`, and `t->tm_wday`. The character sequence inserted by `insert(os, t1, 'x')` shall be extracted by `extractdate(is, t2)` such that `t1->tm_year == t2->tm_year && t1->tm_mon == t2->tm_mon && t1->tm_mday == t2->tm_mday` is nonzero. The character sequences recognized are otherwise locale specific.

17.5.9.1.6.35 `locale::virtuals::extractweekday(istream&, struct tm*)` [lib.locale::virtuals::extractweekday]

```
virtual void extractweekday(istream& is, struct tm* t) const;
```

1 Extracts characters from *is* to determine the encoded time value to store in `t->tm_wday`. The character sequence inserted by `insert(os, t1, 'A')` shall be extracted by `extractweekday(is, t2)` such that `t1->tm_wday == t2->tm_wday` is nonzero. The character sequences recognized are otherwise locale specific.

17.5.9.1.6.36 `locale::virtuals::extractmonthname(istream&, struct tm*)` [lib.locale::virtuals::extractmonthname]

```
virtual void extractmonthname(istream& is, struct tm* t) const;
```

1 Extracts characters from *is* to determine the encoded time value to store in `t->tm_mon`. The character sequence inserted by `insert(os, t1, 'B')` shall be extracted by `extractmonthname(is, t2)` such that `t1->tm_mon == t2->tm_mon` is nonzero. The character sequences recognized are otherwise locale specific.

17.5.9.1.6.37 `locale::virtuals::date_order()` [lib.locale::virtuals::date.order]

```
virtual dateorder date_order() const;
```

1 Returns the value of type `dateorder` that describes the locale-specific date order.

17.5.9.1.6.38 `locale::virtuals::insert(ostream&, double, moneysymbol)` [lib.locale::virtuals::insert.money.u]

```
virtual void insert(ostream& os, double units, moneysymbol sym) const;
```

1 Inserts in *os* a sequence of characters that represent the monetary value *units*.

2 If *sym* is `LOCAL`, the function displays the currency symbol `localeconv()->currency_symbol` (for a global locale that matches the locale designated by `*this<F25D>), and divides` *units* `by 10 raised to the power localeconv()->frac_digits. Otherwise, if` *sym* `is INTL, the function displays the currency symbol localeconv()->int_curr_symbol, and divides` *units* `by 10 raised to the power localeconv()->int_frac_digits. Otherwise, the function displays no currency symbol. The function signature localeconv() is declared in <clocale>.`

17.5.9.1.6.39 `locale::virtuals::insert(ostream&, char*, moneysymbol)` [lib.locale::virtuals::insert.money.d]

```
virtual void insert(ostream& os, char* digits, moneysymbol sym) const;
```

1 Inserts in *os* a sequence of characters that represent the monetary value of the NTBS *digits*, which shall consist only of decimal digits.

2 If *sym* is `LOCAL`, the function displays the currency symbol `localeconv()->currency_symbol` (for a global locale that matches the locale designated by `*this<F25D>), and displays localeconv()->frac_digits to the right of the monetary decimal point. Otherwise, if sym is INTL, the function displays the currency symbol localeconv()->int_curr_symbol, and displays localeconv()->int_frac_digits to the right of the monetary decimal point. Otherwise, the function displays no currency symbol. The function signature localeconv() is declared in <clocale>.`

17.5.9.1.6.40 `locale::virtuals::extractmoney(istream&, double&, moneysymbol)` [lib.locale::virtuals::extractmoney.u]

```
virtual void extractmoney(istream& is, double& units,
        moneysymbol sym) const;
```

1 Extracts characters from *is* to determine the encoded monetary value to store in *units*. The character sequence inserted by `insert(os, x, sym)` shall be extracted by `extractmoney(is, y, sym)` such that *x* `==` *y* is nonzero. The character sequences recognized are otherwise locale specific.

17.5.9.1.6.41 `locale::virtuals::extractmoney(istream&, ostream&, moneysymbol)` [lib.locale::virtuals::extractmoney.d]

```
virtual void extractmoney(istream& is, ostream& digits, moneysymbol sym) const;
```

1 Extracts characters from *is* to determine the sequence of digits to insert into *digits* to represent the monetary value. The character sequence inserted by `insert(os, x, sym)` shall be extracted by `extractmoney(is, y, sym)` such that the digit sequence in the NTBS *x* is the same as the digit

sequence inserted in the `ostream` object *y*. The character sequences recognized are otherwise locale specific.

