1 General [intro]

1.1 Scope [intro.scope]

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

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

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

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

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

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

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

1.2 Normative references [intro.refs]

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

— ANSI X3.172:1990, *American National Dictionary for Information Processing Systems*. |

— ISO/IEC 9899:1990, *C Standard*

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

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

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

1.3 Definitions [intro.defs]

1 For the purposes of this International Standard, the definitions given in 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 can change during the execution of a program. If a pointer (8.3.1) whose static type is "pointer to class `B`" is pointing to an object of class `D`, derived from B (10), the dynamic type of the pointer is "pointer to `D`." References (8.3.2) are treated similarly.

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

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

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

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

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

— **signature:** The signature of a function is the information about that function that participates in overload resolution (13.2): 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 (3.8) 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.

---

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

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

1.4 Syntax notation [syntax]

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

{ $expression_{opt}$ }

indicates an optional expression enclosed in braces.

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

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

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

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

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

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

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

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

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

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

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

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

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

4 Certain types have *alignment* restrictions. An object of one of those types shall appear only at an address that is divisible by a particular integer.

1.7 Processor compliance [intro.compliance]

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

2 Well-formed C++ programs are those that are constructed according to the syntax rules, diagnosable semantic rules, 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.

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

1.8 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 can thus have more than one possible execution sequence for a given program and a given input.

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

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

6 The observable behavior of the abstract machine is its sequence of reads and writes to `volatile` data and calls to library I/O functions.[3)]

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

8 It is important to note that certain contexts in C++ cause the evaluation of a full-expression that results from a syntactic construct other than *expression*(5.18). For example, in 8.5 one syntax for *initializer* is

(*expression-list*)

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

= *initializer-clause*

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

3) An implementation can offer additional library I/O functions as an extension. Implementations that do so should treat calls to those functions as ''observable behavior'' as well.

9 Also note that the evaluation of a full-expression can include the evaluation of subexpressions that are not lexically part of the full-expression. For example, subexpressions involved in evaluating default argument expressions (8.3.6) are considered to be created in the expression that calls the function, not the expression that defines the default argument.

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

11 When calling a function (whether or not the function is inline), there is a sequence point after the evaluation of all function arguments (if any) which takes place before execution of any expressions or statements in the function body. There is also a sequence point after the copying of a returned value and before the execution of any expressions outside the function[5]. Several contexts in C++ cause evaluation of a function call, even though no corresponding function-call syntax appears in the translation unit. For example, evaluation of a `new` expression invokes one or more allocation and constructor functions; see 5.3.4. For another example, invocation of a conversion function (12.3.2) can arise in contexts in which no function-call syntax appears. The sequence points at function-entry and function-exit (as described above) are features of the function-calls as evaluated, whatever the syntax of the translation unit might be.

12 In the evaluation of each of the expressions

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

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

Box 1

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

[4] As specified in 12.2, after the "end-of-full-expression" sequence point, a sequence of zero or more invocations of destructor functions takes place, in reverse order of the construction of each temporary object.

[5] The sequence point at the function return is not explicitly specified in ISO C, and can be considered redundant with sequence points at full-expressions, but the extra clarity is important in C++. In C++, there are more ways in which a called function can terminate its execution, such as the throw of an exception, as discussed below.

[6] The operators indicated in this paragraph are the builtin operators, as described in Clause 5. When one of these operators is overloaded (13) in a valid context, thus designating a user-defined operator function, the expression designates a function invocation, and the operands form an argument list, without an implied sequence point between them.

2 Lexical conventions [lex]

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

2.1 Phases of translation [lex.phases]

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

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

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

3 The source file is decomposed into preprocessing tokens (2.3) and sequences of white-space characters (including comments). A source file shall not end in a partial preprocessing token or partial comment[8]. 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 2.5). The resulting tokens are syntactically and semantically analyzed and translated. The result of this process starting from a single source file is called a *translation unit*.

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

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

8 The translation units that will form a program are combined. All external object and function references are resolved.

> **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>??=</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></tr>
<tr><td>??'</td><td>^</td><td>??!</td><td>|</td><td>??-</td><td>~</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]

> **Box 3**
>
> We have deleted the non-terminal for ’digraph’, because the alternate representations are just alternative ways of expressing a "first-class" preprocessing token. In C, # and ## are grouped with operators, but that would involve more work in clause 13, and wouldn’t fit the "spirit of C++". Instead, we simply list under which they are actual tokens.

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

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, 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*, *preprocessing-op-or-punc*, and single non-white-space characters that do not lexically match the other preprocessing token categories. If a `'` or a `"` character matches the last category, the

behavior is undefined. Preprocessing tokens can be separated by *white space*; this consists of comments (2.6), or *white-space characters* (space, horizontal tab, new-line, vertical tab, and form-feed), or both. As described in Clause 16, in certain circumstances during translation phase 4, white space (or the absence thereof) serves as more than preprocessing token separation. White space can appear within a preprocessing token only as part of a header name or between the quotation characters in a character constant or string literal.

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

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

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

2.4 Alternate tokens [lex.digraph]

1 Alternate token representations are provided for some operators and punctuators[9].

2 In all respects of the language, each alternate token behaves the same, respectively, as its primary token, except for its spelling[10]. The set of alternate tokens is defined in Table 2.

Table 2—alternate tokens

<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>%:%:</code></td><td><code>##</code></td><td><code>bitand</code></td><td><code>&</code></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.

[9] These include “digraphs” and additional reserved words. The term “digraph” (token consisting of two characters) is not perfectly descriptive, since one of the alternate preprocessing-tokens is `%:%:` and of course several primary tokens contain two characters. Nonetheless, those alternate tokens that aren’t lexical keywords are colloquially known as “digraphs”.

[10] Thus `[` and `<:` behave differently when “stringized” (16.3.2_), but can otherwise be freely interchanged.

2.6 Comments [lex.comment]

1 The characters `/*` start a comment, which terminates with the characters `*/`. These comments do not nest. The characters `//` start a comment, which terminates with the next new-line character. If there is a form-feed or a vertical-tab character in such a comment, only white-space characters can 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 is a letter; the underscore _ counts as a letter. Upper- and lower-case letters are different. All characters are significant.

2.8 Keywords [lex.key]

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

Table 3—keywords

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

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

Table 4—alternate representations

bitand	and	bitor	or	xor	compl
and_eq	or_eq	xor_eq	not	not_eq	

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

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

preprocessing-op-or-punc: one of

```
{       }       [       ]       #       ##      =       (       )       ,
<:      :>      <%      %>      %:      %:%:    ;       :       ...
new     delete  new[]   delete[]        ?
+       -       *       /       %       ^       &       |       ~
!       =       <       >       +=      -=      *=      /=      %=
^=      &=      |=      <<      >>      >>=     <<=     ==      !=
<=      >=      &&      ||      ++      --      ,       ->*     ->
and     bitand  bitor   compl   new<%%> delete<%%>
not     or      xor     and_eq  not_eq  or_eq   xor_eq
```

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

5 Certain implementation-dependent properties, such as the type of a `sizeof` (5.3.3) expression, the ranges of fundamental types (3.8.1), and the types of the most basic library functions are defined in the standard header files (18)

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

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

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

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

2.9 Literals [lex.literal]

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

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

2.9.1 Integer literals [lex.icon]

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

decimal-literal:
  nonzero-digit
  decimal-literal digit

octal-literal:
  `0`
  octal-literal octal-digit

hexadecimal-literal:
  `0x` *hexadecimal-digit*
  `0X` *hexadecimal-digit*
  hexadecimal-literal hexadecimal-digit

nonzero-digit: one of
  `1 2 3 4 5 6 7 8 9`

octal-digit: one of
  `0 1 2 3 4 5 6 7`

hexadecimal-digit: one of
  `0 1 2 3 4 5 6 7 8 9`
  `a b c d e f`
  `A B C D E F`

integer-suffix:
  unsigned-suffix long-suffix$_{opt}$
  long-suffix unsigned-suffix$_{opt}$

unsigned-suffix: one of
  `u U`

long-suffix: one of
  `l L`

1 An integer literal consisting of a sequence of digits is taken to be decimal (base ten) unless it begins with `0` (digit zero). A sequence of digits starting with `0` is taken to be an octal integer (base eight). The digits `8` and `9` are not octal digits. A sequence of digits preceded by `0x` or `0X` is taken to be a hexadecimal integer (base sixteen). The hexadecimal digits include `a` or `A` through `f` or `F` with decimal values ten through fifteen. For example, the number twelve can be written `12`, `014`, or `0XC`.

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

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

2.9.2 Character literals [lex.ccon]

character-literal:
  `'` *c-char-sequence* `'`
  `L'` *c-char-sequence* `'`

c-char-sequence:
  c-char
  c-char-sequence c-char

c-char:
  any member of the source character set except
    the single-quote `'`, backslash `\`, or new-line character
  escape-sequence

escape-sequence:
  simple-escape-sequence
  octal-escape-sequence
  hexadecimal-escape-sequence

simple-escape-sequence: one of
```
\'  \"  \?  \\
\a  \b  \f  \n  \r  \t  \v
```

octal-escape-sequence:
  `\` *octal-digit*
  octal-escape-sequence octal-digit

hexadecimal-escape-sequence:
  `\x` *hexadecimal-digit*
  hexadecimal-escape-sequence hexadecimal-digit

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

2 A character literal that begins with the letter `L`, such as `L'ab'`, is a wide-character literal. Wide-character literals have type `wchar_t`. They are intended for character sets where a character does not fit into a single byte. Wide-character literals have implementation-defined values, regardless of the number of characters in the literal.

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

Table 5—escape sequences

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

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

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

2.9.3 Floating literals [lex.fcon]

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

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

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

sign: one of
    `+` `-`

digit-sequence:
    digit
    digit-sequence digit

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

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

2.9.4 String literals [lex.string]

string-literal:
    `"`*s-char-sequence*$_{opt}$`"`
    `L"`*s-char-sequence*$_{opt}$`"`

s-char-sequence:
    s-char
    s-char-sequence s-char

s-char:
    any member of the source character set except
        the double-quote `"`, backslash `\`, or new-line character
    escape-sequence

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

undefined.

2 A string literal that begins with `L`, such as `L"asdf"`, is a wide-character string. A wide-character string is of type "array of *n* `wchar_t`," where *n* is the size of the string as defined below. Concatenation of ordinary and wide-character string literals is undefined.

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

3 Adjacent string literals are concatenated. Characters in concatenated strings are kept distinct. For example,

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

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

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

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

2.9.5 Boolean literals [lex.bool]

boolean-literal:
```
false
true
```

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

3 Basic concepts [basic]

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

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

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

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

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

6 For example, in

```
int j = 24;

main()
{
        int i = j, j;

        j = 42;
}
```

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

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

8 Two names denote the same entity if

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

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

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

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

3.1 Declarations and definitions [basic.def]

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

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

3 The following, for example, are definitions:

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

whereas these are just declarations:

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

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

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

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

the implementation will generate functions to make the definition of `C` equivalent to

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

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

3.2 One definition rule [basic.def.odr]

Box 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 can appear explicitly in the program, it can be found in the standard or a user-defined library, or (when appropriate) the implementation can generate it. If a non-virtual function is not defined, a diagnostic is required only if an attempt is actually made to call that function. If a virtual function is neither called nor defined, no diagnostic is required.

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

Box 10
This will need to be revisited when the ODR is made more precise

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

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 name look up rules are summarized in 3.4.

3.3.1 Local scope [basic.scope.local]

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

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

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

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

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

3.3.2 Function prototype scope [basic.scope.proto]

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

3.3.3 Function scope

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

3.3.4 Namespace scope [basic.scope.namespace]

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

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

3.3.5 Class scope [basic.scope.class]

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

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

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

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

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

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

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

3 The scope rules for classes are summarized in 9.3.

3.3.6 Name hiding [basic.scope.hiding]

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

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

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

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

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

Box 12

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

3.3.7 Explicit qualification [basic.scope.exqual]

Box 13

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

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

3.3.8 Elaborated type specifier [basic.scope.elab]

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

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

main()
{
    int A;

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

The scope of class names first introduced in *elaborated-type-specifiers* is described in (7.1.5.3).

3.3.9 Point of declaration [basic.scope.pdecl]

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

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

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

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

4 For the point of declaration of a function first declared in a `friend` declaration, see 11.4.

5 For the point of declaration of a class first declared in an *elaborated-type-specifier* or in a `friend` declaration, see 7.1.5.3.

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 Name look up [class.scope]

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

2 Name look up associates the use of a name with a visible declaration (3.1) of that name. Name look up shall find an unambiguous declaration for the name (see 10.2). Name look up may associate more than one declaration with a name if it finds the name to be a function name; in this case, all the declarations shall be found in the same scope (10.2); the declarations are said to form a set of overloaded functions (13.1).

Overload resolution (13.2) takes place after name look up has succeeded. The access rules (11) are considered only once name look up and function overload resolution (if applicable) have succeeded. Only after name look up, function overload resolution (if applicable) and access checking have succeeded are the attributes introduced by the name's declaration used further in expression processing (5).

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

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

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

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

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

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

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

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

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

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

Box 14
This subclause should probably say something about look up in template definitions.

3.5 Program and linkage [basic.link]

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

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

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

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

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

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

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

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

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

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

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

— a variable, unless it has internal linkage; or

— a function, unless it has internal linkage; or

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

— a template (14). In addition, a name of class scope has external linkage if the name of the class has external linkage. *

Box 15
What is the linkage of unnamed classes and their members? Unnamed enumeration and their enumerators?

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

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

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

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

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

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

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

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

8 After all adjustments of types (during which typedefs (7.1.3) are replaced by their definitions), the types specified by all declarations of a particular external name shall be identical, except that such types can 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 16
This needs to specified more precisely to deal with function name overloading.

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

3.6 Start and termination [basic.start]

3.6.1 Main function [basic.start.main]

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

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

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

or

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

In the latter form `argc` shall be the number of arguments passed to the program from an environment in which the program is run. If `argc` is nonzero these arguments shall be supplied in `argv[0]` through `argv[argc-1]` as pointers to the initial characters of zero-terminated strings; and `argv[0]` shall be the pointer to the initial character of a zero-terminated string that represents the name used to invoke the program or `""`. It is guaranteed that `argv[argc]==0`.

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

4 Calling the function

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

declared in `<cstdlib>` (18.3) terminates the program without leaving the current block and hence without destroying any 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;
```

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

Box 17
This is still under active discussion by the committee.

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

2 If construction or destruction of a non-local static object ends in throwing an uncaught exception, the result is to call `terminate()` (18.6.1.3).

3.6.3 Termination [basic.start.term]

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

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

3 Calling the function

```
void abort();
```

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

3.7 Storage duration and lifetime [basic.stc]

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

— static storage duration

— automatic storage duration

— dynamic storage duration

2 Static and automatic storage durations are associated with objects introduced by declarations (3.1) and with temporaries (12.2). The dynamic storage duration is associated with objects created with `operator new` (5.3.4).

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

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

5 The *lifetime* of an object is a runtime property of the object. The implementation controls the lifetime of objects with static or automatic storage duration. Users control the lifetime of objects with dynamic storage duration.

Box 18

What is the lifetime of an object? When is it well-formed and well-defined to access an object? When is it ill-formed or undefined to access an object? Subclause 1.5 used to say: "The lifetime of an object starts after any required initialization (8.5) has completed. For objects with destructor, it ends when destruction starts." This description is being worked out by the Core Language WG. In particular, a better description is needed to take into account what happens when users play tricks with objects' lifetime.

6 The lifetime of temporaries is described in (12.2).

3.7.1 Static storage duration [basic.stc.static]

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

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

Box 19

This awaits committee action on the ‘‘as-if’’ rule.

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

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

5 Temporaries created at global scope have static storage duration.

3.7.2 Automatic storage duration [basic.stc.auto]

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

2 These objects are initialized and destroyed as described 6.7.

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

4 Temporaries created in block scope have automatic storage duration.

3.7.3 Dynamic storage duration [basic.stc.dynamic]

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

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

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

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

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

2 The function shall return the address of a block of available storage at least as large as the requested size. The order, contiguity, and initial value of storage allocated by successive calls to an allocation function is unspecified. The pointer returned is suitably aligned so that it can be assigned to a pointer of any type and then used to access such an object or an array of such objects in the storage allocated (until the storage is explicitly deallocated by a call to a corresponding deallocation function). Each such allocation shall yield a pointer to storage (1.5) disjoint from any other currently allocated storage. The pointer returned points to the start (lowest byte address) of the allocated storage. If the size of the space requested is zero, the value returned shall be nonzero and shall not pointer to or within any other currently allocated storage. The results of dereferencing a pointer returned as a request for zero size are undefined.[11)]

3 If an allocation function is unable to obtain an appropriate block of storage, it can invoke the currently installed `new_handler`[12)] and/or throw an exception (15) of class `bad_alloc` (18.4.2.1) or a class derived from `bad_alloc`.

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

[11)] 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.

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

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

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

2 Each deallocation function shall return `void` and its first parameter shall be `void*`. For class member deallocation functions, a second parameter of type `size_t` can be added but deallocation functions shall not be overloaded.

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

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

5 If the argument is non-null, the value of a pointer that refers to deallocated space is *indeterminate*. The effect of dereferencing an indeterminate pointer value is undefined.[13)]

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

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

3.7.5 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 3.9, 5.2.4, 7.1.1 and 7.1.5.1. *

3.8 Types [basic.types]

1 There are two kinds of types: fundamental types and compound types. Types can describe objects (1.6), references (8.3.2), or functions (8.3.5). *

2 Object types have *alignment requirements* (3.8.1, 3.8.2). The alignment of an object type is an implementation-dependent integer value representing a number of bytes; an object is allocated at an address that is divisible by the alignment of its object type.

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

4 A class type (such as "`class X`") can 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 can be incomplete at one point in a translation unit and complete later on; the array types at those two points ("array of unknown bound of `T`" and "array of N `T`") are different types. However, the type of a pointer to array of unknown size, or of a type defined by a `typedef` declaration to be an array of unknown size, cannot be completed.

5 Expressions that have incomplete type are prohibited in some contexts. For example:

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