17.5.9.1.6.42 locale::virtuals::moneyfracdigits(moneysymbol) [lib.locale::virtuals::moneyfracdigits]

```
virtual int moneyfracdigits(moneysymbol sym) const;
```

1 If *sym* is LOCAL, returns `localeconv()->frac_digits` (for a global locale that matches the locale designated by `*this<F25D>). Otherwise, if sym<F25D> is INTL, the function returns localeconv()->int_frac_digits. Otherwise, the function returns zero. The function signature localeconv() is declared in <clocale>.`

17.5.9.1.6.43 locale::virtuals::virtuals(const virtuals&) [lib.cons.locale::virtuals]

```
virtuals(const virtuals&);      // not defined
```

1 Constructs an object of class `virtuals` and initializes it by copying its argument. The Standard C++ library provides no definition for this function.[127)]

17.5.9.1.6.44 locale::virtuals::operator=(const virtuals&) [lib.locale::virtuals::op=]

```
const virtuals& operator=(const virtuals&);     // not defined
```

1 Assigns a value of class `virtuals` to `*this`. The Standard C++ library provides no definition for this function.

17.5.9.1.6.45 locale::virtuals::add_reference() [lib.locale::virtuals::add.reference]

```
void add_reference();
```

1 Adds one to *refs*.

17.5.9.1.6.46 locale::virtuals::remove_reference() [lib.locale::virtuals::remove.reference]

```
void remove_reference();
```

1 Subtracts one from *refs*. If the resulting stored value is zero, the object designated by `*this` may be deleted.

17.5.9.1.7 locale::locale(const char*) [lib.cons.locale.str]

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

1 Constructs an object of class `locale`, initializing *vir* with *localev_byname(name,* `0)`.

17.5.9.1.8 locale::locale(virtuals*) [lib.cons.locale.vir]

```
locale(virtuals* vir_arg);
```

1 Constructs an object of class `locale`, initializing *vir* with *vir_arg*.

127) An object of class `locale::virtuals` cannot be copied or assigned to.

17.5.9.1.9 `locale::locale(const locale&, const char*, category)` [lib.cons.locale.cat]

```
locale(const locale& loc, const char* name, category cat);
```

1 Constructs an object of class `locale`, initializing *`vir`* with *`loc`*`.copybut(`*`name`*`, `*`cat`*`)`.

17.5.9.1.10 `locale::~locale()` [lib.des.locale]

```
~locale();
```

1 Destroys an object of class `locale`.

17.5.9.1.11 `locale::ok()` [lib.locale::ok]

```
bool ok() const;
```

1 Returns a nonzero value if *`vir`* is not a null pointer.

17.5.9.1.12 `locale::operator==(const locale&)` [lib.locale::op==]

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

1 Returns a nonzero value if *`vir`*`->equal(`*`rhs`*`, ALL)` is nonzero.

17.5.9.1.13 `locale::operator!=(const locale&)` [lib.locale::op!=]

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

1 Returns a nonzero value if `!(*this ==` *`rhs`*`)`.

17.5.9.1.14 `locale::equal(const locale&, category)` [lib.locale::equal]

```
bool equal(const locale& rhs, category cat = ALL) const;
```

1 Returns a nonzero value if *`vir`*`->equal(`*`rhs`*`, `*`cat`*`)` is nonzero.

17.5.9.1.15 `locale::insert(ostream&, bool)` [lib.locale::insert.bool]

```
void insert(ostream& os, bool n) const;
```

1 Calls *`vir`*`->insert(`*`os`*`, `*`n`*`)`.

17.5.9.1.16 `locale::insert(ostream&, long)` [lib.locale::insert.li]

```
void insert(ostream& os, long n) const;
```

1 Calls *`vir`*`->insert(`*`os`*`, `*`n`*`)`.

17.5.9.1.17 `locale::insert(ostream&, unsigned long)` [lib.locale::insert.uli]

```
void insert(ostream& os, unsigned long n) const;
```

1 Calls *`vir`*`->insert(`*`os`*`, `*`n`*`)`.

17.5.9.1.18 locale::insert(ostream&, double) | [lib.locale::insert.d]

```
void insert(ostream& os, double n) const;
```

1 Calls *vir*->insert(*os*, *n*).

17.5.9.1.19 locale::insert(ostream&, long double) | [lib.locale::insert.ld]

```
void insert(ostream& os, long double n) const;
```

1 Calls *vir*->insert(*os*, *n*).

17.5.9.1.20 locale::extract(istream&, bool&) | [lib.locale::extract.bool]

```
void extract(istream& is, bool& n) const;
```

1 Calls *vir*->extract(*is*, *n*).

17.5.9.1.21 locale::extract(istream&, long&) | [lib.locale::extract.li]

```
void extract(istream& is, long& n) const;
```

1 Calls *vir*->extract(*is*, *n*).

17.5.9.1.22 locale::extract(istream&, unsigned long&) | [lib.locale::extract.uli]

```
void extract(istream& is, unsigned long& n) const;
```

1 Calls *vir*->extract(*is*, *n*).

17.5.9.1.23 locale::extract(istream&, double&) | [lib.locale::extract.d]

```
void extract(istream& is, double& n) const;
```

1 Calls *vir*->extract(*is*, *n*).

17.5.9.1.24 locale::extract(istream&, long double&) | [lib.locale::extract.ld]

```
void extract(istream& is, long double& n) const;
```

1 Calls *vir*->extract(*is*, *n*).

17.5.9.1.25 locale::narrow(wchar_t, char&) | [lib.locale::narrow]

```
int narrow(wchar_t wc, char& c) const;
```

1 Returns *vir*->narrow(*wc*, *c*).

17.5.9.1.26 locale::widen(char, wchar_t&) | [lib.locale::widen]

```
int widen(char c, wchar_t& wc) const;
```

1 Returns *vir*->widen(*c*, *wc*).

17.5.9.1.27 locale::is(ctype, char) | [lib.locale::is.c]

```
bool is(ctype mask, char c) const;
```

1 Returns a nonzero value if *vir*->ctypetable[(unsigned char)*c*] & *mask* is nonzero.

17.5.9.1.28 `locale::is(ctype, unsigned char)` [lib.locale::is.uc]

```
bool is(ctype mask, unsigned char c) const;
```

1 Returns a nonzero value if `vir->ctypetable[c] & mask` is nonzero.

17.5.9.1.29 `locale::is(ctype, signed char)` [lib.locale::is.sc]

```
bool is(ctype mask, signed char c) const;
```

1 Returns a nonzero value if `vir->ctypetable[(unsigned char)c] & mask` is nonzero.

17.5.9.1.30 `locale::is(ctype, int)` [lib.locale::is.i]

```
bool is(ctype mask, int c) const;
```

1 Returns a nonzero value if `(unsigned char)c == c && vir->ctypetable[(unsigned char)c] & mask` is nonzero.

17.5.9.1.31 `locale::is(ctype, wchar_t)` [lib.locale::is.wc]

```
bool is(ctype mask, wchar_t wc) const;
```

1 Returns a nonzero value if `vir->is(wc)` is nonzero.

17.5.9.1.32 `locale::is(const char*, size_t, ctype*)` [lib.locale::is.str]

```
size_t is(const char* src, size_t n, ctype* dest) const;
```

1 Returns `vir->is(src, n, dest)`.

17.5.9.1.33 `locale::is(const wchar_t*, size_t, ctype*)` [lib.locale::is.wcs]

```
size_t is(const wchar_t* src, size_t n, ctype* dest) const;
```

1 Returns `vir->is(src, n, dest)`.

17.5.9.1.34 `locale::namedctype(const char*)` [lib.locale::namedctype]

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

1 Returns `vir->namedctype(name)`.

17.5.9.1.35 `locale::to(totype, char)` [lib.locale::to.c]

```
char to(totype way, char c) const;
```

1 Returns `vir->to(way, c)`.

17.5.9.1.36 `locale::to(totype, unsigned char)` [lib.locale::to.uc]

```
char to(totype way, unsigned char c) const;
```

1 Returns `vir->to(way, (char)c)`.

17.5.9.1.37 `locale::to(totype, signed char)` [lib.locale::to.sc]

```
char to(totype way, signed char c) const;
```

1 Returns `vir->to(way, (char)c)`.

17.5.9.1.38 `locale::to(totype, wchar_t)` [lib.locale::to.wc]

```
char to(totype way, wchar_t c) const;
```