```
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
}
```

6 Clauses 5 and 6 indicate in more details in which contexts incomplete types are allowed or prohibited.

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

3.8.1 Fundamental types [basic.fundamental]

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

2 Objects declared as characters (`char`) are large enough to store any member of the implementation's basic character set. If a character from this set is stored in a character variable, its value is equivalent to the integer code of that character. It is implementation-specified whether a `char` object can take on negative values. Characters can be explicitly declared `unsigned` or `signed`. Plain `char`, `signed char`, and `unsigned char` are three distinct types. A `char`, a `signed char`, and an `unsigned char` occupy the same amount of storage and have the same alignment requirements (3.8). In any particular implementation, a plain `char` object can take on either the same values as a `signed char` or an `unsigned char`; which one is implementation-defined.

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

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

5 For each of the signed integer types, there exists a corresponding (but different) *unsigned integer type*: "`unsigned char`", "`unsigned short int`", "`unsigned int`", and "`unsigned long int`," each of which occupies the same amount of storage and has the same alignment requirements (3.8) as the corresponding signed integer type.[14] The range of nonnegative values of a *signed integer* type is a

14) See 7.1.5.2 regarding the correspondence between types and the sequences of *type-specifier*s that designate them.

subrange of the corresponding *unsigned integer* type, and the representation of the same value in each type is the same.

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

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

8 Values of type `bool` can be either `true` or `false`.[15] There are no `signed`, `unsigned`, `short`, or `long bool` types or values. As described below, `bool` values behave as integral types. Thus, for example, they participate in integral promotions (4.5, 5.2.3). Although values of type `bool` generally behave as signed integers, for example by promoting (4.5) to `int` instead of `unsigned int`, a `bool` value can successfully be stored in a bit-field of any (nonzero) size.

9 Types `bool`, `char`, `wchar_t`, and the signed and unsigned integer types are collectively called *integral* types. A synonym for integral type is *integer type*. Enumerations (7.2) are not integral, but they can be promoted (4.5) to `int`, `unsigned int`, `long`, or `unsigned long`. The representations of integral types shall define values by use of a pure binary numeration system.

Box 20

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

10 There are three *floating point* types: `float`, `double`, and `long double`. The type `double` provides at least as much precision as `float`, and the type `long double` provides at least as much precision as `double`. *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. Objects of type `void` shall not be declared. Any expression can be explicitly converted to type `void` (5.4); the resulting expression can be used only as an expression statement (6.2), as the left operand of a comma expression (5.18), or as a second or third operand of `?:` (5.16).

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

3.8.2 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;

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

— *pointers to non-static*[16] *class members*, which identify members of a given type within objects of a

[15] 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`.

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

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 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 (3.8). Pointers to qualified or unqualified versions (3.8.3) of layout-compatible types shall have the same representation and alignment requirements (3.8). *

4 Objects of cv-qualified (3.8.3) or unqualified type `void*` (pointer to void), can be used to point to objects of unknown type. A `void*` has enough bits to hold any object pointer. A qualified or unqualified (3.8.3) `void*` shall have the same representation and alignment requirements as a qualified or unqualified `char*`.

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

3.8.3 CV-qualifiers [basic.type.qualifier]

1 Any type so far mentioned is an *unqualified type*. Each unqualified fundamental type (3.8.1) has three corresponding qualified versions of its type: a *const-qualified* version, a *volatile-qualified* version, and a *const-volatile-qualified* version. The term *object type* (1.6) includes the cv-qualifiers specified when the object is created. The presence of a `const` specifier in a *decl-specifier-seq* declares an object of *const-qualified object type*; such object is called a *const object*. The presence of a `volatile` specifier in a *decl-specifier-seq* declares an object of *volatile-qualified object type*; such object is called a *volatile object*. The presence of both *cv-qualifiers* in a *decl-specifier-seq* declares an object of *const-volatile-qualified object type*; such object is called a *const volatile object*. The cv-qualified or unqualified versions of a type are distinct types; however, they have the same representation and alignment requirements (3.8).[17] A compound type (3.8.2) is not cv-qualified by the cv-qualifiers (if any) of the type from which it is compounded. However, an array type is considered to be cv-qualified by the cv-qualifiers of its element type. Moreover, when an array type is cv-qualified, its element type is considered to have the same cv-qualifiers (8.3.4). *

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

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

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

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

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

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

3.8.4 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).

3.9 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.[18)]

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

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

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

9 An lvalue for an object is generally necessary in order to modify the object. An rvalue of class type can also be used to modify its referent under certain circumstances. For example, a member function called for an object (9.4) can modify the object.

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

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

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

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

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.

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

4 Standard conversions [conv]

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

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

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

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

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

2 Standard conversions are implicit conversions defined for built-in types. For user-defined types, user-defined conversions are considered as well; see 12.3. In general, an implicit conversion sequence (13.2.3.1) consists of zero or more standard conversions and zero or one user-defined conversion.

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

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

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

1 An lvalue (3.9) of a non-array type `T` can be converted to an rvalue. If `T` is an incomplete type, a program that necessitates this conversion is ill-formed. If `T` is a non-class type, the type of the rvalue is the unqualified version of `T`. Otherwise (i.e., `T` is a class type), the type of the rvalue is `T`. [19]

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

3 See also 3.9.

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

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

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

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

1 An lvalue of function type `T` can be converted to an rvalue of type “pointer to `T`.” The result is a pointer to the function.[20]

2 See 13.3 for additional rules for the case where the function is overloaded.

4.4 Qualification conversions [conv.qual]

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

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

3 A conversion can add type qualifiers at levels other than the first in multi-level pointers, subject to the following rules:[21]

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

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

and

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

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

— the pointer types are similar.

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

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

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

4.5 Integral promotions [conv.prom]

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

2 An rvalue of type `wchar_t` (3.8.1) or an enumeration type (7.2) can be converted to an rvalue of the first of the following types that can represent all the values of the source type: `int`, `unsigned int`, `long`, or `unsigned long`.

3 An rvalue for an integral bit-field (9.7) can be converted to an rvalue of type `int` if `int` can represent all the values of the bit-field; otherwise, it can be converted to `unsigned int` if `unsigned int` can represent all the values of the bit-field[22].

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

5 These conversions are called integral promotions.

---

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

21) These rules ensure that const-safety is preserved by the conversion.

22) If the bit-field is larger yet, it is not eligible for integral promotion. If the bit-field has an enumerated type, it is treated as any other value of that type for promotion purposes.

4.6 Floating point promotion [conv.fpprom]

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

2 This conversion is called floating point promotion.

4.7 Integral conversions [conv.integral]

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

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

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

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

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

4.8 Floating point conversions [conv.double]

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

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

4.9 Floating-integral conversions [conv.fpint]

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

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

4.10 Pointer conversions [conv.ptr]

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

2 An rvalue of type "pointer to *cv* `T`," where `T` is an object type, can be converted to an rvalue of type "pointer to *cv* `void`."

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

4.11 Pointer to member conversions [conv.mem]

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

2 An rvalue of type "pointer to member of `B` of type *cv* `T`," where `B` is a class type, can be converted to an rvalue of type "pointer to member of `D` of type *cv* `T`," where `D` is a derived class (10) of `B`. If `B` is an inaccessible (11) or ambiguous (10.2) base class of `D`, a program that necessitates this conversion is ill-formed. The result of the conversion refers to the same member as the pointer to member before the conversion took place, but it refers to the base class member as if it were a member of the derived class. The result refers to the member in `D`'s instance of `B`. Since the result has type "pointer to member of `D` of type *cv* `T`," it can be dereferenced with a `D` object. The result is the same as if the pointer to member of `B` were dereferenced with the `B` sub-object of `D`. The null member pointer value is converted to the null member pointer value of the destination type.[23)]

4.12 Base class conversion [conv.class]

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

4.13 Boolean conversions [conv.bool]

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

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

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

5 Expressions [expr]

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

2 Operators can be overloaded, that is, given meaning when applied to expressions of class type (9). Uses of overloaded operators are transformed into function calls as described in 13.4. 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).[24)]

3 This clause defines the operators when applied to types for which they have not been overloaded. Operator overloading shall not modify the rules for the *built-in operators*, that is, for operators applied to types for which they are defined by the language itself. However, these built-in operators participate in overload resolution; see 13.2.1.2.

4 Operators can be regrouped according to the usual mathematical rules only where the operators really are associative or commutative. Overloaded operators are never assumed to be associative or commutative. Except where noted, the order of evaluation of operands of individual operators and subexpressions of individual expressions is unspecified. In particular, if a value is modified twice in an expression, the result of the expression is unspecified except where an ordering is guaranteed by the operators involved. For example,

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

5 The handling of overflow and divide by zero in expression evaluation is implementation dependent. Most existing implementations of C++ ignore integer overflows. Treatment of division by zero and all floating point exceptions vary among machines, and is usually adjustable by a library function.

6 Except where noted, operands of types `const T`, `volatile T`, `T&`, `const T&`, and `volatile T&` can be used as if they were of the plain type `T`. Similarly, except where noted, operands of type `T* const` and `T* volatile` can be used as if they were of the plain type `T*`. Similarly, a plain `T` can be used where a `volatile T` or a `const T` is required. These rules apply in combination so that, except where noted, a `T* const volatile` can be used where a `T*` is required. Such uses do not count as standard conversions when considering overloading resolution (13.2).

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

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

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

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

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

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

Box 21

Enumerations are handled correctly by the usual arithmetic conversions, and for any operator that invokes the integral promotions. However, there may be other places in this Clause that fail to treat enumerations appropriately. *

12

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

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

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

— Otherwise, the integral promotions (4.5) are performed on both operands.[25)]

— 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`.

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

— the 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),

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

— a character type.[26)] the result is undefined.

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

5.1 Primary expressions [expr.prim]

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

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

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

3 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` shall not be used outside a class member function body.

Box 22
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* shall be of namespace 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 23
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 data member type or function member type; it is not an object type. The result is the member. The result is an lvalue if the member is. The *class-name* might 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 shall 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 shall have the type “pointer to `T`” and the other shall be of enumeration or integral type. The result is an lvalue of type “`T`.” The type “`T`” shall be complete. The expression `E1[E2]` is identical (by definition) to `*((E1)+(E2))`. 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[27] (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 shall be a function name, or a pointer or reference to function. For member function call, the postfix expression shall 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. In the case of an implicit class member access, the implied object is the one pointed to by `this`. That is, a member function call of the form `f()` is interpreted as `this->f()` (see 9.4.2). If a function or member function name is used, the name can 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

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

value of the object expression). 12.7 describes the behavior of virtual function calls when the object-expression refers to an object under construction or destruction.

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

3 When a function is called, each parameter (8.3.5) 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 can change the values of its nonconstant parameters, but these changes cannot affect the values of the arguments except where a parameter is of a non-`const` reference type (8.3.2). Where a parameter is of reference type a temporary 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 can be declared to accept fewer arguments (by declaring default arguments (8.3.6)) or more arguments (by using the ellipsis, `...` 8.3.5) than the number of parameters in the function definition (8.4).

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

6 Any argument of type `float` for which there is no parameter is converted to `double` before the call; any of `char`, `short`, or a bit-field type for which there is no parameter are converted to `int` or `unsigned` by integral promotion (4.5). Any argument of enumeration type is converted to `int`, `unsigned`, `long`, or `unsigned long` by integral promotion. An object of a class for which no parameter is declared is passed as a data structure.

Box 24

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

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

8 The order of evaluation of arguments is unspecified; take note that compilers differ. All side effects of argument expressions take effect before the function is entered. The order of evaluation of the postfix expression and the argument expression list is unspecified.

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

10 Recursive calls are permitted.

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

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

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

2 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;[28] the result of that evaluation, together with the *id-expression*, determine the result of the entire postfix expression.

2 For the first option (dot) the type of the first expression (the *object expression*) shall be "class object" (of a complete type). For the second option (arrow) the type of the first expression (the *pointer expression*) shall be "pointer to class object" (of a complete type). The *id-expression* shall name a member of that class, except that an imputed destructor can be explicitly invoked for a built-in type (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)[29].

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

4 Similarly, if the *id-expression* is a *conversion-function-id*, its *conversion-type-id* shall denote the same type in both the context in which the entire *postfix-expression* occurs and in the context of the class of the object expression (or the class pointed to by the pointer expression). For the purpose of this evaluation, the name, if any, of each class is also considered a nested class member of that class.

5 Abbreviating *object-expression.id-expression* as `E1.E2`, then the type and lvalue properties of this expression are determined as follows. In the remainder of this subclause, *cq* represents either `const` or the absence of `const`; *vq* represents either `volatile` or the absence of `volatile`. *cv* represents an arbitrary set of cv-qualifiers, as defined in 3.8.3.

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

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

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

[28] 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.
[29] Note that if `E1` has the type "pointer to class `X`", then `(*(E1))` is an lvalue.

function call (9.4). The member function shall be at least as cv-qualified as the left-hand operand. The type of `E1.E2` is "class `X`'s *cv* member function of (parameter type list) returning `T`".

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

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

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

5.2.5 Increment and decrement [expr.post.incr]

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

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

5.2.6 Dynamic cast [expr.dynamic.cast]

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

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

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

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

5 If `T` is "pointer to *cv1* `B`" and `v` has type "pointer to *cv2* `D`" such that `B` is a base class of `D`, the result is a pointer to the unique `B` sub-object of the `D` object pointed to by `v`. Similarly, if `T` is "reference to *cv1* `B`" and `v` has type "*cv2* `D`" such that `B` is a base class of `D`, the result is an lvalue for the unique[30] `B` sub-object of the `D` object referred to by `v`. In both the pointer and reference cases, *cv1* shall be the same cv-qualification as, or greater cv-qualification than, *cv2*, and `B` shall be an accessible nonambiguous base class of `D`. For example,

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

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

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

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

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

Box 25

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

9 The value of a failed cast to pointer type is the null pointer value of the required result type. A failed cast to reference type throws `bad_cast` (18.5.2.1). For example,

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

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

12.7 describes the behavior of a `dynamic_cast` applied to an object under construction or destruction.

5.2.7 Type identification [expr.typeid]

1 The result of a `typeid` expression is of type `const type_info&`. The value is a reference to a `type_info` object (18.5.1.1) that represents the *type-id* or the type of the *expression* respectively.

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

3 If the *expression* is the result of applying unary `*` to a pointer to a polymorphic type,[31] then the pointer shall either be zero or point to a valid object. If the pointer is zero, the `typeid` expression throws the `bad_typeid` exception (18.5.2.2). Otherwise, the result of the `typeid` expression is the value that represents the type of the complete object to which the pointer points.

[31] If `p` is a pointer, then `*p`, `(*p)`, `((*p))`, and so on all meet this requirement.

4 If the *expression* is the result of subscripting (5.2.1) a pointer, say `p`, that points to a polymorphic type,[32)] then the result of the `typeid` expression is that of `typeid(*p)`. The subscript is not evaluated.

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

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

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

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

12.7 describes the behavior of `typeid` applied to an object under construction or destrcution.

5.2.8 Static cast [expr.static.cast]

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

2 Any implicit conversion (including standard conversions and/or user-defined conversions; see 4 and 13.2.3.1) can be performed explicitly using `static_cast`. More precisely, if `T t(v);` is a well-formed declaration, for some invented temporary variable `t`, then the result of `static_cast<T>(v)` is defined to be the temporary `t`, and is an lvalue if `T` is a reference type, and an rvalue otherwise. The expression `v` shall be an lvalue if the equivalent declaration requires an lvalue for `v`.

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

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

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

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

6 The inverse of any standard conversion (4) can be performed explicitly using `static_cast` subject to the restriction that the explicit conversion does not cast away constness (5.2.10), and the following additional rules for specific cases:

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

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

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

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

5.2.9 Reinterpret cast [expr.reinterpret.cast]

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

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

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

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

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

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

7 A pointer to a function can be explicitly converted to a pointer to a function of a different type. The effect of calling a function through a pointer to a function type that differs from the type used in the definition of the function is undefined. See also 4.10.

8 A pointer to an object can be explicitly converted to a pointer to an object of different type. In general, the results of this are unspecified; except that converting an rvalue of type "pointer to `T1`" to the type "pointer to `T2`" (where `T1` and `T2` are object types and where the alignment requirements of `T2` are no stricter than those of `T1`) and back to its original type yields the original pointer value.

Box 26
This does not allow conversion of function pointers to other function pointer types and back. Should it?

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

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

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

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

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

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

5.2.10 Const cast [expr.const.cast]

1

Box 27
Editorial change from previous edition: it is permitted to use `const_cast` as a no-op.

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

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

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

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

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

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

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

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

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

to

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

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

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

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

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

Box 28
This will need to be reworked once the memory model and object model are ironed out.

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

5.3 Unary expressions [expr.unary]

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

unary-expression:
  postfix-expression
  `++` *unary-expression*
  `--` *unary-expression*
  unary-operator cast-expression
  `sizeof` *unary-expression*
  `sizeof (` *type-id* `)`
  new-expression
  delete-expression

unary-operator: one of
  `* & + - ! ~`

5.3.1 Unary operators [expr.unary.op]

1 The unary `*` operator means *indirection*: the expression shall 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 shall be an lvalue or a *qualified-id*. In the first case, if the type of the expression is "`T`," the type of the result is "pointer to `T`." In particular, the address of an object of type "*cv* `T`" is "pointer to *cv* `T`," with the same cv-qualifiers. 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 a nonstatic member of class `C` of type `T`, the type of the result is "pointer to member of `class C` of type `T`." For example:

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

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

Box 29
This section probably needs to take into account `const` and its relationship to `mutable`.

3 The address of an object of incomplete type can be taken, but only if the complete type of that object does not have the address-of operator (`operator&()`) overloaded; no diagnostic is required.

4 The address of an overloaded function (13) can be taken only in a context that uniquely determines which version of the overloaded function is referred to (see 13.3). Note that since the context might determine whether the operand is a static or nonstatic member function, the context can also affect whether the expression has type "pointer to function" or "pointer to member function."

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

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

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

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

5.3.2 Increment and decrement [expr.pre.incr]

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

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

5.3.3 Sizeof [expr.sizeof]

1 The `sizeof` operator yields the size, in bytes, of its operand. The operand is either an expression, which is not evaluated, or a parenthesized *type-id*. The `sizeof` operator shall not be applied to an expression that has function or incomplete type, or to the parenthesized name of such a type, or to an lvalue that designates a bit-field. A *byte* is unspecified by the language except in terms of the value of `sizeof`; `sizeof(char)` is `1`, but `sizeof(bool)` and `sizeof(wchar_t)` are implementation-defined.[33]

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.

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

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

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

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

6 The result is a constant of type `size_t`, an implementation-dependent unsigned integral type defined in the standard header `<cstddef>`(18.1).

5.3.4 New [expr.new]

1 The *new-expression* attempts to create an object of the *type-id* (8.1) to which it is applied. This type shall be a complete object or array type (1.5, 3.8).

new-expression:
  `::`$_{opt}$ `new` *new-placement*$_{opt}$ *new-type-id* *new-initializer*$_{opt}$
  `::`$_{opt}$ `new` *new-placement*$_{opt}$ `(` *type-id* `)` *new-initializer*$_{opt}$

new-placement:
  `(` *expression-list* `)`

new-type-id:
  type-specifier-seq new-declarator$_{opt}$

new-declarator:
  `*` *cv-qualifier-seq*$_{opt}$ *new-declarator*$_{opt}$
  `::`$_{opt}$ *nested-name-specifier* `*` *cv-qualifier-seq*$_{opt}$ *new-declarator*$_{opt}$
  direct-new-declarator

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

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

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

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

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

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

3 Parentheses shall 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 (3.8.2). 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 class declarations, or enumeration declarations.

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

7 Every *constant-expression* in a *direct-new-declarator* shall be an integral constant expression (5.19) with a strictly positive value. The *expression* in a *direct-new-declarator* shall be of integral type (3.8.1) 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* (3.7.3.1). When the allocation function is called, the first argument will be amount of space requested (which might be larger than the size of the object being created only if that object is an array).

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

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[](sizeof(T)*5+x)`, and

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

13 The 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 A *new-expression* for a class calls one of the class constructors (12.1) to initialize i the object. An object of a class can be created by `new` only if suitable arguments are provided for the class' constructors by the *new-initializer*, or if the class has a default constructor.[34] If no user-declared constructor is used and a *new-initializer* is provided, the new-initializer shall be of the form *(expression)* or *();* if the expression is present, it shall be of class type and is used to initialize the object.

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.[35] In that case, the default constructor will be called for each element of the array, in order of increasing address.

[34] This means that `struct s{}; s* ps = new s;` is allowed on the grounds that `class s` has an implicitly-declared default constructor.
[35] PODS structs have an implicitly-declared default constructor.

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

17 The allocation function can indicate failure by throwing a `bad_alloc` exception (15, 18.4.2.1). 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 (3.7.3.2, 12.5) is used to free the memory in which the object was being constructed, after which the exception continues to propagate in the context of the *new-expression*.

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 30

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 31

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

In the second alternative (*delete array*), the value of the operand of `delete` shall be a pointer to an array created by a *new-expression* without a *new-placement* specification.

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

Box 32

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

4 The deletion of an object might 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 (3.7.3.2).

5 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. *

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

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

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

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

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

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

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

2 Types shall not be defined in casts.

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

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

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

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

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

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

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

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

4 If the result of `.*` or `->*` is a function, then that result can be used only as the operand for the function call operator `()`. For example,

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

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

5.6 Multiplicative operators [expr.mul]

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

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

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

3 The binary `*` operator indicates multiplication.

4 The binary `/` operator yields the quotient, and the binary `%` operator yields the remainder from the division of the first expression by the second. If the second operand of `/` or `%` is zero the 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 are performed for operands of arithmetic type.

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

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

2 For subtraction, one of the following shall hold:

— both operands have arithmetic type;

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

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

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

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

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

6 When two pointers to elements of the same array object are subtracted, the result is the difference of the subscripts of the two array elements. The type of the result is an implementation-defined signed integral type; this type shall be the same type that is defined as `ptrdiff_t` in the `<cstddef>` header (18.1). As with any other arithmetic overflow, if the result does not fit in the space provided, the behavior is undefined. In other words, if the expressions `P` and `Q` point to, respectively, the *i*-th and *j*-th elements of an array object, the expression `(P)-(Q)` has the value $i-j$ provided the value fits in an object of type `ptrdiff_t`. Moreover, if the expression `P` points either to an element of an array object or one past the last element of an array object, and the expression `Q` points to the last element of the same array object, the expression `((Q)+1)-(P)` has the same value as `((Q)-(P))+1` and as `-((P)-((Q)+1))`, and has the value zero if the expression `P` points one past the last element of the array object, even though the expression `(Q)+1` does not point to an element of the array object. Unless both pointers point to elements of the same array object, or one past the last element of the array object, the behavior is undefined.[36]

5.8 Shift operators [expr.shift]

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

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

The operands shall be of integral type and integral promotions are performed. The type of the result is that of the promoted left operand. The 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 shall have arithmetic or pointer type. The operators < (less than), > (greater than), <= (less than or equal to), and >= (greater than or equal to) all yield `false` or `true`. The type of the result is `bool`.

2 The usual arithmetic conversions are performed on arithmetic operands. Pointer conversions are performed on pointer operands to bring them to the same type, which shall be a qualified or unqualified version of the type of one of the operands. This implies that any pointer can be compared to an integral constant expression evaluating to zero and any pointer can be compared to a pointer of 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) can be compared; the result depends on the relative positions of the pointed-to objects or functions in the address space.

---

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

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

3 If two pointers of the same type point to the same object or function, or both point one past the end of the same array, or are both null, they compare equal. If two pointers of the same type point to different objects or functions, or only one of them is null, they compare unequal. If two pointers point to nonstatic data members of the same object, the pointer to the later declared member compares higher provided the two members not separated by an *access-specifier* label (11.1) and provided their class is not a union. If two pointers point to nonstatic members of the same object separated by an *access-specifier* label (11.1) the result is unspecified. If two pointers point to data members of the same union, they compare equal (after conversion to `void*`, if necessary). If two pointers point to elements of the same array or one beyond the end of the array, the pointer to the object with the higher subscript compares higher. Other pointer comparisons are implementation-defined.

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

5.11 Bitwise AND operator [expr.bit.and]

1

and-expression:
equality-expression
and-expression `&` *equality-expression*

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

5.12 Bitwise exclusive OR operator [expr.xor]

1

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

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

5.13 Bitwise inclusive OR operator [expr.or]

1

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

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

5.14 Logical AND operator [expr.log.and]

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

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

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

5.15 Logical OR operator [expr.log.or]

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

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

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

5.16 Conditional operator [expr.cond]

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

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

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

3 If both the second and the third expressions are of arithmetic type, then if they are of the same type the result is of that type; otherwise the usual arithmetic conversions are performed to bring them to a common type. Otherwise, if both the second and the third expressions are either a pointer or an integral constant expression that evaluates to zero, pointer conversions (4.10) are performed to bring them to a common type, which shall be a qualified or unqualified version of the type of either the second or the third expression. Otherwise, if both the second and the third expressions are either a pointer to member or an integral constant expression that evaluates to zero, pointer to member conversions (4.11) are performed to bring them to a common type[37)] which shall be a qualified or unqualified version of the type of either the second or the third expression. Otherwise, if both the second and the third expressions are lvalues of related class types, they are converted to a common type as if by a cast to a reference to the common type (5.2.8). Otherwise, if both the second and the third expressions are of the same class `T`, the common type is `T`. Otherwise, if both the second and the third expressions have type "*cv* `void`", the common type is "*cv* `void`." Otherwise the expression is ill formed. The result has the common type; only one of the second and third expressions is evaluated. The result is an lvalue if the second and the third operands are of the same type and both are lvalues.

37) 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.

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

4 Assignment to objects of a class (9) `X` is defined by the function `X::operator=()` (13.4.3). Unless the user defines an `X::operator=()`, the default version is used for assignment (12.8). This implies that an object of a class derived from `X` (directly or indirectly) by unambiguous public derivation (10) can be assigned to an `X`.

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

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

7 The behavior of an expression of the form `E1` *op*= `E2` is equivalent to `E1 = E1` *op* `E2` except that `E1` is evaluated only once. `E1` shall not have `bool` type. In += and -=, `E1` can be a pointer to a possibly-qualified completely defined object type, in which case `E2` shall have integral type and is converted as explained in 5.7; In all other cases, `E1` and `E2` shall have arithmetic type.

8 See 15.1 for throw expressions.

5.18 Comma operator [expr.comma]

1 The comma operator groups left-to-right.

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

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

2 In contexts where comma is given a special meaning, for example, in lists of arguments to functions (5.2.2) and lists of initializers (8.5), the comma operator as described in this clause can appear only in parentheses; for example,

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

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

5.19 Constant expressions [expr.const]

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

constant-expression:
  conditional-expression

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

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

— a null pointer constant (4.10),

— a null member pointer value (4.11),

— an arithmetic constant expression,

— an address constant,

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

— a pointer to member constant expression.

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

4 An *address constant* is a pointer to an lvalue designating an object of static storage duration or a function. The pointer shall be created explicitly, using the unary `&` operator, or implicitly using an expression of array (4.2) or function (4.3) type. The subscripting operator `[]` and the class member access `.` and `->` operators, the `&` and `*` unary operators, and pointer casts (except `dynamic_cast`s, 5.2.6) can be used in the creation of an address constant, but the value of an object shall not be accessed by the use of these operators. An expression that designates the address of a member or base class of a non-POD class object (9) is never an address constant expression (12.7). Function calls shall not be used in an address constant expression, even if the function is `inline` and has a reference return type.

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

6 Statements [stmt.stmt]

1 Except as indicated, statements are executed in sequence.

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

6.1 Labeled statement [stmt.label]

1 A statement can be labeled.

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

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

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

6.2 Expression statement [stmt.expr]

1 Most statements are expression statements, which have the form

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

Usually expression statements are assignments or function calls. All side effects from an expression statement are completed before the next statement is executed. An expression statement with the expression missing is called a null statement; it is useful to carry a label just before the `}` of a compound statement and to supply a null body to an iteration statement such as `while` (6.5.1).

6.3 Compound statement or block [stmt.block]

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

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

statement-seq:
  statement
  statement-seq statement

A compound statement defines a local scope (3.3).

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

6.4 Selection statements [stmt.select]

1 Selection statements choose one of several flows of control.

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

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

The *statement* in a *selection-statement* (both statements, in the `else` form of the `if` statement) implicitly defines a local scope (3.3). That is, if the statement in a selection-statement is a single statement and not a *compound-statement,* it is as if it was rewritten to be a compound-statement containing the original statement. For example,

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

can be equivalently rewritten as

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

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

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

3 A name introduced by a declaration in a *condition* is in scope from its point of declaration until the end of the 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 shall 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 shall 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 can be labeled with one or more case labels as follows:

`case` *constant-expression* `:`

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

3 There shall be at most one label of the form

`default :`

within a `switch` statement.

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

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

6 `case` and `default` labels in themselves do not alter the flow of control, which continues unimpeded across such labels. To exit from a switch, see `break`, 6.6.1.

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

6.5 Iteration statements [stmt.iter]

1 Iteration statements specify looping.

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

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

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

3 The *statement* in an *iteration-statement* implicitly defines a local scope (3.3) which is entered and exited each time through the loop. That is, if the statement in an iteration-statement is a single statement and not a *compound-statement,* it is as if it was rewritten to be a compound-statement containing the original statement. For example,

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

can be equivalently rewritten as

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

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

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

6.5.1 The `while` statement [stmt.while]

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

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

6.5.2 The `do` statement [stmt.do]

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

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

6.5.3 The `for` statement [stmt.for]

1 The `for` statement

```
for ( for-init-statement condition_opt ; expression_opt ) statement
```

is equivalent to

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

except that a `continue` in *statement* (not enclosed in another iteration statement) will execute *expression* before re-evaluating *condition*. Thus the first statement specifies initialization for the loop; the condition specifies a test, made before each iteration, such that the loop is exited when the condition becomes `false`; the expression often specifies incrementing that is done after each iteration. The condition is converted to `bool` (4.13).

2 Either or both of the condition and the expression can 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 (3.7.2) (named objects or temporaries) that are declared in that scope, in the reverse order of their declaration. Transfer out of a loop, out of a block, or back past an initialized variable

with automatic storage duration involves the destruction of variables with automatic storage duration that are in scope at the point transferred from but not at the point transferred to. (See 6.7 for transfers into blocks). However, the program can be terminated (by calling `exit()` or `abort()`(18.3), for example) without destroying class objects with automatic storage duration.

6.6.1 The `break` statement [stmt.break]

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

6.6.2 The `continue` statement [stmt.cont]

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

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

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

6.6.3 The `return` statement [stmt.return]

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

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

6.6.4 The `goto` statement [stmt.goto]

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

6.7 Declaration statement [stmt.dcl]

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

declaration-statement:
  declaration

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

2 Variables with automatic storage duration (3.7.2) are initialized each time their *declaration-statement* is executed. Variables with automatic storage duration declared in the block are destroyed on exit from the block (6.6).

3 It is possible to transfer into a block, but not in a way that bypasses declarations with initialization. A program that jumps from a point where a local variable with automatic storage duration is not in scope to a point where it is in scope is ill-formed unless the variable has pointer or arithmetic type or is an aggregate (8.5.1), and is declared without an *initializer* (8.5). For example,

```
void f()
{
    // ...
    goto lx;    // ill-formed: jump into scope of 'a'
    // ...
ly:
    X a = 1;
    // ...
lx:
    goto ly;    // ok, jump implies destructor
                // call for 'a' followed by construction
                // again immediately following label ly
}
```

4 The default initialization to zero (8.5) of all local objects with static storage duration (3.7.1) is performed before any other initialization takes place. A local object with static storage duration (3.7.1) initialized with a *constant-expression* is initialized before its block is first entered. A local object with static storage duration not initialized with a *constant-expression* is initialized the first time control passes completely through its declaration. If the initialization exits by throwing an exception, the initialization is not complete, so it will be tried again the next time the function is called.

5 The destructor for a local object with static storage duration will be executed if and only if the variable was constructed. The destructor is called either immediately before or as part of the calls of the `atexit()` functions (18.3). Exactly when is unspecified.

6.8 Ambiguity resolution [stmt.ambig]

1 There is an ambiguity in the grammar involving *expression-statement*s and *declaration*s: An *expression-statement* with a function-style explicit type conversion (5.2.3) as its leftmost subexpression can be indistinguishable from a *declaration* where the first *declarator* starts with a `(`. In those cases the *statement* is a *declaration*.

2 To disambiguate, the whole *statement* might have to be examined to determine if it is an *expression-statement* or a *declaration*. This disambiguates many examples. 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 3.4. A declaration that declares a function or defines a class, namespace, template, or function also has one or more scopes nested within it. These nested scopes, in turn, can have declarations nested within them. Unless otherwise stated, utterances in this 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* can be omitted only when declaring a class (9), enumeration (7.2) or namespace (7.3.1), that is, when the *decl-specifier-seq* contains either a *class-specifier*, an *elaborated-type-specifier* with a *class-key* (9.1), an *enum-specifier*, or a *namespace-definition*. In these cases and whenever a *class-specifier*, *enum-specifier*, or *namespace-definition* is present in the *decl-specifier-seq*, the identifiers in these specifiers are among the names being declared by the declaration (as *class-names*, *enum-names*, *enumerators*, or *namespace-name*, depending on the syntax).

4 Each *init-declarator* in the *init-declarator-list* contains exactly one *declarator-id*, which is the name declared by that *init-declarator* and hence one of the names declared by the declaration. The *type-specifiers* (7.1.5) in the *decl-specifier-seq* and the recursive *declarator* structure of the *init-declarator* describe a type (8.3), which is then associated with the name being declared by the *init-declarator*.

5 If the *decl-specifier-seq* contains the `typedef` specifier, the declaration is called a *typedef declaration* and the name of each *init-declarator* is declared to be a *typedef-name*, synonymous with its associated type (7.1.3). If the *decl-specifier-seq* contains no `typedef` specifier, the declaration is called a *function declaration* if the type associated with the name is a function type (8.3.5) and an *object declaration* otherwise.

6 Syntactic components beyond those found in the general form of declaration are added to a function declaration to make a *function-definition*. An object declaration, however, is also a definition unless it contains the `extern` specifier and has no initializer (3.1). A definition causes the appropriate amount of storage to be reserved and any appropriate initialization (8.5) to be done.

7 Only in *function-definitions* (8.4) and in function declarations for constructors, destructors, and type conversions can the *decl-specifier-seq* be omitted.

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

7.1 Specifiers [dcl.spec]

1 The specifiers that can be used in a declaration are

decl-specifier:
  storage-class-specifier
  type-specifier
  function-specifier
  `friend`
  `typedef`

decl-specifier-seq:
  decl-specifier-seq$_{opt}$ *decl-specifier*

2 The longest sequence of *decl-specifier*s that could possibly be a type name is taken as the *decl-specifier-seq* of a *declaration*. The sequence shall 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` shall 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* shall appear in a given *decl-specifier-seq*. If a *storage-class-specifier* appears in a *decl-specifier-seq*, there can be no `typedef` specifier in the same *decl-specifier-seq* and the *init-declarator-list* of the declaration shall not be empty. The *storage-class-specifier* applies to the name declared by each *init-declarator* in the list and not to any names declared by other specifiers.

2 The `auto` or `register` specifiers can be applied only to names of objects declared in a block (6.3) or to function parameters (8.4). They specify that the named object has automatic storage duration (3.7.2). An object declared without a *storage-class-specifier* at block scope or declared as a function parameter has automatic storage duration by default. Hence, the `auto` specifier is almost always redundant and not often used; one use of `auto` is to distinguish a *declaration-statement* from an *expression-statement* (6.2) explicitly.

3 A `register` specifier has the same semantics as an `auto` specifier together with a hint to the compiler that the object so declared will be heavily used. The hint can be ignored and in most implementations it will be ignored if the address of the object is taken.

4 The `static` specifier can be applied only to names of objects and functions and to anonymous unions (9.6). There can be no `static` function declarations within a block, nor any `static` function parameters. A `static` specifier used in the declaration of an object declares the object to have static storage duration (3.7.1). A `static` specifier can be used in the declaration of class members and its effect is described in 9.5. A name declared with a `static` specifier in a scope other than class scope (3.3.5) has internal linkage. For a nonmember function, an `inline` specifier is equivalent to a `static` specifier for linkage purposes (3.5) unless the inline declaration matches a previous declaration of the function, in which case the function name retains the linkage of the previous declaration.

5 The `extern` specifier can be applied only to the names of objects and functions. The `extern` specifier cannot be used in the declaration of class members or function parameters. A name declared in namespace scope with the `extern` specifier has external linkage unless the declaration matches a previous declaration, in which case the name retains the linkage of the previous declaration. An object or function declared at block scope with the `extern` specifier has external linkage unless the declaration matches a visible declaration of namespace scope that has internal linkage, in which case the object or function has internal linkage and refers to the same object or function denoted by the declaration of namespace scope.[38)]

6 A name declared in a namespace scope without a *storage-class-specifier* has external linkage unless it has internal linkage because of a previous declaration and provided it is not declared `const`. Objects declared `const` and not explicitly declared `extern` have internal linkage.

7 The linkages implied by successive declarations for a given entity shall agree. That is, within a given scope, each declaration declaring the same object name or the same overloading of a function name shall imply the same linkage. Each function in a given set of overloaded functions can have a different linkage, however. For example,

```
static char* f(); // f() has internal linkage
char* f()         // f() still has internal linkage
    { /* ... */ }

char* g();        // g() has external linkage
static char* g()  // error: inconsistent linkage
    { /* ... */ }

void h();
inline void h();  // external linkage

inline void l();
void l();         // internal linkage

inline void m();
extern void m();  // internal linkage
```

38) Here, ''previously'' includes enclosing scopes. This implies that a name specified `static` and then specified `extern` in an inner scope still has internal linkage.

```
static void n();
inline void n();  // internal linkage

static int a;     // 'a' has internal linkage
int a;            // error: two definitions

static int b;     // 'b' has internal linkage
extern int b;     // 'b' still has internal linkage

int c;            // 'c' has external linkage
static int c;     // error: inconsistent linkage

extern int d;     // 'd' has external linkage
static int d;     // error: inconsistent linkage
```

8 The name of a declared but undefined class can be used in an `extern` declaration. Such a declaration, however, cannot be used before the class has been defined. For example,

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

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

The `mutable` specifier can be applied only to names of class data members (9.2) and can not be applied to names declared `const` or `static`. For example

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

9 The `mutable` specifier on a class data member nullifies a `const` specifier applied to the containing class object and permits modification of the mutable class member even though the rest of the object is *const* (7.1.5.1).

7.1.2 Function specifiers [dcl.fct.spec]

1 *Function-specifiers* can be used only in function declarations.

function-specifier:
    `inline`
    `virtual`
    `explicit`

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 can be ignored. The `inline` specifier shall not appear on a block scope function declaration. For the linkage of inline functions, see 3.5 and 7.1.1. A function (8.3.5, 9.4, 11.4) defined within the class definition is inline by default.

3 An inline function shall be defined in every translation unit in which it is used (3.2), and shall have exactly the same definition in every case (see one definition rule, 3.2). If a function with external linkage is declared inline in one translation unit, it shall be declared inline in all translation units in which it appears. A call to an inline function shall not precede its definition. For example:

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

void k(X* p)
{
    int i = p->f();
    int j = p->g();  // A call appears before X::g is defined
                     // ill-formed
    // ...
}

inline int X::f()    // Declares X::f as an inline function
                     // A call appears before X::f is defined
                     // ill-formed
{
    // ...
}

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

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

5 The `explicit` specifier shall be used only in declarations of constructors within a class declaration; see 12.3.1.

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 (3.8.1) or compound (3.8.2) types. The `typedef` specifier shall not be used in a *function-definition* (8.4), and it shall not be combined in a *decl-specifier-seq* with any other kind of specifier except a *type-specifier*.

typedef-name:
    identifier

A name declared with the `typedef` specifier becomes a *typedef-name*. Within the scope of its declaration, a *typedef-name* is syntactically equivalent to a keyword and names the type associated with the identifier in the way described in 8. 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 can 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 shall 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 shall not be declared with the same name as a *typedef-name* that is declared in that scope and refers to a type other than the class 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* shall not be used after a `class`, `struct`, or `union` prefix and not in the names for constructors and destructors within the class declaration itself. 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 (3.5), the first *typedef-name* declared by the declaration is used to denote the class type in place of the dummy name. For example,

```
typedef struct { } S, R; // 'S' is the class name for linkage purposes
```

The *typedef-name* is still only a synonym for the dummy name and shall 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;
```

If an unnamed class is defined in a `typedef` declaration but the declaration does not declare a class type, the name of the class for linkage purposes is a dummy name. For example,

```
typedef struct { }* ps; // 'ps' is not the linkage name of the class
```

6 A *typedef-name* that names an enumeration is an *enum-name* (7.2). The *typedef-name* shall not be used after an `enum` prefix.

7.1.4 The `friend` specifier [dcl.friend]

1 The `friend` specifier is used to specify access to class members; see 11.4.

7.1.5 Type specifiers [dcl.type]

1 The type-specifiers are

> *type-specifier:*
> *simple-type-specifier*
> *class-specifier*
> *enum-specifier*
> *elaborated-type-specifier*
> *cv-qualifier*

As a general rule, at most one *type-specifier* is allowed in the complete *decl-specifier-seq* of a *declaration*. The only exceptions to this rule are the following:

2

— `const` or `volatile` can be combined with any other *type-specifier*. However, redundant cv-qualifiers are prohibited except when introduced through the use of typedefs (7.1.3) or template type arguments (14.7), in which case the redundant cv-qualifiers are ignored.

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

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

— `long` can be combined with `double`.

3 At least one *type-specifier* is required in a typedef declaration. At least one *type-specifier* is required in a function declaration unless it declares a constructor, destructor or type conversion operator. 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. Regarding the prohibition of the default `int` specifier in `typedef` declarations, see 7.1.3; in all other instances, the use of *decl-specifier-seq*s which contain no *simple-type-specifier*s (and thus default to plain `int`) is deprecated.

4 *class-specifier*s and *enum-specifier*s are discussed in 9 and 7.2, respectively. The remaining *type-specifier*s are discussed in the rest of this section.

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

1 There are two *cv-qualifiers*, `const` and `volatile`. 3.8.3 describes how cv-qualifiers affect object and function types. *

2 Unless explicitly declared `extern`, a `const` object does not have external linkage and shall be initialized (8.5; 12.1). An integral `const` object initialized by an integral constant expression can be used in integral constant expressions (5.19).

3 CV-qualifiers are supported by the type system so that they cannot be subverted without casting (5.2.10). A pointer or reference to a cv-qualified type need not actually point or refer to a cv-qualified object, but it is treated as if it does; a const-qualified access path cannot be used to modify an object even if the object referenced is a non-const object and can be modified through some other access path.

4 Except that any class member declare `mutable` (7.1.1) can be modified, any attempt to modify a `const` object during its lifetime (3.7) results in undefined behavior.

5 Example

```
const int ci = 3;   // cv-qualified (initialized as required)
ci = 4;             // ill-formed: attempt to modify const

int i = 2;          // not cv-qualified
const int* cip;     // pointer to const int
cip = &i;           // okay: cv-qualified access path to unqualified
*cip = 4;           // ill-formed: attempt to modify through ptr to const
```

```
int* ip;
ip = const_cast<int*> cip; // cast needed to convert const int* to int*
*ip = 4;            // defined: *ip points to i, a non-const object

const int* ciq = new const int (3); // initialized as required
int* iq = const_cast<int*> ciq;     // cast required
iq = 4;             // undefined: modifies a const object
```

6 Example

```
class X {
    public:
        mutable int i;
        int j;
};
class Y { public: X x; }

const Y y;
y.x.i++;        // well-formed: mutable member can be modified
y.x.j++;        // ill-formed: const-qualified member modified
Y* p = const_cast<Y*>(&y);      // cast away const-ness of y
p->x.i = 99;    // well-formed: mutable member can be modified
p->x.j = 99;    // undefined: modifies a const member
```

7 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 might be changed by means undetectable by a compiler.

Box 33
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 (3.8.1). Table 7 summarizes the valid combinations of *simple-type-specifier*s and the types they specify.

Table 7—*simple-type-specifier*s and the types they specify

Specifier(s)	Type
type-name	the type named
`char`	“`char`”
`unsigned char`	“`unsigned char`”
`signed char`	“`signed char`”
`bool`	“`bool`”
`unsigned`	“`unsigned int`”
`unsigned int`	“`unsigned int`”
`signed`	“`int`”
`signed int`	“`int`”
`int`	“`int`”
`unsigned short int`	“`unsigned short int`”
`unsigned short`	“`unsigned short int`”
`unsigned long int`	“`unsigned long int`”
`unsigned long`	“`unsigned long int`”
`signed long int`	“`long int`”
`signed long`	“`long int`”
`long int`	“`long int`”
`long`	“`long int`”
`signed short int`	“`short int`”
`signed short`	“`short int`”
`short int`	“`short int`”
`short`	“`short int`”
`wchar_t`	“`wchar_t`”
`float`	“`float`”
`double`	“`double`”
`long double`	“`long double`”
`void`	“`void`”

When multiple *simple-type-specifiers* are allowed, they can be freely intermixed with other *decl-specifiers* in any order. It is implementation-defined whether bit-fields and objects of `char` type are represented as signed or unsigned quantities. The `signed` specifier forces `char` objects and bit-fields to be signed; it is redundant with other integral types.

7.1.5.3 Elaborated type specifiers [dcl.type.elab]

1 Generally speaking, the *elaborated-type-specifier* is used to refer to a previously declared *class-name* or *enum-name* even though the name can be hidden by an intervening object, function, or enumerator declaration (3.3), but in some cases it also can be used to declare a *class-name*.

elaborated-type-specifier:
  class-key `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *identifier*
  `enum` *::*$_{opt}$ *nested-name-specifier*$_{opt}$ *identifier*

class-key:
  `class`
  `struct`
  `union`

2 If an *elaborated-type-specifier* is the sole constituent of a declaration, the declaration is ill-formed unless it has one of the following forms:

— *class-key identifier* `;`

in which case the *elaborated-type-specifier* declares the *identifier* to be a class-name in the scope that contains the declaration (9.1);

3 — `friend` *class-key identifier* `;`

in which case the *elaborated-type-specifier* declares the *identifier* to be a class-name in the smallest enclosing non-class, non-function prototype scope that contains the declaration;

4 — `friend` *class-key ::identifier* `;`
`friend` *class-key nested-name-specifier identifier* `;`

in which case the *identifier* is resolved as when the *elaborated-type-specifier* is not the sole constituent of a declaration.

5 If the *elaborated-type-specifier* is not the sole constituent of the declaration, the *identifier* following the *class-key* or `enum` keyword is resolved as described in 3.4 according to its qualifications, if any, but ignoring any objects, functions, or enumerators that have been declared. If the *identifier* resolves to a *class-name* or *enum-name*, the *elaborated-type-specifier* introduces it into the declaration the same way a *simple-type-specifier* introduces its *type-name*. If the *identifier* resolves to a *typedef-name*, the *elaborated-type-specifier* is ill-formed. If the resolution is unsuccessful, the *elaborated-type-specifier* is ill-formed unless it is of the simple form *class-key identifier*. In this case, the *identifier* is declared in the smallest non-class, non-function prototype scope that contains the declaration.