1 Returns `vir->to(way, c)`.

17.5.9.1.39 `locale::to(totype, char*, size_t)` [lib.locale::to.str]

```
size_t to(totype way, char* s, size_t n) const;
```

1 Returns `vir->to(way, s, n)`.

17.5.9.1.40 `locale::to(totype, wchar_t*, size_t)` [lib.locale::to.wcs]

```
size_t to(totype way, wchar_t* s, size_t n) const;
```

1 Returns `vir->to(way, s, n)`.

17.5.9.1.41 `locale::namedto(const char*)` [lib.locale::namedto]

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

1 Returns `vir->namedto(name)`.

17.5.9.1.42 `locale::collate(const char*, size_t, const char*, size_t)` [lib.locale::collate.str]

```
int collate(const char* s1, size_t n1, const char* s2, size_t n2)
        const;
```

1 Returns `vir->collate(s1, n1, s2, n2)`.

17.5.9.1.43 `locale::collate(const wchar_t*, size_t, const wchar_t*, size_t)` [lib.locale::collate.wcs]

```
int collate(const wchar_t* s1, size_t n1, const wchar_t* s2,
        size_t n2) const;
```

1 Returns `vir->collate(s1, n1, s2, n2)`.

17.5.9.1.44 `locale::transform(ostream&, const char*, size_t)` [lib.locale::transform.str]

```
size_t transform(ostream& os, const char* s, size_t n) const;
```

1 Returns `vir->transform(os, s, n)`.

17.5.9.1.45 `locale::transform(ostream&, const wchar_t*, size_t)` [lib.locale::transform.wcs]

```
size_t transform(ostream& os, const wchar_t* s, size_t n) const;
```

1 Returns *vir*->transform(*os*, *s*, *n*).

17.5.9.1.46 `locale::hash(const char*, size_t)` [lib.locale::hash.str]

```
long hash(const char* s, size_t n) const;
```

1 Returns *vir*->hash(*s*, *n*).

17.5.9.1.47 `locale::hash(const wchar_t*, size_t)` [lib.locale::hash.wcs]

```
long hash(const wchar_t* s, size_t n) const;
```

1 Returns *vir*->hash(*s*, *n*).

17.5.9.1.48 `locale::insert(ostream&, const struct tm*, const char*)` [lib.locale::insert.tm.str]

```
void insert(ostream& os, const struct tm* t, const char* fmt) const;
```

1 Inserts characters into *os* under control of a format string. *fmt* shall be an NTBS. The function processes each element *x* in succession, up to but not including the terminating null character.

2 For each element, if *x* is not `'%'`, or if the character *y* following *x* is not a null character, the function inserts *x* into *os*. Otherwise, the function calls insert(*os*, *t*, *y*).

17.5.9.1.49 `locale::insert(ostream&, const struct tm*, char)` [lib.locale::insert.tm]

```
void insert(ostream& os, const struct tm* t, char code) const;
```

1 Calls *vir*->insert(*os*, *t*, *code*).

17.5.9.1.50 `locale::extracttime(istream&, struct tm*)` [lib.locale::extracttime]

```
void extracttime(istream& is, struct tm* t) const;
```

1 Calls *vir*->extracttime(*is*, *t*).

17.5.9.1.51 `locale::extractdate(istream&, struct tm*)` [lib.locale::extractdate]

```
void extractdate(istream& is, struct tm* t) const;
```

1 Calls *vir*->extractdate(*is*, *t*).

17.5.9.1.52 `locale::extractweekday(istream&, struct tm*)` [lib.locale::extractweekday]

```
void extractweekday(istream& is, struct tm* t) const;
```

1 Calls *vir*->extractweekday(*is*, *t*).

17.5.9.1.53 `locale::extractmonthname(istream&, struct tm*)` [lib.locale::extractmonthname]

```
void extractmonthname(istream& is, struct tm* t) const;
```

1 Calls *vir*->extractmonthname(*is*, *t*).

17.5.9.1.54 locale::date_order() [lib.locale::date.order]

```
dateorder date_order() const;
```

1 Returns *vir*->date_order().

17.5.9.1.55 locale::insert(ostream&, double, moneysymbol) [lib.locale::insert.money.u]

```
void insert(ostream& os, double units, moneysymbol sym) const;
```

1 Calls *vir*->insert(*os*, *units*, *sym*).

17.5.9.1.56 locale::insert(ostream&, char*, moneysymbol) [lib.locale::insert.money.d]

```
void insert(ostream& os, char* digits, moneysymbol sym) const;
```

1 Calls *vir*->insert(*os*, *digits*, *sym*).

17.5.9.1.57 locale::extractmoney(istream&, double&, moneysymbol) [lib.locale::extractmoney.u]

```
void extractmoney(istream& is, double& units, moneysymbol sym) const;
```

1 Calls *vir*->extractmoney(*is*, *units*, *syn*).

17.5.9.1.58 locale::extractmoney(istream&, ostream&, moneysymbol) [lib.locale::extractmoney.d]

```
void extractmoney(istream& is, ostream& digits, moneysymbol sym)
        const;
```

1 Calls *vir*->extractmoney(*is*, *digits*, *syn*).

17.5.9.1.59 locale::moneyfracdigits(moneysymbol) [lib.locale::moneyfracdigits]

```
int moneyfracdigits(moneysymbol sym) const;
```

1 Returns *vir*->moneyfracdigits(*sym*).

17.5.9.1.60 locale::global() [lib.locale::global]

```
static locale global();
```

1 Constructs an object *new_loc* of class locale and initializes it to describe the same locale as the current global locale. The function returns *new_loc*.

17.5.9.1.61 locale::global() [lib.locale::global.loc]

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

1 Constructs an object *new_loc* of class locale and initializes it to describe the same locale as *loc*. The function then alters the global locale to match *loc* and returns *new_loc*.

17.5.9.1.62 locale::classic() [lib.locale::classic]

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

1 The function returns an object of class *locale* that describes the "C" locale.

17.5.9.1.63 locale::transparent() [lib.locale::transparent]

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

1 The function returns an object of class *locale* that, at all times, describes the global locale.

17.5.9.1.64 locale::name() [lib.locale::name]

```
void name(ostream& os) const;
```

1 Calls *vir*->name(*os*).

17.5.9.2 Class localev_byname [lib.localev.byname]

```
class localev_byname : public locale::virtuals {
public:
        localev_byname(const char* name, size_t refs);
};
```

1 The class localev_byname is derived from class locale::virtuals to assist in constructing an object that describes a named locale.

17.5.9.2.1 localev_byname::localev_byname(const char*, size_t) [lib.cons.localev.byname]

```
localev_byname(const char* name, size_t refs);
```

1 Constructs an object of class localev_byname, initializing the base class with locale::virtuals(*refs*), then altering the object to describe the locale whose name is *name*.

17.5.9.3 collate(const string&, const string&, const locale&) [lib.locale.collate.string]

```
int collate(const string& s1, const string& s2,
        const locale& loc = locale::global());
```

1 Returns *loc*.collate(*s1*.data(), *s1*.length(), *s2*.data(), *s2*.length()).

17.5.9.4 collate(const wstring&, const wstring&, const locale&) [lib.locale.collate.wstring]

```
int collate(const wstring& s1, const wstring& s2,
        const locale& loc = locale::global());
```

1 Returns *loc*.collate(*s1*.data(), *s1*.length(), *s2*.data(), *s2*.length()).

17.5.9.5 operator<<(ostream&, const locale&) [lib.locale.ins]

```
ostream& operator<<(ostream& os, const locale& loc);
```

1 A formatted output function, executes *loc.name*(*os*) and returns *os*.

17.5.9.6 `operator>>(istream&, locale&)` [lib.locale.ext]

```
istream& operator>>(istream& is, locale& loc);
```

1 A formatted input function, evaluates the expression *`is >> name`*, where *`name`* is an array of `char` large enough to hold an arbitrary locale name. If *`is.good()`* is nonzero, the function then stores *`localev_byname(name`*`, 0)` in *`loc.vir`*.