6 The *class-key* or `enum` keyword present in the *elaborated-type-specifier* shall agree in kind with the declaration to which the name in the *elaborated-type-specifier* refers. This rule also applies to the form of *elaborated-type-specifier* that declares a *class-name* or `friend` class since it can be construed as referring to the definition of the class. Thus, in any *elaborated-type-specifier*, the `enum` keyword shall be used to refer to an enumeration (7.2), the `union` *class-key* shall be used to refer to a union (9), and either the `class` or `struct` *class-key* shall be used to refer to a structure (9) or to a class declared using the `class` *class-key*. For example:

```
struct Node {
        struct Node* Next;      // ok: Refers to Node at global scope
        struct Data* Data;      // ok: Declares type Data
                                // at global scope and member Data
};

struct Data {
        struct Node* Node;      // ok: Refers to Node at global scope
        friend struct ::Glob;   // error: Glob is not declared
                                // cannot introduce a qualified type
        friend struct Glob;     // ok: Declares Glob in global scope
        /* ... */
};

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

        friend class ::Data;            // ok: global Data is a friend
        struct Data { /* ... */ };      // Defines nested Data

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

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

struct ::Data;          // error: cannot introduce a qualified type
struct Base::Data;      // error: cannot introduce a qualified type
struct Base::Datum;     // error: Datum undefined

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

7.2 Enumeration declarations [dcl.enum]

1 An enumeration is a distinct type (3.8.1) 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 can appear wherever constants are required. If no *enumerator-definition*s with = appear, then the values of the corresponding constants begin at zero and increase by one as the *enumerator-list* is read from left to right. An *enumerator-definition* with = gives the associated *enumerator* the value indicated by the *constant-expression*; subsequent *enumerator*s without initializers continue the progression from the assigned value. The *constant-expression* shall be of integral 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`,[39] 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 $\max(abs(e_{\mathrm{min}}), abs(e_{\mathrm{max}}))$ of the form $2^M - 1$; b_{min} is zero if e_{min} is non-negative and $-(b_{\mathrm{max}} + 1)$ otherwise. It is possible to define an enumeration that has values not defined by any of its enumerators.

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

Box 34
Shouldn't this be the same *underlying type*?

8 The value of an enumerator or an object of an enumeration type is converted to an integer by integral promotion (4.5). For example,

```
enum color { red, yellow, green=20, blue };
color col = red;
color* cp = &col;
if (*cp == blue) // ...
```

makes `color` a type describing various colors, and then declares `col` as an object of that type, and `cp` as a pointer to an object of that type. The possible values of an object of type `color` are `red`, `yellow`, `green`, `blue`; these values can be converted to the integral values `0`, `1`, `20`, and `21`. Since enumerations are distinct types, objects of type `color` can 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.

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

9 An expression of arithmetic type or of type `wchar_t` can be converted to an enumeration type explicitly. The value is unchanged if it is in the range of enumeration values of the enumeration type; otherwise the resulting enumeration value is unspecified.

Box 35
This means the program does not crash.

10 The enum-name and each enumerator declared by an enum-specifier is declared in the scope that immediately contains the enum-specifier. These names obey the scope rules defined for all names in (3.3) and (3.4). An enumerator declared in class scope can be referred to using the class member access operators (`::`, `.` (dot) and `->` (arrow)), see 5.2.4. For example,

```
class X {
public:
    enum direction { left='l', right='r' };
    int f(int i)
        { return i==left ? 0 : i==right ? 1 : 2; }
};

void g(X* p)
{
    direction d;        // error: `direction' not in scope
    int i;
    i = p->f(left);     // error: `left' not in scope
    i = p->f(X::right); // ok
    i = p->f(p->left);  // ok
    // ...
}
```

7.3 Namespaces [basic.namespace]

1 A namespace is an optionally-named declarative region. The name of a namespace can be used to access entities declared in that namespace; that is, the members of the namespace. Unlike other declarative regions, the definition of a namespace can be split over several parts of one or more translation units.

2 A name declared outside all named namespaces, blocks (6.3) and classes (9) has global namespace scope (3.3.4).

7.3.1 Namespace definition [namespace.def]

1 The grammar for a *namespace-definition* is

original-namespace-name:
  identifier

namespace-definition:
  named-namespace-definition
  unnamed-namespace-definition

named-namespace-definition:
  original-namespace-definition
  extension-namespace-definition

original-namespace-definition: *
  `namespace` *identifier* `{` *namespace-body* `}`

extension-namespace-definition:
  `namespace` *original-namespace-name* `{` *namespace-body* `}`

unnamed-namespace-definition:
  `namespace` `{` *namespace-body* `}`

namespace-body:
  declaration-seq$_{opt}$

2 The *identifier* in an *original-namespace-definition* shall not have been previously defined in the declarative region in which the *original-namespace-definition* appears. The *identifier* in an *original-namespace-definition* is the name of the namespace. Subsequently in that declarative region, it is treated as an *original-namespace-name*.

3 The *original-namespace-name* in an *extension-namespace-definition* shall have previously been defined in an *original-namespace-definition* in the same declarative region.

4 Every *namespace-definition* shall appear in the global scope or in a namespace scope (3.3.4).

7.3.1.1 Explict qualification [namespace.qual]

Box 36
The information in this section is very similar to the information provided in section 3.3.7. The information should probably be consolidated in one place.

1 A name in a class or namespace can be accessed using qualification according to the grammar:

id-expression:
  unqualified-id
  qualified-id

nested-name-specifier:
  class-or-namespace-name `::` *nested-name-specifier*$_{opt}$

class-or-namespace-name:
  class-name
  namespace-name

namespace-name:
  original-namespace-name
  namespace-alias

2 The *namespace-name*s in a *nested-name-specifier* shall have been previously defined by a *named-namespace-definition* or a *namespace-alias-definition*. *

3 The search for the initial qualifier preceding any `::` operator locates only the names of types or namespaces. The search for a name after a `::` locates only names members of a namespace or class. In particular, *using-directive*s (7.3.4) are ignored, as is any enclosing declarative region.

7.3.1.2 Unnamed namespaces [namespace.unnamed]

1 An *unnamed-namespace-definition* behaves as if it were replaced by

```
namespace unique { namespace-body }
using namespace unique;
```

where, for each translation unit, all occurrences of ***unique*** in that translation unit are replaced by an identifier that differs from all other identifiers in the entire program.[40)] For example:

```
namespace { int i; }        // unique::i
void f() { i++; }           // unique::i++

namespace A {
        namespace {
                int i;      // A::unique::i
                int j;      // A::unique::j
        }
        void g() { i++; }   // A::unique::i++
}

using namespace A;
void h() {
        i++;                // error: unique::i or A::unique::i
        A::i++;             // error: A::i undefined
        j++;                // A::unique::j
}
```

7.3.1.3 Namespace scope [namespace.scope]

1 The declarative region of a *namespace-definition* is its *namespace-body*. The potential scope denoted by an *original-namespace-name* is the concatenation of the declarative regions established by each of the *namespace-definition*s in the same declarative region with that *original-namespace-name*. Entities declared in a *namespace-body* are said to be *member*s of the namespace, and names introduced by these declarations into the declarative region of the namespace are said to be *member names* of the namespace. For example

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

namespace { int k=1; }

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

[40)] 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.

```
        int i;                  // error: duplicate definition

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

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

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

2 Because a *namespace-definition* contains *declaration*s in its *namespace-body* and a *namespace-definition* is itself a *declaration*, it follows that *namespace-definition*s can be nested. For example:

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

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

7.3.1.4 Namespace member definitions [namespace.memdef]

1 Members of a namespace can be defined within that namespace. For example:

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

2 Members of a named namespace can also be defined outside that namespace by explicit qualification (7.3.1.1) of the name being defined, provided that the entity being defined was already declared in the namespace and the definition appears after the point of declaration in a namespace that encloses the declaration's namespace. For example:

```
namespace Q {
        namespace V {
                void f();
        }
        void V::f() { /* ... */ }  // fine
        void V::g() { /* ... */ }  // error: g() is not yet a member of V
        namespace V {
                void g();
        }
}

namespace R {
        void Q::V::g() { /* ... */ } // error: R doesn't enclose Q
}
```

3 Every name first declared in a namespace is a member of that namespace. A `friend` function first declared within a class is a member of the innermost enclosing namespace. For example:

```
// Assume f and g have not yet been defined.
namespace A {
        class X {
                friend void f(X);  // declaration of f
                class Y {
                        friend void g();
                };
        };

        void f(X) { /* ... */}     // definition of f declared above
        X x;
        void g() { f(x); }         // f and g are members of A
}

using A::x;

void h()
{
        A::f(x);
        A::X::f(x);    // error: f is not a member of A::X
        A::X::Y::g();  // error: g is not a member of A::X::Y
}
```

The scope of class names first introduced in *elaborated-type-specifiers* is described in (7.1.5.3).

4 When an entity declared with the `extern` specifier is not found to refer to some other declaration, then that entity is a member of the innermost enclosing namespace. However such a declaration does not introduce the member name in its namespace scope. For example:

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

        void middle()
        {
                q();              // error: q not yet declared
        }

        void q() { /* ... */ }    // definition of X::q
}

void q() { /* ... */ }            // some other, unrelated q
```

7.3.2 Namespace or class alias [namespace.alias]

1 A *namespace-alias-definition* declares an alternate name for a namespace according to the following grammar:

namespace-alias:
    identifier

namespace-alias-definition:
    `namespace` *identifier* = *qualified-namespace-specifier ;*

qualified-namespace-specifier:
    $::_{opt}$ *nested-name-specifier*$_{opt}$ *class-or-namespace-name*

2 The *identifier* in a *namespace-alias-definition* is a synonym for the name of the namespace denoted by the *qualified-namespace-specifier* and becomes a *namespace-alias*.

3 In a declarative region, a *namespace-alias-definition* can be used to redefine a *namespace-alias* declared in that declarative region to refer to the namespace to which it already refers. For example, the following declarations are well-formed:

```
namespace Company_with_very_long_name { /* ... */ }
namespace CWVLN = Company_with_very_long_name;
namespace CWVLN = Company_with_very_long_name;  // ok: duplicate
namespace CWVLN = CWVLN;
```

4 A *namespace-name* shall not be declared as the name of any other entity in the same declarative region. A *namespace-name* defined at global scope shall not be declared as the name of any other entity in any global scope of the program. No diagnostic is required for a violation of this rule by declarations in different translation units.

7.3.3 The `using` declaration [namespace.udecl]

1 A *using-declaration* introduces a name into the declarative region in which the *using-declaration* appears. That name is a synonym for the name of some entity declared elsewhere.

using-declaration:
  `using ::`$_{opt}$ *nested-name-specifier unqualified-id* `;`
  `using ::` *unqualified-id* `;`

Box 37
There is still an open issue regarding the "opt" on the nested-name-specifier.

2 The member names specified in a *using-declaration* are declared in the declarative region in which the *using-declaration* appears.

3 Every *using-declaration* is a *declaration* and a *member-declaration* and so can be used in a class definition. For example:

```
struct B {
        void f(char);
        void g(char);
};

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

4 A *using-declaration* used as a *member-declaration* shall refer to a member of a base class of the class being defined. For example:

```
class C {
        int g();
};

class D2 : public B {
        using B::f;  // ok: B is a base of D
        using C::g;  // error: C isn't a base of D2
};
```

5 A *using-declaration* for a member shall be a *member-declaration*. For example:

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

void f()
{
        using X::i;  // error: X::i is a class member
                     // and this is not a member declaration.
        using X::s;  // error: X::s is a class member
                     // and this is not a member declaration.
}
```

6 Members declared by a *using-declaration* can be referred to by explicit qualification just like other member names (7.3.1.1). In a *using-declaration*, a prefix `::` refers to the global namespace (as ever). For example: *

```
void f();

namespace A {
        void g();
}

namespace X {
        using ::f;   // global f
        using A::g;  // A's g
}

void h()
{
        X::f();      // calls ::f
        X::g();      // calls A::g
}
```

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

```
namespace A {
        int i;
}

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

class B {
        int i;
};

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

8 The entity declared by an *using-declaration* shall be known in the context using it according to its definition at the point of the *using-declaration*. Definitions added to the namespace after the *using-declaration* are not considered when a use of the name is made. For example: *

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

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

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

void bar()
{
        using A::f;         // f is a synonym for A::f;
                            // that is, for A::f(int) and A::f(char).
        f('a');             // calls f(char)
}
```

9 A name defined by a *using-declaration* is an alias for its original declarations so that the *using-declaration* does not affect the type, linkage or other attributes of the members referred to.

10 If the set of local declarations and *using-declaration*s for a single name are given in a declarative region, they shall all refer to the same entity, or all refer to functions. For example

```
namespace B {
        int i;
        void f(int);
        void f(double);
}

void g()
{
        int i;
        using B::i;       // error: i declared twice
        void f(char);
        using B::f;       // fine: each f is a function
}
```

11 If a local function declaration has the same name and type as a function introduced by a *using-declaration*, the program is ill-formed. For example:

```
namespace C {
        void f(int);
        void f(double);
        void f(char);
}

void h()
{
        using B::f;    // B::f(int) and B::f(double)
        using C::f;    // C::f(int), C::f(double), and C::f(char)
        f('h');        // calls C::f(char)
        f(1);          // error: ambiguous: B::f(int) or C::f(int) ?
        void f(int);   // error: f(int) conflicts with C::f(int)
}
```

12 When a *using-declaration* brings names from a base class into a derived class scope, member functions in the derived class override virtual member functions with the same name and argument types in a base class (rather than conflicting). For example:

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

struct D : B {
        using B::f;
        void f(int);   // ok: D::f(int) overrides B::f(int);

        using B::g;
        void g(char);  // ok

        using B::h;
        void h(int);   // error: D::h(int) conflicts with B::h(int)
};

void k(D* p)
{
        p->f(1);    // calls D::f(int)
        p->f('a');  // calls B::f(char)
        p->g(1);    // calls B::g(int)
        p->g('a');  // calls D::g(char)
}
```

Box 38

For `p->g(1)` to be unambiguous, the `D::g(int)` synonym for `B::g(int)` must take part in the overload resolution as if it was a member of `D`, though its type must be ''member of `B`.'' A proper phrasing for this is being prepared for a vote.

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

14 The alias created by the *using-declaration* has the usual accessibility for a *member-declaration*. For example:

```
class A {
private:
        void f(char);
public:
        void f(int);
protected:
        void g();
};

class B : public A {
        using A::f; // error: A::f(char) is inaccessible
public:
        using A::g; // B::g is a public synonym for A::g
};
```

15 Use of *access-declaration*s (11.3) is deprecated; member *using-declaration*s provide a better alternative.

7.3.4 Using directive [namespace.udir]

1 *using-directive:*
`using namespace ::`$_{opt}$ *nested-name-specifier*$_{opt}$ *namespace-name ;*

2 A *using-directive* specifies that the names in the namespace with the given *namespace-name*, including those specified by any *using-directive*s in that namespace, can be used in the scope in which the *using-directive* appears after the using directive, exactly as if the names from the namespace had been declared outside a namespace at the points where the namespace was defined. A *using-directive* does not add any members to the declarative region in which it appears. If a namespace is extended by an *extended-namespace-definition* after a *using-directive* is given, the additional members of the extended namespace can be used after the *extended-namespace-definition.*

3 The *using-directive* is transitive: if a namespace contains a *using-directive* that nominates a second namespace that itself contains *using-directive*s, the effect is as if the *using-directive*s from the second namespace also appeared in the first. In particular, a name in a namespace does not hide names in a second namespace which is the subject of a *using-directive* in the first namespace. For example:

```
namespace M {
        int i;
}

namespace N {
        int i;
        using namespace M;
}

void f()
{
        N::i = 7; // well-formed: M::i is not a member of N
        using namespace N;
        i = 7;    // error: both M::i and N::i are accessible
}
```

4 During overload resolution, all functions from the transitive search are considered for argument matching. An ambiguity exists if the best match finds two functions with the same signature, even if one might seem to ‘‘hide’’ the other in the *using-directive* lattice. For example:

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

int d1;            // ok: no conflict with D::d1

namespace E {
        int e;
        void f(int);
}

namespace D {       // namespace extension
        int d2;
        using namespace E;
        void f(int);
}
```

```
void f()
{
        d1++;       // error: ambiguous ::d1 or D::d1?
        ::d1++;     // ok
        D::d1++;    // ok
        d2++;       // ok: D::d2
        e++;        // ok: E::e
        f(1);       // error: ambiguous: D::f(int) or E::f(int)?
        f('a');     // ok: D::f(char)
}
```

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 (3.5) between C++ and non-C++ code fragments can be achieved using a *linkage-specification*:

linkage-specification:
  `extern` *string-literal* `{` *declaration-seq$_{opt}$* `}`
  `extern` *string-literal declaration*

declaration-seq:
  declaration
  declaration-seq declaration

The *string-literal* indicates the required linkage. The meaning of the *string-literal* is implementation 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 39

This example might 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* can occur only in namespace scope (3.3). A *linkage-specification* for a class applies to nonmember functions and objects declared within it. A *linkage-specification* for a function also applies to functions and objects declared within it. A linkage declaration with a string that is unknown to the implementation is ill-formed.

3 If a function has more than one *linkage-specification*, they shall agree; that is, they shall specify the same *string-literal*. Except for functions with C++ linkage, a function declaration without a linkage specification shall not precede the first linkage specification for that function. A function can be declared without a linkage specification after an explicit linkage specification has been seen; the linkage explicitly specified in the earlier declaration is not affected by such a function declaration.

4 At most one of a set of overloaded functions (13) with a particular name can have C linkage.

5 Linkage can be specified for objects. For example,

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

Functions and objects can be declared `static` or `inline` within the `{}` of a linkage specification. The linkage directive is ignored for a function or object with internal linkage (3.5). A function first declared in a linkage specification behaves as a function with external linkage. 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. Taking the address of a function whose linkage is other than C++ or C produces undefined behavior.

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 can have an initializer.

> *init-declarator-list:*
>   *init-declarator*
>   *init-declarator-list* `,` *init-declarator*
>
> *init-declarator:*
>   *declarator initializer*$_{opt}$

2 The two components of a *declaration* are the specifiers (*decl-specifier-seq*; 7.1) and the declarators (*init-declarator-list*). The specifiers indicate the fundamental type, storage class, or other properties of the objects and functions being declared. The declarators specify the names of these objects and functions and (optionally) modify the type with operators such as `*` (pointer to) and `()` (function returning). Initial values can also be specified in a declarator; initializers are discussed in 8.5 and 12.6.

3 Each *init-declarator* in a declaration is analyzed separately as if it was in a declaration by itself.[41)]

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* `)`

---

41) A declaration with several declarators is usually equivalent to the corresponding sequence of declarations each with a single declarator. That is

```
T  D1, D2, ... Dn;
```

is usually equvalent to

```
T  D1; T D2; ... T Dn;
```

where `T` is a *decl-specifier-seq* and each `Di` is a *init-declarator*. The exception occurs when one declarator modifies the name environment used by a following declarator, as in

```
struct S { ... };
S   S, T;  // declare two instances of struct S
```

which is not equivalent to

```
struct S { ... };
S   S;
S   T;   // error
```

ptr-operator:
  `*` *cv-qualifier-seq*$_{opt}$
  `&`
  `::`$_{opt}$ *nested-name-specifier* `*` *cv-qualifier-seq*$_{opt}$

cv-qualifier-seq:
  cv-qualifier cv-qualifier-seq$_{opt}$

cv-qualifier:
  `const`
  `volatile`

declarator-id:
  id-expression
  nested-name-specifier$_{opt}$ *type-name*

A *class-name* has special meaning in a declaration of the class of that name and when qualified by that name using the scope resolution operator `::` (5.1, 12.1, 12.4).

8.1 Type names [dcl.name]

1 To specify type conversions explicitly, and as an argument of `sizeof` or `new`, the name of a type shall be specified. This can be done with a *type-id*, which is syntactically a declaration for an object or function of that type that omits the name of the object or function.

type-id:
  type-specifier-seq abstract-declarator$_{opt}$

type-specifier-seq:
  type-specifier type-specifier-seq$_{opt}$

abstract-declarator:
  ptr-operator abstract-declarator$_{opt}$
  direct-abstract-declarator

direct-abstract-declarator:
  direct-abstract-declarator$_{opt}$ `(` *parameter-declaration-clause* `)` *cv-qualifier-seq*$_{opt}$ *exception-specification*$_{opt}$
  direct-abstract-declarator$_{opt}$ `[` *constant-expression*$_{opt}$ `]`
  `(` *abstract-declarator* `)`

It is possible to identify uniquely the location in the *abstract-declarator* where the identifier would appear if the construction were a declarator in a declaration. The named type is then the same as the type of the hypothetical identifier. For example,

```
int                 // int i
int *               // int *pi
int *[3]            // int *p[3]
int (*)[3]          // int (*p3i)[3]
int *()             // int *f()
int (*)(double)     // int (*pf)(double)
```

name respectively the types "integer," "pointer to integer," "array of 3 pointers to integers," "pointer to array of 3 integers," "function having no parameters and returning pointer to integer," and "pointer to function of `double` returning an integer."

2 A type can also be named (often more easily) by using a *typedef* (7.1.3).

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

8.2 Ambiguity resolution [dcl.ambig.res]

1 The ambiguity arising from the similarity between a function-style cast and a declaration mentioned in 6.8 can also occur in the context of a declaration. In that context, it surfaces as a choice between a function declaration with a redundant set of parentheses around a parameter name and an object declaration with a function-style cast as the initializer. Just as for statements, the resolution is to consider any construct that could possibly be a declaration a declaration. A declaration can be explicitly disambiguated by a nonfunction-style cast or a = to indicate initialization. For example,

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

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

2 The ambiguity arising from the similarity between a function-style cast and a *type-id* can occur in many different contexts. The ambiguity surfaces as a choice between a function-style cast expression and a declaration of a type. The resolution is that any construct that could possibly be a *type-id* in its syntactic context shall be considered a *type-id*.

3 For example,

```
#include <stddef.h>
char *p;
void *operator new(size_t, int);
void foo(int x)  {
        new (int(*p)) int;      // new-placement expression
        new (int(*[x]));        // new type-id
}
```

4 For example,

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

5 For example,

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

6 For example,

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

8.3 Meaning of declarators [dcl.meaning]

1 A list of declarators appears after an optional (7) *decl-specifier-seq* (7.1). Each declarator contains exactly one *declarator-id*; it names the identifier that is declared. A *declarator-id* shall be a simple *identifier*, except for the following cases: the declaration of some special functions (12.3, 12.4, 13.4), the definition of a member function (9.4), the definition of a static data member (9.5), the declaration of a friend function that is a member of another class (11.4). An `auto`, `static`, `extern`, `register`, `friend`, `inline`, `virtual`, or `typedef` specifier applies directly to each *declarator-id* in a *init-declarator-list*; the type specified for each *declarator-id* depends on both the *decl-specifier-seq* and its *declarator*.

2 Thus, a declaration of a particular identifier has the form

```
T D
```

where `T` is a *decl-specifier-seq* and `D` is a declarator. The following subsections give an inductive procedure for determining the type specified for the contained *declarator-id* by such a declaration.

3 First, the *decl-specifier-seq* determines a type. For example, in the declaration

```
int unsigned i;
```

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

```
T D
```

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

4 In a declaration `T D` where `D` is an unadorned identifier the type of this identifier is "`T`."

5 In a declaration `T D` where `D` has the form

```
( D1 )
```

the type of the contained *declarator-id* is the same as that of the contained *declarator-id* in the declaration

```
T D1
```

Parentheses do not alter the type of the embedded *declarator-id*, but they can alter the binding of complex declarators.

8.3.1 Pointers [dcl.ptr]

1 In a declaration `T D` where `D` has the form

`*` *cv-qualifier-seq*$_{opt}$ `D1`

and the type of the identifier in the declaration `T D1` is "*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 A reference may or may not require storage (3.7).

4 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 shall contain an *initializer* (8.5.3) except when the declaration contains an explicit `extern` specifier (7.1.1), is a class member (9.2) declaration within a class declaration, or is the declaration of an parameter or a return type (8.3.5); see 3.1. A reference shall be initialized to refer to a valid object or function. In particular, null references are prohibited; no diagnostic is required.

8.3.3 Pointers to members [dcl.mptr]

1 In a declaration `T D` where `D` has the form

`::`$_{opt}$ *nested-name-specifier* `:: *` *cv-qualifier-seq*$_{opt}$ `D1`

and the *nested-name-specifier* names a class, and the type of the identifier in the declaration `T D1` is "*type-modifier* `T`," then the type of the identifier of `D` is "*type-modifier cv-qualifier-seq* pointer to member of *class nested-name-specifier of type* `T`."

2 For example,

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

int X::* pmi = &X::a;
void (X::* pmf)(int) = &X::f;
double X::* pmd;
char Y::* pmc;
```

declares `pmi`, `pmf`, `pmd` and `pmc` to be a pointer to a member of `X` of type `int`, a pointer to a member of `X` of type `void(int)`, a pointer to a member of `X` of type `double` and a pointer to a member of `Y` of type `char` respectively. The declaration of `pmd` is well-formed even though `X` has no members of type `double`. Similarly, the declaration of `pmc` is well-formed even though `Y` is an incomplete type. `pmi` and `pmf` can be used like this:

```
X obj;
//...
obj.*pmi = 7;     // assign 7 to an integer
                  // member of obj
(obj.*pmf)(7);    // call a function member of obj
                  // with the argument 7
```

3 Note that a pointer to member cannot point to a static member of a class (9.5), a member with reference type, or "*cv* `void`." There are no references to members. See also 5.5 and 5.3.