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-digit octal-digit*
`\` *octal-digit octal-digit octal-digit*

hexadecimal-escape-sequence:
`\x` *hexadecimal-digit*
hexadecimal-escape-sequence hexadecimal-digit

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

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

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

sign: one of
  `+ -`

digit-sequence:
  digit
  digit-sequence digit

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

string-literal:
  `"`*s-char-sequence*$_{opt}$`"`
  `L"`*s-char-sequence*$_{opt}$`"`

s-char-sequence:
  s-char
  s-char-sequence s-char

s-char:
  any member of the source character set except
    the double-quote `"`, backslash `\`, or new-line character
  escape-sequence

boolean-literal:
  `false`
  `true`

A.3 Basic concepts [gram.basic]

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

A.4 Expressions [gram.expr]

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

id-expression:
  unqualified-id
  qualified-id

unqualified-id:
  identifier
  operator-function-id
  conversion-function-id
  ~ *class-name*

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

postfix-expression:
  primary-expression
  postfix-expression `[` *expression* `]`
  postfix-expression `(` *expression-list*$_{opt}$ `)`
  simple-type-specifier `(` *expression-list*$_{opt}$ `)`
  postfix-expression `.` *id-expression*
  postfix-expression `->` *id-expression*
  postfix-expression `++`
  postfix-expression `--`
  `dynamic_cast <` *type-id* `> (` *expression* `)`
  `static_cast <` *type-id* `> (` *expression* `)`
  `reinterpret_cast <` *type-id* `> (` *expression* `)`
  `const_cast <` *type-id* `> (` *expression* `)`
  `typeid (` *expression* `)`
  `typeid (` *type-id* `)`

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

unary-expression:
  postfix-expression
  `++` *unary-expression*
  `--` *unary-expression*
  unary-operator cast-expression
  `sizeof` *unary-expression*
  `sizeof (` *type-id* `)`
  new-expression
  delete-expression

unary-operator: one of
  `* & + - ! ~`

new-expression:
  `::`$_{opt}$ `new` *new-placement*$_{opt}$ *new-type-id new-initializer*$_{opt}$
  `::`$_{opt}$ `new` *new-placement*$_{opt}$ `(` *type-id* `)` *new-initializer*$_{opt}$

new-placement:
  `(` *expression-list* `)`

new-type-id:
  type-specifier-seq new-declarator$_{opt}$

new-declarator:
  `*` *cv-qualifier-seq*$_{opt}$ *new-declarator*$_{opt}$
  `::`$_{opt}$ *nested-name-specifier* `*` *cv-qualifier-seq*$_{opt}$ *new-declarator*$_{opt}$
  direct-new-declarator

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

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

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

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

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

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

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

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

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

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

and-expression:
  equality-expression
  and-expression `&` *equality-expression*

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

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

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

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

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

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

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

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

constant-expression:
    conditional-expression

A.5 Statements [gram.stmt.stmt]

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

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

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

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

statement-seq:
statement
statement-seq statement

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

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

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

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

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

declaration-statement:
declaration

A.6 Declarations [gram.dcl.dcl]

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

decl-specifier-seq$_{opt}$ *init-declarator-list*$_{opt}$ `;`

decl-specifier:
storage-class-specifier
type-specifier
function-specifier
`friend`
`typedef`

decl-specifier-seq:
decl-specifier-seq$_{opt}$ *decl-specifier*

storage-class-specifier:
    `auto`
    `register`
    `static`
    `extern`
    `mutable`

function-specifier:
    `inline`
    `virtual`

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$_{opt}$

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 ;*

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

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]

postfix-expression:
    primary-expression
    postfix-expression . *id-expression*
    postfix-expression -> *id-expression*

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* ;

instantiation:
    `template` *specialization*

specialization:
    template-name < *template-argument-list* > *declaration*

template-parameter:
    type-parameter
    parameter-declaration

type-parameter:
    `class` *identifier*$_{opt}$
    `class` *identifier*$_{opt}$ = *type-name*
    `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*$_{\texttt{opt}}$

exception-specification:
    `throw` `(` *type-id-list*$_{opt}$ `)`

type-id-list:
    type-id
    type-id-list `,` *type-id*

Annex B (informative)
Implementation quantities [limits]

1 Because computers are finite, C++ implementations are inevitably limited in the size of the programs they can successfully process. Every implementation shall

Box 141

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].
- Significant initial characters in an internal identifier or macro name [1 024].

Box 142

Is there any reason that a C++ implementation should ever be permitted quietly to treat two different identifiers as identical, merely because they're long? I don't think so.

Editorial proposal. Change ''Significant initial characters in'' to ''Length of'' in thie two quantities surrounding this box.

- Significant 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 143
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 144
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 4.3. 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*`; 4.6. This is also in ISO C.

13 A pointer to function can be assigned to a `void*`; 4.6.

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 4.6

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 4.6

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;'
```