8.3.4 Arrays [dcl.array]

1 In a declaration `T D` where `D` has the form

`D1 [`*constant-expression*$_{opt}$`]`

and the type of the identifier in the declaration `T D1` is "*type-modifier* `T`," then the type of the identifier of `D` is an array type. `T` shall not be a reference type, an incomplete type, or an abstract class type. If the *constant-expression* (5.19) is present, its value shall be greater than zero. The constant expression specifies the *bound* of (number of elements in) the array. If the value of the constant expression is `N`, the array has `N` elements numbered `0` to `N-1`, and the type of the identifier of `D` is "*type-modifier* array of `N T`." If the constant expression is omitted, the type of the identifier of `D` is "*type-modifier* array of unknown bound of `T`," an incomplete object type. The type "*type-modifier* array of `N T`" is a different type from the type "*type-modifier* array of unknown bound of `T`," see 3.8. 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 can be constructed from one of the fundamental types[42] (except `void`), from a pointer, from a pointer to member, from a class, or from another array.

3 When several "array of" specifications are adjacent, a multidimensional array is created; the constant expressions that specify the bounds of the arrays can be omitted only for the first member of the sequence. This elision is useful for function parameters of array types, and when the array is external and the definition, which allocates storage, is given elsewhere. The first *constant-expression* can also be omitted when the declarator is followed by an *initializer* (8.5). In this case the bound is calculated from the number of initial elements (say, `N`) supplied (8.5.1), and the type of the identifier of `D` is "array of `N T`."

4 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]` can reasonably appear in an expression.

5 Conversions affecting lvalues of array type are described in 4.2. Objects of array types cannot be modified, see 3.9.

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

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

appearance, subscripting is a commutative operation.

7 A consistent rule is followed for multidimensional arrays. If `E` is an n-dimensional array of rank $i \times j \times \cdots \times k$, then `E` appearing in an expression is converted to a pointer to an $(n-1)$-dimensional array with rank $j \times \cdots \times k$. If the `*` operator, either explicitly or implicitly as a result of subscripting, is applied to this pointer, the result is the pointed-to $(n-1)$-dimensional array, which itself is immediately converted into a pointer.

8 For example, consider

```
int x[3][5];
```

Here `x` is a 3×5 array of integers. When `x` appears in an expression, it is converted to a pointer to (the first of three) five-membered arrays of integers. In the expression `x[i]`, which is equivalent to `*(x+i)`, `x` is first converted to a pointer as described; then `x+i` is converted to the type of `x`, which involves multiplying `i` by the length of the object to which the pointer points, namely five integer objects. The results are added and indirection applied to yield an array (of five integers), which in turn is converted to a pointer to the first of the integers. If there is another subscript the same argument applies again; this time the result is an integer.

9 It follows from all this that arrays in C++ are stored row-wise (last subscript varies fastest) and that the first subscript in the declaration helps determine the amount of storage consumed by an array but plays no other part in subscript calculations.

8.3.5 Functions [dcl.fct]

1 In a declaration `T D` where `D` has the form

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

and the type of the contained *declarator-id* in the declaration `T D1` is "*type-modifier* `T1`," the type of the *declarator-id* in `D` is "*type-modifier cv-qualifier-seq*$_{opt}$ function with parameters of type *parameter-declaration-clause* and returning `T1`"; a type of this form is a *function type*[43].

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` shall not be a parameter type (though types derived from `void`, such as `void*`, can). Where syntactically correct, "`, ...`" is synonymous with "`...`". The standard header `<cstdarg>` contains a mechanism for accessing arguments passed using the ellipsis (see 5.2.2 and 18.7).

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

3 A single name can be used for several different functions in a single scope; this is function overloading (13). All declarations for a function with a given parameter list shall agree exactly both in the type of the value returned and in the number and type of parameters; the presence or absence of the ellipsis is considered part of the function type. The type of each parameter is determined from its own *decl-specifier-seq* and *declarator*. After determining the type of each parameter, any parameter of type "array of `T`" or "function returning `T`" is adjusted to be "pointer to `T`" or "pointer to function returning `T`," respectively. After producing the list of parameter types, several transformations take place upon the types. Any *cv-qualifier* modifying a parameter type is deleted; e.g., the type `void(const int)` becomes `void(int)`. Such *cv-qualifier*s affect only the definition of the parameter within the body of the function. If the *storage-class-specifier* `register` modifies a parameter type, the specifier is deleted; e.g., `register char*` becomes `char*`. Such *storage-class-qualifier*s affect only the definition of the parameter within the body of the function. The resulting list of transformed parameter types is the function's *parameter type list*.

Box 40

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

The return type and the parameter type list, but not the default arguments (8.3.6), are part of the function type. If the type of a parameter includes a type of the form "pointer to array of unknown bound of `T`" or "reference to array of unknown bound of `T`," the program is ill-formed.[44] A *cv-qualifier-seq* can only be part of a declaration or definition of a nonstatic member function, and of a pointer to a member function; see 9.4.2. It is part of the function type.

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

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

6 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 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.

[44] 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.

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 shall 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. Default arguments will be used in calls where trailing arguments are missing.

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 can be called in any of these ways:

```
point(1,2);  point(1);  point();
```

The last two calls are equivalent to `point(1,4)` and `point(3,4)`, respectively.

3 A default argument expression shall be specified only in the *parameter-declaration-clause* of a function declaration or in a *template-parameter* (14.6). If it is specified in a *parameter-declaration-clause*, it shall not occur within a *declarator* or *abstract-declarator* of a *parameter-declaration*.[45)]

Box 41

This restriction, voted in at the Valley Forge meeting, is expected to be reviewed at the Austin meeting. Mike Miller has promised a paper.

4 Default arguments can be added in later declarations of a function, but only in the same scope. Declarations in different scopes have completely distinct sets of default arguments. That is, declarations in inner scopes do not acquire default arguments from declarations in outer scopes, and vice versa. In a given function declaration, all parameters subsequent to a parameter with a default argument shall have default arguments supplied in this or previous declarations. A default argument shall not be redefined by a later declaration (not even to the same value). For example:

45) This means that default arguments cannot appear, for example, in declarations of pointers to functions, references to functions, or `typedef` declarations.

```
void f(int, int);
void f(int, int = 7);
void h()
{
    f(3);                         // ok, calls f(3, 7)
    void f(int = 1, int);         // error: does not use default
                                  // from surrounding scope
}
void m()
{
    void f(int, int);             // has no defaults
    f(4);                         // error: wrong number of arguments
    void f(int, int = 5);         // ok
    f(4);                         // ok, calls f(4, 5);
    void f(int, int = 5);         // error: cannot redefine, even to
                                  // same value
}
void n()
{
    f(6);                         // ok, calls f(6, 7)
}
```

Declarations of a given nonmember function in different translation units need not specify the same default arguments. Declarations of a given member function in different translation units, however, shall specify the same default arguments (the accumulated sets of default arguments at the end of the translation units shall be the same).

Box 42

This was decided on the basis of guesses regarding the One Definition Rule and should be reviewed once that section is finished.

5 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 (_class.inline_). In the following example, `g` will be called with the value `f(1)`:

```
int a = 1;
int f(int);
int g(int x = f(a)); // default argument: f(::a)

void h() {
    a = 2;
    {
        int a = 3;
        g();        // g(f(::a))
    }
}
```

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

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

`this` shall not be used in a default argument of a member function. For example,

```
class A {
    void f(A* p = this);        // error
};
```

6 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 shall not be used in default argument expressions, even if they are not evaluated. Parameters of a function declared before a default argument expression are in scope and can hide namespace and class member names. 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
int h(int a, int b = sizeof(a));  // error, parameter 'a' used
                                  // in default argument
```

7 Similarly, a nonstatic member shall not be used in a default argument expression, even if it is not evaluated, unless it appears as the id-expression of a class member access expression (5.2.4). For example, the declaration of `X::mem1()` in the following example is ill-formed because no object is supplied for the nonstatic member `X::a` used as an initializer.

```
int b;
class X {
    int a;
    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.

8 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
```

When a declaration of a function is introduced by way of a `using` declaration (7.3.3), any default argument information associated with the declaration is imported as well.

Box 43

Can additional default arguments be added to the function thereafter by way of redeclarations of the function? Can the function be redeclared in the namespace with added default arguments, and if so, are those added arguments visible to those who have imported the function via using?

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

10 A virtual function call (10.3) uses the default arguments in the declaration of the virtual function determined by the static type of the pointer or reference denoting the object. An overriding function in a derived class does not acquire default arguments from the function it overrides. For example,

```
struct A {
    virtual void f(int a = 7);
};
struct B : public A {
    void f(int a);
};
void m()
{
    B* pb = new B;
    A* pa = pb;
    pa->f();          // ok, calls pa->A::f(7)
    pb->f();          // error: wrong number of arguments for B::f()
}
```

8.4 Function definitions [dcl.fct.def]

1 Function definitions have the form

function-definition:
  decl-specifier-seq$_{opt}$ *declarator* *ctor-initializer*$_{opt}$ *function-body*

function-body:
  compound-statement

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

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

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

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

3 A simple example of a complete function definition is

```
int max(int a, int b, int c)
{
    int m = (a > b) ? a : b;
    return (m > c) ? m : c;
}
```

Here `int` is the *decl-specifier-seq*; `max(int a, int b, int c)` is the *declarator*; `{ /* ... */ }` is the *function-body*.

4 A *ctor-initializer* is used only in a constructor; see 12.1 and 12.6.

5 A *cv-qualifier-seq* can be part of a non-static member function declaration, non-static member function definition, or pointer to member function only; see 9.4.2. 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 can specify an initial value for the identifier being declared. The identifier designates an object or reference being initialized. The process of initialization described in the remainder of this subclause (8.5) applies also to initializations specified by other syntactic contexts, such as the initialization of function parameters with argument expressions (5.2.2) or the initialization of return values (6.6.3).

initializer:
    = *initializer-clause*
    (*expression-list*)

initializer-clause:
    assignment-expression
    { *initializer-list* ,$_{opt}$ }
    { }

initializer-list:
    initializer-clause
    initializer-list , *initializer-clause*

2 Automatic, register, static, and external variables of namespace scope can be initialized by arbitrary expressions involving constants and previously declared variables and functions.

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

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

4 The order of initialization of static objects is described in 3.6 and 6.7.

5 Variables with static storage duration (3.7) that are not initialized and do not have a user-declared constructor are guaranteed to start off as zero converted to the appropriate type. If the object is a `class` or `struct`, its nonstatic data members start off as zero converted to the appropriate type. If the object is a `union`, its first nonstatic 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.

6 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
```

7 The form of initialization (using parentheses or =) is generally insignificant, but does matter when the entity being initialized has a class type; see below. A parenthesized initializer can be a list of expressions only when the entity being initialized has a class type. *

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 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), `static_cast` expressions (5.2.8), functional notation type conversions (5.2.3), and base and member initializers (12.6.2) is equivalent to the form

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

10 The semantics of initializers are as follows. The *destination type* is the type of the object or reference being initialized and the *source type* is the type of the initializer expression. The source type is not defined when the initializer is brace-enclosed or when it is a parenthesized list of expressions.

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

— If the destination type is an array of characters or an array of `wchar_t`, and the initializer is a string literal, see 8.5.2.

— Otherwise, if the destination type is an array, see 8.5.1.

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

— Otherwise, if the destination type is a (possibly cv-qualified) class type and the initializer has the parenthesized form, constructors are considered. The applicable constructors are enumerated (13.2.1.4), and the best one is chosen through overload resolution (13.2). The constructor so selected is called to initialize the object, with the initializer expression(s) as its argument(s). If no constructor applies, or the overload resolution is ambiguous, the initialization is ill-formed.

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

— Otherwise, the initial value of the object being initialized is the (possibly converted) value of the initializer expression. Standard conversions (clause 4) will be used, if necessary, to convert the initializer expression to the unqualified version of the destination type; no user-defined conversions are considered. If the conversion cannot be done, the initialization is ill-formed. Note that an expression of type "*cv1* `T`" can initialize an object of type "*cv2* `T`" independently of the cv-qualifiers *cv1* and *cv2*. For example,

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

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* can 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.[46)]

2 For example,

```
struct S { int a; char* b; int c; };
S ss = { 1, "asdf" };
```

initializes `ss.a` with `1`, `ss.b` with `"asdf"`, and `ss.c` with zero.

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

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

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

5 For example,

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

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

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

is a completely-bracketed initialization: 1, 3, and 5 initialize the first row of the array `y[0]`, namely `y[0][0]`, `y[0][1]`, and `y[0][2]`. Likewise the next two lines initialize `y[1]` and `y[2]`. The initializer ends early and therefore `y[3]` is initialized with zeros. Precisely the same effect could have been achieved by

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

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

6 The initializer for `y` begins with a left brace, but the one for `y[0]` does not, therefore three elements from the list are used. Likewise the next three are taken successively for `y[1]` and `y[2]`. Also,

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

initializes the first column of `y` (regarded as a two-dimensional array) and leaves the rest zero.

7 Initialization of arrays of objects of a class with non-trivial constructors (12.1) is described in 12.6.1.

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

8.5.2 Character arrays [dcl.init.string]

1 A `char` array (whether plain `char`, `signed`, or `unsigned`) can be initialized by a string; a `wchar_t` array can be initialized by a wide-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 shall 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), shall be initialized by an object, or function, of type `T` or by an object that can be converted into a `T`. For example,

```
int g(int);
void f()
{
    int i;
    int& r = i;  // `r' refers to `i'
    r = 1;       // the value of `i' becomes 1
    int* p = &r; // `p' points to `i'
    int& rr = r; // `rr' refers to what `r' refers to,
                 // that is, to `i'
    int (&rg)(int) = g; // `rg' refers to the function `g'
    rg(i);              // calls function `g'
    int a[3];
    int (&ra)[3] = a;   // `ra' refers to the array `a'
    ra[1] = i;          // modifies `a[1]'
}
```

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

3 The initializer can be omitted for a reference only in a parameter declaration (8.3.5), in the declaration of a function return type, in the declaration of a class member within its class declaration (9.2), and where the `extern` specifier is explicitly used. For example,

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

4 Given types "*cv1* `T1`" and "*cv2* `T2`," "*cv1* `T1`" is *reference-related* to "*cv2* `T2`" if `T1` is the same type as `T2`, or `T1` is a base class of `T2`. "*cv1* `T1`" is *reference-compatible* with "*cv2* `T2`" if `T1` is reference-related to `T2` and *cv1* is the same cv-qualification as, or greater cv-qualification than, *cv2*. For purposes of overload resolution, cases for which *cv1* is greater cv-qualification than *cv2* are identified as *reference-compatible with added qualification* (see 13.2.3.2). In all cases where the reference-related or reference-compatible relationship of two types is used to establish the validity of a reference binding, and `T1` is a base class of `T2`, a program that necessitates such a binding is ill-formed if `T1` is an inaccessible (11) or ambiguous (10.2) base class of `T2`.

5 A reference to type "*cv1* `T1`" is initialized by an expression of type "*cv2* `T2`" as follows:

— If the initializer expression is an lvalue (but not an lvalue for a bit-field), and

6

  — "*cv1* `T1`" is reference-compatible with "*cv2* `T2`," or

  — the initializer expression can be implicitly converted to an lvalue of type "cv3 `T1`," where *cv3* is the same cv-qualification as, or lesser cv-qualification than, *cv1*, [47] then

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

7 the reference is bound directly to the initializer expression lvalue. Note that the usual lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) standard conversions are not needed, and therefore are suppressed, when such direct bindings to lvalues are done.

```
double d = 2.0;
double& rd = d;          // rd refers to `d'
const double& rcd = d;   // rcd refers to `d'

struct A { };
struct B : public A { } b;
A& ra = b;               // ra refers to A sub-object in `b'
const A& rca = b;        // rca refers to A sub-object in `b'
```

8 — Otherwise, the reference shall be to a non-volatile const type (i.e., *cv1* shall be `const`).

```
double& rd2 = 2.0;       // error: not an lvalue and reference
                         //   not const
int  i = 2;
double& rd3 = i;         // error: type mismatch and reference
                         //   not const
```

— If the initializer expression is an rvalue, with `T2` a class type, and "*cv1* `T1`" is reference-compatible with "*cv2* `T2`," the reference is bound in one of the following ways (the choice is implementation-defined):

— The reference is bound directly to the object represented by the rvalue (see 3.9) or to a sub-object within that object.

— A temporary of type "*cv1* `T2`" [sic] is created, and a copy constructor is called to copy the entire rvalue object into the temporary. The reference is bound to the temporary or to a sub-object within the temporary. [48)]

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

```
struct A { };
struct B : public A { } b;
extern B f();
const A& rca = f();      // Either bound directly or
                         //   the entire B object is copied and
                         //   the reference is bound to the
                         //   A sub-object of the copy
```

10 — Otherwise, a temporary of type "*cv1* `T1`" is created and initialized from the initializer expression using the rules for a non-reference initialization (8.5). The reference is then bound to the temporary. If `T1` is reference-related to `T2`, *cv1* must be the same cv-qualification as, or greater cv-qualification than, *cv2*; otherwise, the program is ill-formed.

```
const double& rcd2 = 2; // rcd2 refers to temporary
                        // with value `2.0'
const volatile int cvi = 1;
const int& r = cvi;     // error: type qualifiers dropped
```
*

11 12.2 describes the lifetime of temporaries bound to references.

48) Clearly, if the reference initialization being processed is one for the first argument of a copy constructor call, an implementation must eventually choose the direct-binding alternative to avoid infinite recursion.

9 Classes [class]

1 A class is a type. Its name becomes a *class-name* (9.1) within its scope.

class-name:
    identifier
    template-id

*Class-specifier*s and *elaborated-type-specifier*s (7.1.5.3) are used to make *class-name*s. An object of a class consists of a (possibly empty) sequence of members and base class objects.

class-specifier:
    class-head { *member-specification*$_{opt}$ }

class-head:
    class-key identifier$_{opt}$ *base-clause*$_{opt}$
    class-key nested-name-specifier identifier base-clause$_{opt}$

class-key:
    `class`
    `struct`
    `union`

2 The name of a class can be used as a *class-name* even within the *base-clause* and *member-specification* of the class specifier itself. A *class-specifier* is commonly referred to as a class definition. A class is considered defined after the closing brace of its *class-specifier* has been seen even though its member functions are in general not yet defined.

3 Objects of an empty class have a nonzero size.

Box 44
Bill Gibbons suggest that a base class subobject should be allowed to occupy zero bytes of the complete object. This would permit two base class subobjects to have the same address, for example.

4 Class objects can be assigned, passed as arguments to functions, and returned by functions (except objects of classes for which copying has been restricted; see 12.8). Other plausible operators, such as equality comparison, can be defined by the user; see 13.4.

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).

6 Aggregates of class type are described in 8.5.1. A *POD-struct*[49] is an aggregate class that has no members of type reference, pointer to member, non-POD-struct or non-POD-union. Similarly, a *POD-union* is an aggregate union that has no members of type reference, pointer to member, non-POD-struct or non-POD-union.

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

9.1 Class names [class.name]

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

```
struct X { int a; };
struct Y { int a; };
X a1;
Y a2;
int a3;
```

declares three variables of three different types. This implies that

```
a1 = a2;        // error: Y assigned to X
a1 = a3;        // error: int assigned to X
```

are type mismatches, and that

```
int f(X);
int f(Y);
```

declare an overloaded (13) function `f()` and not simply a single function `f()` twice. For the same reason,

```
struct S { int a; };
struct S { int a; };  // error, double definition
```

is ill-formed because it defines `S` twice.

2 A class definition introduces the class name into the scope where it is defined and hides any class, object, function, or other declaration of that name in an enclosing scope (3.3). If a class name is declared in a scope where an object, function, or enumerator of the same name is also declared, then when both declarations are in scope, the class can be referred to only using an *elaborated-type-specifier* (7.1.5.3). For example,

```
struct stat {
    // ...
};

stat gstat;              // use plain 'stat' to
                         // define variable

int stat(struct stat*); // redefine 'stat' as function

void f()
{
    struct stat* ps;     // 'struct' prefix needed
                         // to name struct stat
    // ...
    stat(ps);            // call stat()
    // ...
}
```

A *declaration* consisting solely of *class-key identifier ;* is either a redeclaration of the name in the current scope or a forward declaration of the identifier as a class name. It introduces the class name into the current scope. For example,

```
struct s { int a; };

void g()
{
    struct s;                // hide global struct 's'
    s* p;                    // refer to local struct 's'
    struct s { char* p; };  // declare local struct 's'
    struct s;                // receclaration, has no effect
}
```

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

```
class vector;

class matrix {
    // ...
    friend vector operator*(matrix&, vector&);
};

class vector {
    // ...
    friend vector operator*(matrix&, vector&);
};
```

Declaration of `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*, but shall not be used in an *elaborated-type-specifier*; see also 7.1.3.

9.2 Class members [class.mem]

member-specification:
  member-declaration member-specification$_{opt}$
  access-specifier **:** *member-specification*$_{opt}$

member-declaration:
  decl-specifier-seq$_{opt}$ *member-declarator-list*$_{opt}$ **;**
  function-definition **;**$_{opt}$
  qualified-id **;**
  using-declaration

member-declarator-list:
  member-declarator
  member-declarator-list **,** *member-declarator*

member-declarator:
  declarator pure-specifier$_{opt}$
  declarator constant-initializer$_{opt}$
  identifier$_{opt}$ **:** *constant-expression*

pure-specifier:
  `= 0`

constant-initializer:
= *constant-expression*

1 The *member-specification* in a class definition declares the full set of members of the class; no member can be added elsewhere. Members of a class are data members, member functions (9.4), nested types, and member constants. Data members and member functions are static or nonstatic; see 9.5. Nested types are classes (9.1, 9.8) and enumerations (7.2) defined in the class, and arbitrary types declared as members by use of a typedef declaration (7.1.3). The enumerators of an enumeration (7.2) defined in the class are member constants of the class. Except when used to declare friends (11.4) or to adjust the access to a member of a base class (11.3), *member-declaration*s declare members of the class, and each such *member-declaration* must declare at least one member name of the class. A member shall not be declared twice in the *member-specification*, except that a nested class can be declared and then later defined.

2 Note that a single name can denote several function members provided their types are sufficiently different (13).

3 A *member-declarator* can contain a *constant-initializer* only if it declares a `static` member (9.5) of integral or enumeration type, see 9.5.2.

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

5 A member shall not be `auto`, `extern`, or `register`.

6 The *decl-specifier-seq* can be omitted in constructor, destructor, and conversion function declarations only. The *member-declarator-list* can be omitted only after a *class-specifier*, an *enum-specifier*, or a *decl-specifier-seq* of the form `friend` *elaborated-type-specifier*. A *pure-specifier* shall be used only in the declaration of a virtual function (10.3).

7 Non-`static` (9.5) members that are class objects shall be objects of previously defined classes. In particular, a class `cl` shall not contain an object of class `cl`, but it can contain a pointer or reference to an object of class `cl`. When an array is used as the type of a nonstatic member all dimensions shall be specified.

8 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 might cause two adjacent members not to be allocated immediately after each other; so might requirements for space for managing virtual functions (10.3) and virtual base classes (10.1); see also 5.4.

10 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 shall not have the same name as its class.

11 Two POD-struct (9) types are layout-compatible if they have the same number of members, and corresponding members (in order) have layout-compatible types (3.8).

12 Two POD-union (9) types are layout-compatible if they have the same number of members, and corresponding members (in any order) have layout-compatible types (3.8).

Box 45
Shouldn't this be the same *set* of types?

13 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.

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

9.3 Scope rules for classes [class.scope0]

1 The following rules describe the scope of names declared in classes.

1) The scope of a name declared in a class consists not only of the 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` shal refer to the same declaration when re-evaluated in its context and in the completed scope of S.

3) If reordering member declarations in a class yields an alternate valid program under (1) and (2), the program's meaning is undefined.

4) A declaration in a nested declarative region hides a declaration whose declarative region contains the nested declarative region.

5) A declaration within a member function hides a declaration whose scope extends to or past the end of the member function's class.

6) The scope of a declaration that extends to or past the end of a class definition also extends to the regions defined by its member definitions, even if defined lexically outside the class (this includes static data member initializations, nested class definitions and member function definitions (that is, the *parameter-declaration-clause* including default arguments (8.3.6), the member function body and, for constructor functions (12.1), the ctor-initializer (12.6.2)).

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

Box 46
This subclause does not take into account inheritance. Should it?

2 Functions declared in the definition of a class (excluding those declared with a `friend` specifier; 11.4) are called member functions of that class. A member function may be declared `static` in which case it is a *static* member function of its class (9.5); otherwise it is a *nonstatic* member function of its class (9.4.1, 9.4.2).

3 A member function may be defined (8.4) in its class definition, in which case it is an *inline* member function, or it may be defined outside of its class definition if it has already been declared but not defined in its class definition. This *out-of-line* definition shall appear in a namespace scope containing the definition of the member function's class.

4 An `inline` member function (whether static or nonstatic) may also be defined outside of its class definition provided either its declaration in the class definition or its definition outside of the class definition declares the function as `inline`, see 7.1.2.

5 Member functions of a class in namespace scope have external linkage. Member functions of a local class (9.9) have no linkage. See 3.5.

6 There shall be exactly one definition of a non-inline member function in a program; no diagnostic is required. There may be more than one `inline` member function definition in a program. See 3.2 and 7.1.2.

7 If the definition of a member function is lexically outside its class definition, the member function name shall be qualified by its class name using the `::` operator. A member function definition (that is, the *parameter-declaration-clause* including the default arguments (8.3.6), the member function body and, for a constructor function (12.1), the ctor-initializer (12.6.2)) is in the scope of the member function's class (_class.scope0). For example,

```
struct X {
        typedef int T;
        static T count;
        void f(T);
};
void X::f(T t = count) { }
```

The member function `f` of class `X` is defined in global scope; the notation `X::f` specifies that the function `f` is a member of class `X` and in the scope of class `X`. In the function definition, the parameter type `T` refers to the typedef member CW T declared in class `X` and the default argument `count` refers to the static data member `count` declared in class `X`.

8 A `static` local variable in a member function always refers to the same object, whether or not the member function is `inline`.

9 Member functions may be mentioned in `friend` declarations after their class has been defined.

10 Member functions of a local class shall be defined inline in their class definition.

9.4.1 Nonstatic member functions [class.mfct.nonstatic]

1 A *nonstatic* member function may be called for an object of its class type using the class member access syntax (5.2.4, 13.2.1.1). A nonstatic member function may also be called directly from within the body of the member functions of its class using the function call syntax (5.2.2, 13.2.1.1). The effect of calling a nonstatic member function of a class `X` for something that is not an object of class `X` is undefined.

2 The names of a member of class `X` may be used directly in the body of a nonstatic member function of `X`. During name lookup, when an *id-expression* (5.1) used in a nonstatic member function body resolves to a nonstatic member of the member function's class, the *id-expression* is transformed into a class member access expression (5.2.4) using `(*this)` (9.4.2) as the *postfix-expression* to the left of the `.` operator. The member name then refers to the member of the object for which the function is called. Similarly during name look up, when an *unqualified-id* (5.1) used in the definition of a member function resolves to a `static` member, an enumerator or a nested type of member function's class, the *unqualified-id* is transformed into a *qualified-id* (5.1) in which the *nested-name-specifier* names the class of the member function. For example,

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

void tnode::set(char* w, tnode* l, tnode* r)
{
        count = strlen(w)+1;
        if (sizeof(tword)<=count)
                error("tnode string too long");
        strcpy(tword,w);
        left = l;
        right = r;
}

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

The member names `tword`, `count`, `left`, and `right` refer to members of the object for which the function is called. Thus, in the call ,`n1.set`(abc",&n2,0)" `tword` refers to `n1.tword`, and in the call ,`n2.set`(def",0,0)" it refers to `n2.tword`. The functions `strlen`, `error`, and `strcpy` are not members of the class `tnode` and shall be declared elsewhere.[50)]

3 The type of a nonstatic member function involves its class name; thus the type of the *qualified-id* expression `tnode::set` is member function and the type of `&tnode::set` is pointer to member function (that is, `void (tnode::*)(char*,tnote*,tnode*)`, see 5.3.1).

50) See, for example, `<cstring>` (21.2).

4 A nonstatic member function may be declared `const`, `volatile`, or `const volatile`. These *cv-qualifiers* affect the type of the `this` pointer, see 9.4.2. They also affect the type of the member function; a member function declared `const` is a *const* member function, a member function declared `volatile` is a *volatile* member function and a member function declared `const volatile` is a *const volatile* member function. For example,

```
struct X {
        void g() const;
        void h() const volatile;
};
```

`X::g` is a `const` member function and `X::h` is a `const volatile` member function.

5 A nonstatic member function may be declared *virtual* (10.3) or *pure virtual* (10.4).

9.4.2 The `this` pointer [class.this]

1 In the body of a nonstatic (9.4) member function, the keyword `this` is a non-lvalue expression whose value is the address of the object for which the function is called. The type of `this` in a member function of a class `X` is `X*`. If the member function is declared `const`, the type of `this` is `const X*`, if the member function is declared `volatile`, the type of `this` is `volatile X*`, and if the member function is declared `const volatile`, the type of `this` is `const volatile X*`.

2 In a `const` member function, the object for which the function is called is accessed through a `const` access path; therefore, a `const` member function shall not modify the object and its non-static members. 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`.

3 Similarly, `volatile` semantics (7.1.5.1) apply in `volatile` member functions when accessing the object and its non-static members.

4 A *cv-qualified* member function can be called on an object-expression (5.2.4) only if the object-expression is as qualified or less-qualified than the member function. 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 is, `s::g()` is less-qualified than the object-expression y.

5 Constructors (12.1) and destructors (12.4) cannot be declared `const`, `volatile` or `const volatile`; however, these functions can be invoked to create and destroy objects with cv-qualified types, see 12.1 and 12.4.

9.5 Static members [class.static]

1 A data or function member of a class may be declared `static` in a class definition, in which case it is a *static member* of the class.

2 A `static` member `s` of class `X` may be referred to using the *qualified-id* expression `X::s`; it is not necessary to use the class member access syntax (_class.ref_) to refer to a `static` member. A `static` member may be referred to using the class member access syntax, in which case the *object-expression* is always evaluated. For example,

```
class process {
public:
        static void reschedule();
};
process& g();

void f()
{
        process::reschedule(); // ok: no object necessary
        g().reschedule();      // g() is called
}
```

A `static` member may be referred to directly in the scope of its class; in this case, the `static` member is referred to as if a *qualified-id* expression was used in which the *nested-name-specifier* names the class of the static member. For example,

```
class X {
public:
        static int i;
        static int g();
};
int X::i = g(); // equivalent to X::g();
```

3 The definition of a `static` member function or the *initializer* expression of a `static` data member definition may use the names of the `static` members, enumerators, and nested types of the member's class directly. During name lookup, when an *unqualified-id* (5.1) used in the definition of a static member resolves to a `static` member, enumerator or nested type of its class, the *unqualified-id* is transformed into a *qualified-id* expression in which the *nested-name-specifier* names the class of the `static` member. The definition of a `static` member shall not use directly the names of the nonstatic members of its class (including as operands of the `sizeof` operator). The definition of a `static` member may only refer to the nonstatic members of its class by using the class member access syntax (5.2.4) with an *object-expression* of its class type.

4 Static members obey the usual class member access rules (11).

5 The type of a `static` member does not involve its class name; thus, in the example above, the type of the *qualified-id* expression `X::g` is a function type and the type of `&X::g` is pointer to function type (that is, `void(*)()`, see 5.3.1).

9.5.1 Static member functions [class.static.mfct]

1 The rules described in 9.4 apply to `static` member functions.

2 A `static` member function does not have a `this` pointer (9.4.2). A `static` member function shall not be `virtual`. There shall not be a `static` and a nonstatic member function with the same name and the same parameter types (13.1). A nonstatic member function shall not be declared `const`, or`volatile`, `const volatile`.

9.5.2 Static data members [class.static.data]

1 A `static` data member is not part of the subobjects of a class. There is only one copy of a `static` data member shared by all the objects of the class.

2 The declaration of a `static` data member in its class definition is not a definition and may be of an incomplete type. A definition shall be provided for the `static` data member in a namespace scope enclosing the member's class definition. In the definition at namespace scope, the name of the `static` data member shall be qualified by its class name using the `::` operator. The *initializer* expression in the definition of a `static` data member is in the scope of its class (9.3). For example,

```
class process {
        static process* run_chain;
        static process* running;
};

process* process::running = get_main();
process* process::run_chain = running;
```

The `static` data member `run_chain` of class `process` is defined in global scope; the notation `process::run_chain` specifies that the member `run_chain` is a member of class `process` and in the scope of class `process`. In the `static` data member definition, the *initializer* expression refers to the `static` data member `running` of class `process`.

3 Once the `static` data member has been defined, it exists even if no objects of its class have i been created. For example, in the example above, `run_chain` and `running` exist even if no objects of class `process` are been created by the program.

4 If a `static` data member is of integral or enumeration type, its declaration in the class definition may specify a *constant-initializer*. In that case, the member can appear in integral constant expressions (5.19) within its declarative region after its declaration. The member shall still be defined in a namespace scope and the definition of the member in namespace scope shall not contain an *initializer*.

5 There shall be exactly one definition of a `static` data member in a program; no diagnostic is required; see 3.2.

6 `Static` data members of a class in namespace scope have external linkage (3.5). A local class cannot have `static` data members.

7 `Static` data members are initialized and destroyed exactly like global objects; see 3.6.2 and 3.6.3.

8 A `static` data member cannot be `mutable`(7.1.1).

9.6 Unions [class.union]

1 A union can be thought of as a class whose member objects all begin at offset zero and whose size is sufficient to contain any of its member objects. At most one of the member objects can be stored in a union at any time. A union can have member functions (including constructors and destructors), but not virtual (10.3) functions. A union shall not have base classes. A union shall not be used as a base class. An object of a class with a non-trivial constructor (12.1) or a non-trivial destructor (12.4) or with a user-defined copy assignment operator (13.4.3) cannot be a member of a union. A union can have no `static` data members.

> **Box 47**
> 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 shall be distinct from other names in the scope in which the union is declared; they are used directly in that scope without the usual member access syntax (5.2.4). 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 shall be declared `static`. An anonymous union shall not have `private` or `protected` members (11). An anonymous union shall not have function members.

4 A union for which objects or pointers are declared is not an anonymous union. 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 with no user-declared constructors is described in 8.5.1.

9.7 Bit-fields [class.bit]

1 A *member-declarator* of the form

identifier$_{opt}$ **:** *constant-expression*

specifies a bit-field; its length is set off from the bit-field name by a colon. Allocation of bit-fields within a class object is implementation 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 shall not be a static member. A bit-field shall have integral or enumeration type (3.8.1). It is implementation dependent whether a plain (neither explicitly signed nor unsigned) `int` field is signed or unsigned. The address-of operator `&` shall not be applied to a bit-field, so there are no pointers to bit-fields. Nor are there references to bit-fields.

9.8 Nested class declarations [class.nest]

1 A class can be defined within another class. A class defined within another is called a *nested* class. The name of a nested class is local to its enclosing class. The nested class is in the scope of its enclosing class. Except by using explicit pointers, references, and object names, declarations in a nested class can use only type names, static members, and enumerators from the enclosing class.

```
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 a namespace scope containing the definition of their class. For example,

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

int enclose::inner::x = 1;

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

A nested class `Y` may be declared in a class `X` and later defined in the definition of class `X` or be later defined in a namespace scope containing the definition of class `X`. 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 (11.4) defined within a nested class is in the lexical scope of that class; it obeys the same rules for name binding as a static member function of that class (described in 9.5) and has no special access rights to members of an enclosing class.

9.9 Local class declarations [class.local]

1 A class can be defined within a function definition; such a class is called a *local* class. The name of a local class is local to its enclosing scope. The local class is in the scope of the enclosing scope. Declarations in a local class can use only type names, static variables, `extern` variables and functions, and enumerators from the enclosing scope. 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 shall be defined within their class definition. A local class shall not have static data members.

9.10 Nested type names [class.nested.type]

1 Type names obey exactly the same scope rules as other names. In particular, type names defined within a class definition cannot be used outside their class without qualification. 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 can 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* shall 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) can be used to refer to a base member explicitly. This allows access to a name that has been redefined in the derived class. A derived class can itself serve as a base class subject to access control; see 11.2. A pointer to a derived class can be implicitly converted to a pointer to an accessible unambiguous base class (4.10). An lvalue of a derived class type can be bound to a reference to an accessible unambiguous base class (8.5.3).

2 For example,

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

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

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

3 Here, an object of class `Derived2` will have a sub-object of class `Derived` which in turn will have a sub-object of class `Base`. A derived class and its base class sub-objects can be represented by a directed acyclic graph (DAG) where an arrow means "directly derived from." A DAG of sub-objects is often referred to as a "sub-object lattice." For example,

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

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

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

10.1 Multiple base classes [class.mi]

1 A class can 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 initialization by constructor (12.6.2), for cleanup (12.4), and for storage layout (5.4, 9.2, 11.1).

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

```
class B { /* ... */ };
class D : public B, public B { /* ... */ };  // ill-formed

class L { public: int next;  /* ... */ };
class A : public L { /* ... */ };
class B : public L { /* ... */ };
class C : public A, public B { void f(); /* ... */ };   // well-formed
```

For an object of class `C`, each distinct occurrence of a (non-virtual) base class `L` in the class lattice of `C` corresponds one-to-one with a distinct `L` subobject within the object of type `C`. Given the class `C` defined above, an object of class `C` will have two sub-objects of class `L` as shown below.

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

In such lattices, explicit qualification can be used to specify which subobject is meant. For example, the body of function `C::f` could refer to a member `next` of each `l` subobject:

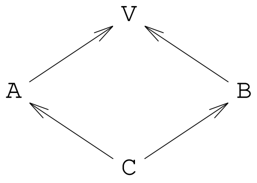
```
void C::f() { A::next = B::next; }   // well-formed
```

Without the `A::` or `B::` qualifiers, the definition of `C::f` above would be ill-formed because of ambiguity.

4 The keyword `virtual` can 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 { /* ... */ };
```

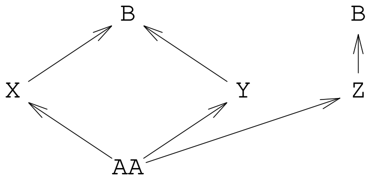
Here class `C` has only one sub-object of class `V`, as shown below.



5 A class can have both virtual and nonvirtual base classes of a given type.

```
class B { /* ... */ };
class X : virtual public B { /* ... */ };
class Y : virtual public B { /* ... */ };
class Z : public B { /* ... */ };
class AA : public X, public Y, public Z { /* ... */ };
```

For an object of class `AA`, all `virtual` occurrences of base class `B` in the class lattice of `AA` correspond to a single `B` subobject within the object of type `AA`, and every other occurrence of a (non-virtual) base class `B` in the class lattice of `AA` corresponds one-to-one with a distinct `B` subject within the object of type `AA`. Given the class `AA` defined above, class `AA` has two sub-objects of class `B`: `Z`'s `B` and the virtual `B` shared by `X` and `Y`, as shown below.



10.2 Member Name Lookup [class.member.lookup]

1 Member name lookup determines the meaning of a name (*id-expression*) in a class scope. Name lookup can result in an *ambiguity*, in which case the program is ill-formed. For an *id-expression*, name lookup begins in the class scope of `this`; for a *qualified-id*, name lookup begins in the scope of the *nested-name-specifier*. Name lookup takes place before access control (11).

2 The following steps define the result of name lookup in a class scope. First, we consider every declaration for the name in the class and in each of its base class sub-objects. A member name `f` in one sub-object `B` *hides* a member name `f` in a sub-object `A` if `A` is a base class sub-object of `B`. We eliminate from consideration any declarations that are so hidden. If the resulting set of declarations are not all from sub-objects of the same type, or the set has a nonstatic member and includes members from distinct sub-objects, there is an ambiguity and the program is ill-formed. Otherwise that set is the result of the lookup.

3 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
}
```

4 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(); }
};
```

5 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 {};
```

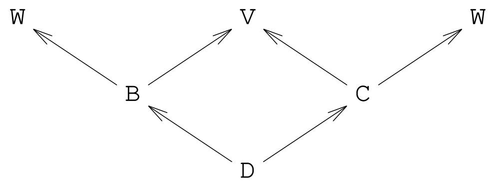
```
class D : public B, public C { };

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'
}
```

6 When virtual base classes are used, a hidden declaration can be reached along a path through the sub-object lattice that does not pass through the hiding declaration. This is not an ambiguity. The identical use with nonvirtual base classes is an ambiguity; in that case there is no unique instance of the name that hides all the others. 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 glorp(); };
```



The names defined in `V` and the left hand instance of `W` are hidden by those in `B`, but the names defined in the right hand instance of `W` are not hidden at all.

```
void D::glorp()
{
    x++;        // ok: B::x hides V::x
    f();        // ok: B::f() hides V::f()
    y++;        // error: B::y and C's W::y
    g();        // error: B::g() and C's W::g()
}
```

7 An explicit or implicit conversion from a pointer to or an lvalue of a derived class to a pointer or reference to one of its base classes shall unambiguously refer to a unique object representing the base class. 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*[51] `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 virtual member function does not have to be visible to be overridden, for example,

```
struct B {
        virtual void f();
};
struct D : B {
        void f(int);
};
struct D2 : D {
        void f();
};
```

the function `f(int)` in class `D` hides the virtual function `f()` in its base class `B`; `D::f(int)` is not a virtual function. However, `f()` declared in class `D2` has the same name and the same parameter list as `B::f()`, and therefore is a virtual function that overrides the function `B::f()` even though `B::f()` is not visible in class `D2`.

4 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();
};
```

[51] 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.

```
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
}
```

5 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.

6 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 shall be defined or declared pure (10.4) in that class.

7 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`.

8 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.

9 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
}
```

10 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 can be created except as sub-objects of a class derived from it. A class is abstract if it has at least one *pure virtual function* (which might be inherited: see below). A virtual function is specified *pure* by using a *pure-specifier* (9.2) in the function declaration in the class declaration. A pure virtual function need be defined only if explicitly called with the *qualified-id* syntax (5.1). 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 shall 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 can 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(); // a definition is required somewhere
};
```

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

4 An abstract class can be derived from a class that is not abstract, and a pure virtual function may override a virtual function which is not pure.

5 Member functions can be called from a constructor 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. *

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
};
```

3 Access control is applied uniformly to all names.

4 It should be noted that it is *access* to members and base classes that is controlled, not their *visibility*. Names of members are still visible, and implicit conversions to base classes are still considered, when those members and base classes are inaccessible. The interpretation of a given construct is established without regard to access control. If the interpretation established makes use of inaccessible member names or base classes, the construct is ill-formed.

5 All access controls in this clause affect the ability of an entire function or member function to access a class member. In particular, access controls apply as usual to members accessed as part of a function return type, even though it is not possible to determine the access privileges of that use without first parsing the rest of the function. For example:

```
class A {
    typedef int I;      // private member
    I f();
    friend I g(I);
    static I x;
};

A::I A::f() { return 0; }
A::I g(A::I);
A::I g(A::I p) { return 0; }
A::I A::x = 0;
```

Here, all the uses of A::I are well-formed because `A::f` and `A::x` are members of class `A` and `g` is a friend of class `A`. This implies, for example, that access checking on the first use of `A::I` must be deferred until it is determined that this use of `A::I` is as the return type of a member of class `A`.

11.1 Access specifiers [class.access.spec]

1 Member declarations can be labeled by an *access-specifier* (10):

access-specifier **:** *member-specification*$_{opt}$

An *access-specifier* specifies the access rules for members following it until the end of the class or until another *access-specifier* is encountered. 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. 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[52].

2 In the absence of an *access-specifier* for a base class, `public` is assumed when the derived class is declared `struct` and `private` is assumed when the class is declared `class`. For example,

```
class B { /* ... */ };
class D1 : private B { /* ... */ };
class D2 : public B { /* ... */ };
class D3 : B { /* ... */ };    // 'B' private by default
struct D4 : public B { /* ... */ };
struct D5 : private B { /* ... */ };
struct D6 : B { /* ... */ };   // 'B' public by default
class D7 : protected B { /* ... */ };
struct D8 : protected B { /* ... */ };
```

Here `B` is a public base of `D2`, `D4`, and `D6`, a private base of `D1`, `D3`, and `D5`, and a protected base of `D7` and `D8`. *

3 Because of the rules on pointer conversion (4.10), a static member of a private base class might be inaccessible as an inherited name, but accessible directly. For example,

[52] 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.

```
class B {
public:
        int mi;          // nonstatic member
        static int si;   // static member
};
class D : private B {
};
class DD : public D {
        void f();
};

void DD::f() {
        mi = 3;         // error: mi is private in D
        si = 3;         // error: si is private in D
        B  b;
        b.mi = 3;       // okay (b.mi is different from this->mi)
        b.si = 3;       // okay (b.si is different from this->si)
        B::si = 3;      // okay
        B* bp1 = this;  // error: B is a private base class
        B* bp2 = (B*)this;  // okay with cast
        bp2->mi = 3;    // okay: access through a pointer to B.
}
```

4 A base class is said to be accessible if an invented public member of the base class is accessible. If a base class is accessible, one can implicitly convert a pointer to a derived class to a pointer to that base class (4.10, 4.11). It follows that members and friends of a class `X` can implicitly convert an `X*` to a pointer to a private or protected immediate base class of `X`.

11.3 Access declarations [class.access.dcl]

1 The access of a member of a base class can be changed in the derived class by mentioning its *qualified-id* in the derived class declaration. Such mention is called an *access declaration*. The base class member is given, in the derived class, the access in effect in the derived class declaration at the point of the access declaration. The effect of an access declaration *qualified-id* `;` is defined to be equivalent to the declaration `using` *qualified-id* `;`.[53)]

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;
};
```

[53)] Access declarations are deprecated; member *using-declarations* (7.3.3) provide a better means of doing the same things. In earlier versions of the C++ language, access declarations were more limited; they were generalized and made equivalent to `using` declarations in the interest of simplicity. Programmers are encouraged to use `using`, rather than the new capabilities of access declarations, in new code.

```
class D : private B {
    int d;
public:
    B::c;  // adjust access to 'B::c'
    B::z;  // adjust access to 'A::z'
    A::z1; // adjust access to 'A::z1'
    int e;
    int df();
protected:
    B::x;  // adjust access to 'B::x'
    int g;
};

class X : public D {
    int xf();
};

int ef(D&);
int ff(X&);
```

The external function `ef` can use only the names `c`, `z`, `z1`, `e`, and `df`. Being a member of `D`, the function `df` can use the names `b`, `c`, `z`, `z1`, `bf`, `x`, `y`, `d`, `e`, `df`, and `g`, but not `a`. Being a member of `B`, the function `bf` can use the members `a`, `b`, `c`, `z`, `z1`, `bf`, `x`, and `y`. The function `xf` can use the public and protected names from `D`, that is, `c`, `z`, `z1`, `e`, and `df` (public), and `x`, and `g` (protected). Thus the external function `ff` has access only to `c`, `z`, `z1`, `e`, and `df`. If `D` were a protected or private base class of `X`, `xf` would have the same privileges as before, but `ff` would have no access at all. *

11.4 Friends [class.friend]

1 A friend of a class is a function that is not a member of the class but is permitted to use the private and protected member names from the class. The name of a friend is not in the scope of the class, and the friend is not called with the member access operators (5.2.4) unless it is a member of another class. 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[54] (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
};
```

3 A function declared as a `friend` and not previously declared, is introduced in the smallest enclosing non-class, non-function prototype scope that contains the `friend` declaration. For a class mentioned as a `friend` and not previously declared, see 7.1.5.3.

4 A function first declared in a friend declaration has external linkage (3.5). Otherwise, it retains its previous linkage (7.1.1). No *storage-class-specifier* shall appear in the *decl-specifier-seq* of a friend declaration.

5 A function of namespace scope can be defined in a `friend` declaration of a non-local class (9.9). The function is then `inline`. A `friend` function defined in a class is in the (lexical) scope of the class in which it is defined. A friend function defined outside the class is not.

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
    }
};
```

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

```
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, type or enumerator constant of a base class; if the access is through a *qualified-id*, the *nested-name-specifier* must name the derived class (or any class derived from that class).

2 A friend or a member function of a derived class can access a protected nonstatic member of a base class. Except when forming a pointer to member, the access must be through a pointer to, reference to, or object of the derived class itself (or any class derived from that class). If the nonstatic protected member thus accessed is also qualified, the qualification is ignored for the purpose of this access checking. If the access is to form a pointer to member (5.3.1), the *nested-name-specifier* shall name the derived class (or any class derived from that class). For example,

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

class D1 : public B {
};

class D2 : public B {
    friend void fr(B*,D1*,D2*);
    void mem(B*,D1*);
};

void fr(B* pb, D1* p1, D2* p2)
{
    pb->i = 1;  // illegal
    p1->i = 2;  // illegal
    p2->i = 3;  // ok (access through a D2)
    p2->B::i = 4;  // ok (access through a D2, qualification ignored)
    int B::* pmi_B = &B::i;    // illegal
    int B::* pmi_B = &D2::i;   // ok (type of &D2::i is "int B::*")
    B::j = 5;   // illegal
    D2::j =6;   // ok (access through a D2)
}

void D2::mem(B* pb, D1* p1)
{
    pb->i = 1;  // illegal
    p1->i = 2;  // illegal
    i = 3;      // ok (access through ‘this’)
    B::i = 4;   // ok (access through ‘this’, qualification ignored)
    j = 5;      // ok (static member accessed by derived class function)
    B::j = 6;   // illegal
}
```

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

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 int f();
};

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

void f()
{
    D d;
    B* pb = &d;
    D* pd = &d;

    pb->f();  // ok: B::f() is public,
              // D::f() is invoked
    pd->f();  // error: D::f() is private
}
```

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 can be converted to values of other types. Often such special functions are called implicitly. Also, the compiler can generate instances of these functions when the programmer does not supply them. Compiler-generated special functions can 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 initialize objects of its class type. For initialization of objects of class type see 12.6.

2 A constructor can be invoked for a `const`, `volatile` or `const volatile` object.[55] A constructor shall not be declared `const`, `volatile`, or `const volatile` (9.4.2). A constructor shall not be `virtual` or `static`.

3 Constructors are not inherited.

4 A *default constructor* for a class `X` is a constructor of class `X` that can be called without an argument. If there is no *user-declared constructor* for class `X`, a default constructor is implicitly declared. An *implicitly-declared default constructor* is a `public` member of its class. A constructor is `trivial` if it is an implicitly-declared default constructor and if:

— its class has no virtual functions (10.3) and no virtual base classes (10.1), and

— all the direct base classes of its class have trivial constructors, and

— for all the nonstatic data members of its class that are of class type (or array thereof), each such class has a trivial constructor.

5 Otherwise, the constructor is *non-trivial*.

6 An implicitly-declared default constructor for a class is *implicitly-defined* when it is used to create an object of its class type (3.7). A program is ill-formed if the class for which a default constructor is implicitly defined has:

— a nonstatic data member of `const` type, or

— a nonstatic data member of reference type, or

— a nonstatic data member of class type (or array thereof) with an inaccessible default constructor, or

— a base class with an inaccessible default constructor.[56]

55) Volatile semantics might or might not be used.

56) When a default constructor for a derived class is implicitly defined, all the implicitly-declared default constructors for its bases and members are also implicitly defined (and this recursively for the members' base classes and members).

Box 48

Should it be specified more precisely at which point in the program the implicit definition is ill-formed? i.e. is something like this needed: "The declaration or expression causing the implicit definition is ill-formed" ?

7 A *copy constructor* for a class `X` is a constructor that accepts one parameter of type `X&` or of type `const X&`. See 12.8 for more information on copy constructors.

8 12.6.2 describes the order in which constructors for base classes and non-static members are called and describes how arguments can be specified for the calls to these constructors.

9 A union member cannot be of a class type (or array thereof) that has a non-trivial constructor.

10 No return type (not even `void`) can be specified for a constructor. A `return` statement in the body of a constructor shall 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. 12.2 describes the lifetime of temporary objects.

12 Some language constructs have special semantics when used during construction; see 12.6.2 and 12.7.

12.2 Temporary objects [class.temporary]

1 In some circumstances it might be necessary or convenient for the compiler to generate a temporary object. Precisely when such temporaries are introduced is implementation dependent. 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, an implementation might use a temporary in which to construct `X(2)` before passing it to `f()` using `X`'s copy-constructor; alternatively, `X(2)` might be constructed in the space used to hold the argument. Also, a temporary might be used to hold the result of `f(X(2))` before copying it to `b` using `X`'s copy-constructor; alternatively, `f()`'s result might be constructed in `b`. On the other hand, the expression `a=f(a)` requires a temporary for either the argument `a` or the result of `f(a)` to avoid undesired aliasing of `a`. Even if the copy constructor is not called, all the semantic restrictions, such as accessibility, shall be satisfied.

2 When a compiler introduces a temporary object of a class that has a non-trivial constructor (12.1), it shall ensure that a constructor is called for the temporary object. Similarly, the destructor shall be called for a temporary with a non-trivial destructor (12.4). Ordinarily, temporary objects are destroyed as the last step

in evaluating the full-expression (1.8) that (lexically) contains the point where they were created. This is true even if that evaluation ends in throwing an exception.

3 There are two contexts in which temporaries are destroyed at a different point then at the end of the full-expression. The first context is when an expression appears as an initializer for a declarator defining an object. In that context, the temporary that holds the result of the expression shall persist until the object's initialization is complete. The object is initialized from a copy of the temporary; during this copying, an implementation can call the copy constructor many times; the temporary is destroyed as soon as it has been copied.

4 The second context is when a temporary is bound to a reference. The temporary bound to the reference or the temporary containing the sub-object that is bound to the reference persists for the lifetime of the reference initialized or until the end of the scope in which the temporary is created, which ever comes first. A temporary holding the result of an initializer expression for a declarator that declares a reference persists until the end of the scope in which the reference declaration occurs. A temporary bound to a reference in a constructor's ctor-initializer (12.6.2) persists until the constructor exits. A temporary bound to a reference parameter in a function call (5.2.2) persists until the completion of the call. A temporary bound in a function return statement (6.6.3) persists until the function exits.

5 In all cases, temporaries are destroyed in reverse order of creation.

12.3 Conversions [class.conv]

1 Type conversions of class objects can be specified by constructors and by conversion functions.

2 Such conversions, often called *user-defined conversions*, are used implicitly in addition to standard conversions (4). 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 (10.2, 12.3.2). Conversions obey the access control rules (11). As ever access control is applied after ambiguity resolution (3.4).

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 declared without the *function-specifier* `explicit` that can be called with a single parameter specifies a conversion from the type of its first parameter to the type of its class. Such a constructor is called a converting constructor. 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))
}
```

2 A nonconverting constructor constructs objects just like converting constructors, but does so only where a constructor call is explicitly indicated by the syntax.

```
class Z {
public:
        explicit Z(int);
        // ...
};

Z a1 = 1;         // error: no implicit conversion
Z a3 = Z(1);      // ok: explicit use of constructor
Z a2(1);          // ok: explicit use of constructor
Z* p = new Z(1); // ok: explicit use of constructor
```

3 When no converting 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 {
public:
        X(int);
        // ...
};

class Y {
public:
        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 shall not be declared in the *type-specifier-seq*. Neither parameter types nor return type can be specified. A conversion operator is never used to convert a (possibly qualified) object (or reference to an object) to the (possibly qualified) same object type (or a reference to it), or to a (possibly qualified) base class of that type (or a reference to it). 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 objects of type `cl`. A destructor takes no parameters, and no return type can be specified for it (not even `void`). It is not possible to take the address of a destructor. A destructor can be invoked for a `const`, `volatile` or `const volatile` object.[57] A destructor shall not be declared `const`, `volatile` or `const volatile` (9.4.2). A destructor shall not be `static`.

2 If a class has no *user-declared destructor*, a destructor is declared implicitly. An *implicitly-declared destructor* is a `public` member of its class. A destructor is *trivial* if it is an implicitly-declared destructor and if:

— all of the direct base classes of its class have trivial destructors and

— for all of the non-static data members of its class that are of class type (or array thereof), each such class has a trivial destructor.

3 Otherwise, the destructor is `non-trivial`.

4 An implicitly-declared destructor is *implicitly-defined* when it is used to destroy an object of its class type (3.7). A program is ill-formed if the class for which a destructor is implicitly defined has:

— a non-static data member of class type (or array thereof) with an inaccessible destructor, or

— a base class with an inaccessible destructor.[58]

> **Box 49**
> Should it be specified more precisely at which point in the program the implicit definition is ill-formed?

5 Bases and members are destroyed in reverse of their construction (see 12.6.1). Destructors for elements of an array are called in reverse order of their construction. *

6 Destructors are not inherited. A destructor can be declared `virtual` (10.3) or pure `virtual` (10.4); if any objects of that class or any derived class are created in the program, the destructor shall be defined. If a class has a base class with a virtual destructor, its destructor (whether user- or implicitly-declared) is virtual.

7 Some language constructs have special semantics when used during destruction; see 12.7.

8 A union member cannot be of a class type (or array thereof) that requires a non-trivial destructor.

[57] Volatile semantics might or might not be used.
[58] When a destructor for a derived class is implicitly defined, all the implicitly-declared destructors for its bases and members are also implicitly defined (and this recursively for the members' base classes and members).

9 Destructors are invoked implicitly (1) when an automatic variable (3.7) or temporary (12.2, 8.5.3) object goes out of scope, (2) for constructed static (3.7) objects at program termination (3.6), and (3) through use of a *delete-expression* (5.3.5) for objects allocated by a *new-expression* (5.3.4). Destructors can also be invoked explicitly. A *delete-expression* invokes the destructor for the referenced object and passes the address of its memory to a deallocation function (5.3.5, 12.5). 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
}
```

10 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
}
```

11 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.

12 The notation for explicit call of a destructor can 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.

13 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 (3.7.3.1).

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.[59)]

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

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

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

[59)] 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.

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. Hence, even though a different one might actually be executed, the statically visible deallocation function is required to 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]

> **Box 50**
> This needs to be improved to talk about the behavior of all initializations; operator new cannot use an initializer-clause; temporary creation only uses default constructors.

1 When no explicit initialization is specified when creating a class object, if the class has a default constructor (12.1), the default constructor is used to initialize the object. If no default constructor exists for the class and the class has a non-trivial constructor (12.1), the object shall be explicitly initialized. If the class is an aggregate (8.5.1), an *initializer-clause* can be used; otherwise, a call to a user-declared constructor shall be specified.

2 Arrays of objects of class type use constructors in initialization (12.1) just as do individual objects. If the array is not explicitly initialized and the class has a default constructor, implicit initialization of the array elements occurs by calling the default constructor for each element of the array, in order of increasing addresses (8.3.4). If no default constructor exists for the class and the class has a non-trivial constructor, the array shall be explicitly initialized.

12.6.1 Explicit initialization [class.expl.init]

1 Objects of classes with user-declared 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 for declarations, a single value is specified as the initializer using the = operator. This value is used as the argument to a copy constructor (12.1, 12.8). Typically, that call of a copy constructor can be eliminated (12.2). 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'
complex g = { 1, 2 };      // error; constructor is required
```

Overloading of the assignment operator (13.4.3) = has no effect on initialization. See 8.5 for the distinction between the parenthesized and = forms of initialization.

2 If an array of class objects is initialized with an *initializer-clause* (8.5.1), each *assignment-expression* is treated as an argument in a constructor call to initialize one element of the array, using the = form of initialization (8.5). If there are fewer *assignment-expression*s in the *initializer-clause* than elements in the array, the remaining elements are initialized using the default constructor for the class. If there is no default constructor and the *initializer-clause* is incomplete, the array declaration is ill-formed. For example,

```
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()`.

3 The order in which static objects are initialized is described in 3.6.2 and 6.7.

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 nonstatic 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);
```

2 If class `X` has a member `m` of class type `M` and `M` has no default constructor, then a definition of a constructor for class `X` is ill-formed if it does not specify a *mem-initializer* for `m`.

3

Box 51
It needs to be made clear that the order specified below applies for user-declared constructors as well as for implicitly-declared constructors.

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.

4 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.

5 The class of a *complete object* (1.6) 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 shall 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()
```

6 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`.

7 The identifier of a *ctor-initializer*'s *mem-initializer* in a class' constructor is looked up in the scope of the class. It shall 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 can differ from the class definition. For example:

```
struct A { A(); };
typedef A global_A;
struct B { };
struct C: public A, public B { C(); };
C::C(): global_A() { }    // calls A()
```

A base class type in a *ctor-initializer*'s *mem-initializer* 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?
```

8 Member functions (including virtual member functions, 10.3) can be called for an object under construction. Similarly, an object under construction can be the operand of the `typeid` operator (5.2.7) or of a `dynamic_cast` (5.2.6). However, if these operations are performed in a *ctor-intializer* (or in a function called directly or indirectly from a *ctor-intializer*) before all the *mem-initializer*s for base classes have completed, the result of the operation is undefined. For example:

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

```
class B : public A {
        int j;
public:
        int f();
B() : A(f()),           // undefined: calls member function
                        // but base A not yet initialized
        j(f()) { }      // well-defined: bases are all initialized
};

class C {
public:
        C(int);
};

class D : public B, C {
        int i;
public:
        D() : C(f()),   // undefined: calls member function
                        // but base C not yet initialized
        i(f()) {}       // well-defined: bases are all initialized
};
```

9 12.7 describes the result of virtual function calls, `typeid` and `dynamic_casts` during construction for the well-defined cases; that is, describes the *polymorphic behavior* of an object under construction.

12.7 Construction and destruction | [class.cdtor]

1 For an object of non-POD class type (9), before the constructor begins execution and after the destructor finishes execution, referring to any nonstatic member or base class of the object results in undefined behavior. For example,

```
struct X { int i; };
struct Y : X { };
struct A { int a; };
struct B : public A { int j; Y y; };

extern B bobj;
B* pb = &bobj;          // ok
int* p1 = &bobj.a;      // undefined, refers to base class member
int* p2 = &bobj.y.i;    // undefined, refers to member's member

A* pa = &bobj;          // undefined, upcast to a base class type
B bobj;                 // definition of bobj

extern X xobj;
int* p3 = &xobj.i;      // Ok, X is a POD class
X xobj;
```

2 Example

```
struct W { int j; };
struct X : public virtual W { };
struct Y {
        int *p;
        X x;
        Y() : p(&x.j)    // undefined, x is not yet constructed
        { }
};
```

3 To explicitly or implicitly convert a pointer to an object of class X to a pointer to a direct or indirect base class B, the construction of X and the construction of all of its direct or indirect bases that directly or indirectly derive from B and which are also direct or indirect base classes of X[60] shall have started and the destruction of these classes shall not have completed, otherwise the computation results in undefined behavior. To form a pointer to a direct nonstatic member of an object X given a pointer to X, the construction of X shall have started and the destruction of X shall not have completed, otherwise the computation results in undefined behavior. For example,

```
struct A { };
struct B : virtual A { };
struct C : B { };
struct D : virtual A { D(A*); };
struct X { X(A*); };

struct E : C, D, X {
        E() : D(this),  // undefined: upcast from E* to A*
                        // might use path E* -> D* -> A*
                        // but D is not constructed
        // D((C*)this), // defined:
                        // E* -> C* defined because E() has started
                        // and C* -> A* defined because
                        // C fully constructed
        X(this)         // defined: upon construction of X,
                        // C/B/D/A sublattice is fully constructed
        { }
};
```

4 Member functions, including virtual functions (10.3), can be called during construction or destruction (12.6.2). When a virtual function is called directly or indirectly from a constructor (including from its *ctor-initializer*) or from a destructor, the function called is the one defined in the constructor or destructor's own class or in one of its bases, but not a function overriding it in a class derived from the constructor or destructor's class or overriding it in one of the other base classes of the complete object (1.6). If the virtual function call uses an explicit class member access (5.2.4) and the object-expression's type is neither the constructor or destructor's own class or one of its bases, the result of the call is undefined. For example,

```
class V {
public:
        virtual void f();
        virtual void g();
};

class A : public virtual V {
public:
        virtual void f();
};

class B : public virtual V {
public:
        virtual void g();
        B(V*, A*);
};
```

[60] If X is itself a base class, not all classes derived from B are necessarily base classes of X.

```
class D : public A, B {
public:
        virtual void f();
        virtual void g();
        D() : B((A*)this, this) { }
};

B::B(V* v, A* a) {
        f();    // calls V::f, not A::f
        g();    // calls B::g, not D::g
        v->g(); // v is base of B, the call is well-defined, calls B::g
        a->f(); // undefined behavior, a's type not a base of B
}
```

5 The `typeid` operator (5.2.7) can be used during construction or destruction (12.6.2). When `typeid` is used in a constructor (including in its *ctor-initializer*) or in a destructor, or used in a function called (directly or indirectly) from a constructor or destructor, if the operand of `typeid` refers to the object under construction or destruction, `typeid` yields the type_info representing the constructor or destructor's class. If the operand of `typeid` refers to the object under construction or destruction and the static type of the operand is neither the constructor or destructor's class nor one of its bases, the result of `typeid` is undefined.

6 `Dynamic_casts` (5.2.6) can be used during construction or destruction (12.6.2). When a `dynamic_cast` is used in a constructor (including in its *ctor-initializer*) or in a destructor, or used in a function called (directly or indirectly) from a constructor or destructor, if the operand of the `dynamic_cast` refers to the object under construction or destruction, this object is considered to be a complete object that has the type of the constructor or destructor's class. If the operand of the `dynamic_cast` refers to the object under construction or destruction and the static type of the operand is not a pointer to or object of the constructor or destructor's own class or one of its bases, the `dynamic_cast` results in undefined behavior.

7 Example

```
class V {
public:
        virtual void f();
};

class A : public virtual V { };

class B : public virtual V {
public:
        B(V*, A*);
};

class D : public A, B {
public:
        D() : B((A*)this, this) { }
};
```

```
B::B(V* v, A* a) {
        typeid(this);    // type_info for B
        typeid(*v);      // well-defined: *v has type V, a base of B
                         // yields type_info for B
        typeid(*a);      // undefined behavior: type A not a base of B
        dynamic_cast<B*>(v); // well-defined: v of type V*, V base of B
                         // results in B*
        dynamic_cast<B*>(a); // undefined behavior,
                         // a has type A*, A not a base of B
}
```

12.8 Copying class objects [class.copy]

1 A class object can be copied in two ways, by initialization (12.1, 8.5), including for function argument passing (5.2.2) and for function value return (6.6.3), and by assignment (5.17). Conceptually, these two operations are implemented by a copy constructor (12.1) and copy assignment operator (13.4.3).

2 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 default arguments (8.3.6), so that it can be called with a single argument of type `X`. For example, `X::X(const X&)` and `X::X(X&, int=1)` are copy constructors.

Box 52

Should the parameter of the implicitly-declared copy constructor have type `const volatile X&`? See 94-0193R1/N0580R1.

```
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);
```

3 A constructor for a class `X` whose first and only parameter is of type (optionally cv-qualified) `X` is ill-formed.

4 If there is no *user-declared copy constructor*, a copy constructor is implicitly declared [61].

5 If all bases and members of a class `X` have copy constructors accepting `const` parameters, the implicitly-declared copy constructor for `X` has a single parameter of type `const X&`, as follows:

```
X::X(const X&)
```

Otherwise it has a single parameter of type `X&`[62]:

```
X::X(X&)
```

[61] Thus the class definition

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

causes a copy constructor to be implicitly-declared and the member function definition

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

is ill-formed because of ambiguity.

[62] In this case, programs that attempt initialization by copying of `const X` objects are ill-formed.

6 An implicitly-declared copy constructor is a `public` member of its class. Copy constructors are not inherited.

7 An implicitly-declared copy constructor is *implicitly defined* when it is used to copy an object of its class type.

Box 53

We need to refer to subclauses that describe when class copy takes place. Is the concept of trivial copy constructor needed?

A program is ill-formed if the class for which a copy constructor is implicitly defined has:

— a nonstatic data member of class type (or array thereof) with an inaccessible copy constructor, or

— a base class with an inaccessible copy constructor. [63)]

Box 54

Should it be specified more precisely at which point in the program the implicit definition is ill-formed? i.e. is something like this needed: "The first declaration or expression that does a class copy causing the implicitly-declared copy constructor to be implicitly-defined is ill-formed" ?

8 The semantics of the implicitly-declared copy constructor are that of *memberwise initialization* of the base classes and nonstatic data members; memberwise initialization implies that if a class `X` has a member (or array thereof) or base of a class `M`, `M`'s copy constructor is used by `X`'s implicitly-declared copy constructor for the initialization of the member or base `M`. Objects representing virtual base classes will be initialized only once by the implicitly-declared copy constructor. See 12.6.1 for the order of initialization of members and bases.

9 A *copy assignment operator* `operator=` is a non-static member function of class `X` with exactly one parameter of type `X&`or `const X&`. If there is no *user-declared copy assignment operator*, a copy assignment operator is implicitly declared for class `X`. If all bases and members of a class `X` have a copy assignment operators accepting `const` parameters, the implicitly-declared copy assignment operator for `X` will have a single parameter of type `const X&`, as follows:

```
X& X::operator=(const X&)
```

Box 55

Should the parameter of the implicitly-declared copy assignment operator have type `const volatile X&`? See 94-0193R1/N0580R1.

Otherwise it will have a single parameter of type `X&`[64)]:

```
X& X::operator=(X&)
```

The implicitly-declared copy assignment operator for class `X` has the return type `X&`; it returns the object for which the assignment operator is invoked, that is, the object assigned to [65)].

63) When a copy constructor for a derived class is implicitly defined, all the implicitly-declared copy constructors for the bases and members are also implicitly defined (and this recursively for the members' base classes and members).

64) In this case, programs that attempt assignment by copying of const `X` objects will be ill-formed.

65) Given the parameter type for the copy assignment operator, objects of a derived class type can be assigned to objects of an accessible base class type. For example,

```
class X {
public:
        int b;
};
class Y : public X {
public:
        int c;
```

10 An implicitly-declared copy assignment operator is a `public` of its class. Copy assignment operators are not inherited.

11 An implicitly-declared copy assignment operator is *implicitly defined* when an object of its class type is copied.

Box 56

We need to refer to subclauses that describe when class copy takes place. Is the concept of trivial copy assignment operator needed?

A program is ill-formed if the class for which a copy assignment operator is implicitly defined has:

— a nonstatic data member of `const` type, or

— a nonstatic data member of reference type, or

— a nonstatic data member of class type (or array thereof) with an inaccessible copy assignment operator, or

— a base class with an inaccessible copy assignment operator [66)]

Box 57

Should it be specified more precisely at which point in the program the implicit definition is ill-formed? i.e. is something like this needed: "The first expression that does a class assignment causing the implicitly-declared copy assignment operator to be implicitly-defined is ill-formed" ?

12 The semantics of the implicitly-declared copy assignment operator are that of memberwise assignment of the base classes and nonstatic data members; memberwise assignment implies that if a class `X` has a member (or array thereof) or base of a class `M`, `M`'s copy assignment operator is used by `X'` s implicitly-declared copy assignment operator for the assignment of the member or base `M`. Objects representing virtual base classes will be assigned only once by a the implicitly-declared copy assignment operator [67)].

```
};

void f()
{
        X x1;
        Y y1;
        x1 = y1;    //1: ok
        y1 = x1;    // error
}
```

On line //1, `y1.b` is assigned to `x1.b` and `y1.c` is not copied.

66) When a copy assignment operator for a derived class is implicitly defined, all the implicitly-declared copy assignment operators for the bases and members are also implicitly defined (and this recursively for the members' base classes and members).

67) Copying one object into another using the copy constructor or the copy assignment operator does not change the layout or size 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)
```

Box 58

This needs more work. See 94-0193R1/N0580R1.

```
            b = a;        // error
            a.f();        // calls s::f
            b.f();        // calls ss::f
            (s&)b = a;    // assign to b's s part
                          // really ((s&)b).s::operator=(a)
            b.f();        // still calls ss::f
        }
```

The call `a.f()` will invoke `s::f()` (as is suitable for an object of class `s` (10.3)) and the call `b.f()` will call `ss::f()` (as is suitable for an object of class `ss`).

13 Overloading [over]

1 When two or more different declarations are specified for a single name in the same scope, that name is said to be *overloaded*. By extension, two declarations in the same scope that declare the same name but with different types are called *overloaded declarations*. Only function declarations can be overloaded; object and type declarations cannot be overloaded.

2 When an overloaded function name is used, which overloaded function declaration is being referenced is determined by comparing the types of the arguments at the point of use with the types of the parameters in the overloaded declarations that are visible at the point of use. This function selection process is called *overload resolution* and is defined in 13.2. For example,

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

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

13.1 Overloadable declarations [over.load]

1 Not all function declarations can be overloaded. Those that cannot be overloaded are specified here. A program is ill-formed if it contains two such non-overloadable declarations in the same scope.

2 Certain function declarations that cannot be distinguished by overload resolution cannot be overloaded:

— Since for any type "`T`," a parameter of type "`T`" and a parameter of type ''reference to `T`" accept the same set of initializer values, function declarations with parameter types differing only in this respect cannot be overloaded.

> **Box 59**
>
> This restriction is hard to check across translation units. Moreover, ambiguities can be detected just fine at call time. Perhaps we should remove it.

For example,

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

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

It is, however, possible to distinguish between "reference to `const T`," "reference to `volatile T`," and plain "reference to `T`" so function declarations that differ only in this respect can be overloaded. Similarly, it is possible to distinguish between "pointer to `const T`," "pointer to `volatile T`," and plain "reference to `T`" so function declarations that differ only in this respect can be overloaded.

— Function declarations that differ only in the return type cannot be overloaded.

— Member function declarations with the same name and the same parameter types cannot be overloaded if any of them is a `static` member function declaration (9.5). The types of the implicit object parameters constructed for the member functions for the purpose of overload resolution (13.2.1) are not considered when comparing parameter types for enforcement of this rule. In contrast, if there is no `static` member function declaration among a set of member function declarations with the same name and the same parameter types, then these member function declarations can be overloaded if they differ in the type of their implicit object parameter. The following example illustrates this distinction:

```
class X {
    static void f();
    void f();                  // ill-formed
    void f() const;            // ill-formed
    void f() const volatile;   // ill-formed
    void g();
    void g() const;            // Ok: no static g
    void g() const volatile;   // Ok: no static g
};
```

3 Function declarations that have equivalent parameter declarations declare the same function and therefore cannot be overloaded:

— Parameter declarations that differ only in the use of equivalent typedef "types" are equivalent. A `typedef` is not a separate type, but only a synonym for another type (7.1.3). For example,

```
typedef int Int;

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

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

```
enum E { a };

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

— Parameter declarations that differ only in a pointer `*` versus an array `[]` are equivalent. That is, the array declaration is adjusted to become a pointer declaration (8.3.5). Note that only the second and subsequent array dimensions are significant in parameter types (8.3.4).

```
f(char*);
f(char[]);      // same as f(char*);
f(char[7]);     // same as f(char*);
f(char[9]);     // same as f(char*);

g(char(*)[10]);
g(char[5][10]);  // same as g(char(*)[10]);
g(char[7][10]);  // same as g(char(*)[10]);
g(char(*)[20]);  // different from g(char(*)[10]);
```

— Parameter declarations that differ only in the presence or absence of `const` and/or `volatile` are equivalent. That is, the `const` and `volatile` type-specifiers for each parameter type are ignored when determining which function is being declared, defined, or called. For example,

```
typedef const int cInt;

int f (int);
int f (const int);      // redeclaration of f (int);
int f (int) { ... }     // definition of f (int)
int f (cInt) { ... }    // error: redefinition of f (int)
```

Only the `const` and `volatile` type-specifiers at the outermost level of the parameter type specification are ignored in this fashion; `const` and `volatile` type-specifiers buried within a parameter type specification are significant and can be used to distinguish overloaded function declarations. In particular, for any type `T`, "pointer to `T`," "pointer to `const T`," and "pointer to `volatile T`" are considered distinct parameter types, as are "reference to `T`," "reference to `const T`," and "reference to `volatile T`."

— Two parameter declarations that differ only in their default initialization are equivalent. Consider the following example

```
void f (int i, int j);
void f (int i, int j = 99);          // Ok: redeclaration of f (int, int)
void f (int i = 88, int j = 99);     // Ok: redeclaration of f (int, int)
void f ();                           // Ok: overloaded declaration of f

void prog ()
{
    f (1, 2);  // Ok: call f (int, int)
    f (1);     // Ok: call f (int, int)
    f ();      // Error: f (int, int) or f ()?
}
```

13.1.1 Declaration matching [over.dcl]

1 Two function declarations of the same name refer to the same function if they are in the same scope and have equivalent parameter declarations (13.1). A function member of a derived class is *not* in the same scope as a function member of the same name in a base class. For example,

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

class D : public B {
public:
    int f(char*);
};
```

Here `D::f(char*)` hides `B::f(int)` rather than overloading it.

```
void h(D* pd)
{
    pd->f(1);       // error:
                    // D::f(char*) hides B::f(int)
    pd->B::f(1);    // ok
    pd->f("Ben");   // ok, calls D::f
}
```

A locally declared function is not in the same scope as a function in a containing scope.

```
int f(char*);
void g()
{
    extern f(int);
    f("asdf");  // error: f(int) hides f(char*)
                // so there is no f(char*) in this scope
}

void caller ()
{
    void callee (int, int);
    {
        void callee (int);  // hides callee (int, int)
        callee (88, 99);    // error: only callee (int) in scope
    }
)
```

2 Different versions of an overloaded member function can be given different access rules. For example,

```
class buffer {
private:
    char* p;
    int size;

protected:
    buffer(int s, char* store) { size = s; p = store; }
    // ...

public:
    buffer(int s) { p = new char[size = s]; }
    // ...
};
```

13.2 Overload resolution [over.match]

1 Overload resolution is a mechanism for selecting the best function to call given a list of expressions that are to be the arguments of the call and a set of *candidate functions* that can be called based on the context of the call. The selection criteria for the best function are the number of arguments, how well the arguments match the types of the parameters of the candidate function, and certain other properties of the candidate function. The function selected by overload resolution is not guaranteed to be appropriate for the context. Other restrictions, such as the accessibility of the function, can make its use in the calling context ill-formed.

2 Overload resolution selects the function to call in five distinct contexts within the language:

— Invocation of a function named in the function call syntax (5.2.2)

— Invocation of a function call operator, a pointer-to-function conversion function, or a reference-to-function conversion function of a class object named in the function call syntax (13.2.1.1)

— Invocation of the operator referenced in an expression (5)

— Invocation of a constructor during initialization of a class object via a parenthesized expression list (12.6.1)

— Invocation of a user-defined conversion during initialization from an expression (8.5, 8.5.3)

3 Each of these contexts defines the set of candidate functions and the list of arguments in its own unique way. But, once the candidate functions and argument lists have been identified, the selection of the best function is the same in all cases:

— First, a subset of the candidate functions—those that have the proper number of arguments and meet certain other conditions—-is selected to form a set of *viable functions*.

— Then the best viable function is selected based on the implicit conversion sequences (13.2.3.1) needed to match each argument to the corresponding parameter of each viable function.

4 If a best viable function exists and is unique, overload resolution succeeds and produces it as the result. Otherwise overload resolution fails and the invocation is ill-formed.

13.2.1 Candidate functions and argument lists [over.match.funcs]

1 The following subclauses describe the set of candidate functions and the argument list submitted to overload resolution in each of the five contexts in which overload resolution is used. The source transformations and constructions defined in these subclauses are only for the purpose of describing the overload resolution process. An implementation is not required to use such transformations and constructions.

2 The set of candidate functions can contain both member and non-member functions to be resolved against the same argument list. So that argument and parameter lists are comparable within this heterogeneous set, a member function is considered to have an extra parameter, called the *implicit object parameter*, which represents the object for which the member function has been called. For the purposes of overload resolution, both static and non-static member functions have an implicit object parameter, but constructors do not.

3 Similarly, when appropriate, the context can construct an argument list that contains an *implied object argument* to denote the object to be operated on. Since arguments and parameters are associated by position within their respective lists, the convention is that the implicit object parameter, if present, is always the first parameter and the implied object argument, if present, is always the first argument.

4 For non-static member functions, the type of the implicit object parameter is "reference to *cv* `X`" where `X` is the class that defines the member function and *cv* is the cv-qualification on the member function declaration. For example, for a `const` member function of class `X`, the extra parameter is assumed to have type "reference to `const X`". For static member functions, the implicit object parameter is considered to match any object (since if the function is selected, the object is discarded).

5 During overload resolution, the implied object argument is indistinguishable from other arguments. The implicit object parameter, however, retains its identity since conversions on the corresponding argument shall obey these additional rules:

— no temporary object can be introduced to hold the argument for the implicit object parameter

— no user-defined conversions can be applied to achieve a type match with it

— even if the implicit object parameter is not `const`-qualified, an rvalue temporary can be bound to the parameter as long as in all other respects the temporary can be converted to the type of the implicit object parameter.

13.2.1.1 Function call syntax [over.match.call]

1 Recall from 5.2.2, that a *function call* is a *postfix-expression*, possibly nested arbitrarily deep in parentheses, followed by an optional *expression-list* enclosed in parentheses:

`(`... `(`$_{\texttt{opt}}$ *postfix-expression* `)` ...`)`$_{\texttt{opt}}$ `(`*expression-list*$_{\texttt{opt}}$`)`

Overload resolution is required if the *postfix-expression* yields the name of a function, an object of class type, or a set of pointers-to-function.

2 Subclauses 13.2.1.1.1 and 13.2.1.1.2, respectively, describe how overload resolution is used in the first two cases to determine the function to call.

3 The third case arises from a *postfix-expression* of the form `&F`, where `F` names a set of overloaded functions. In the context of a function call, the set of functions named by `F` shall contain only non-member functions and static member functions[68]. And in this context using `&F` behaves the same as using the name

[68] If `F` names a non-static member function, `&F` is a pointer-to-member, which cannot be used with the function call syntax.

F by itself. Thus, (&F)(*expression-list*$_{opt}$) is simply (F)(*expression-list*$_{opt}$), which is discussed in 13.2.1.1.1. (The resolution of &F in other contexts is described in 13.3.)

13.2.1.1.1 Call to named function [over.call.func]

1 Of interest in this subclause are only those function calls in which the *postfix-expression* ultimately contains a name that denotes one or more functions that might be called. Such a *postfix-expression*, perhaps nested arbitrarily deep in parentheses, has one of the following forms:

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

These represent two syntactic subcategories of function calls: qualified function calls and unqualified function calls.

2 In qualified function calls, the name to be resolved is an *id-expression* and is preceded by an -> or . operator. Since the construct A->B is generally equivalent to (*A).B, the rest of this clause assumes, without loss of generality, that all member function calls have been normalized to the form that uses an object and the . operator. Furthermore, this clause assumes that the *postfix-expression* that is the left operand of the . operator has type "*cv* T" where T denotes a class[69]. Under this assumption, the *id-expression* in the call is looked up as a member function of T following the rules for looking up names in classes (10). If a member function is found, that function and its overloaded declarations constitute the set of candidate functions. Because of the usual name hiding rules, these will all be declared in T or they will all be declared in the same base class of T. The argument list is the *expression-list* in the call augmented by the addition of the left operand of the . operator in the normalized member function call as the implied object argument.

3 In unqualified function calls, the name is not qualified by an -> or . operator and has the more general form of a *primary-expression*. The name is looked up in the context of the function call following the normal rules for name lookup. If the name resolves to a non-member function declaration, that function and its overloaded declarations constitute the set of candidate functions. Because of the usual name hiding rules, these will all be declared in the same block or namespace. The argument list is the same as the *expression-list* in the call. If the name resolves to a member function, then the function call is actually a member function call. If the keyword this is in scope and refers to the class of that member function, then the function call is transformed into a normalized qualified function call using (*this) as the *postfix-expression* to the left of the . operator. The candidate functions and argument list are as described for qualified function calls above. If the keyword this is not in scope or refers to another class, then name resolution found a static member of some class T. In this case, all overloaded declarations of the function name in T become candidate functions and a contrived object of type T becomes the implied object argument[70]. The call is ill-formed, however, if overload resolution selects one of the non-static member functions of T in this case.

13.2.1.1.2 Call to object of class type [over.call.object]

1 If the *primary-expression* E in the function call syntax evaluates to a class object of type "*cv* T", then the set of candidate functions includes at least the function call operators of T. The function call operators of T are obtained by ordinary lookup of the name operator() in the context of (E).operator(). Because of the usual name hiding rules, these will all be declared in T or they will all be declared in the same base class of T.

2 In addition, for each conversion function declared in T of the form

operator *conversion-type-id* () *cv-qualifier*;

[69] Note that cv-qualifiers on the type of objects are significant in overload resolution for both lvalue and rvalue objects.
[70] An implied object argument must be contrived to correspond to the implicit object parameter attributed to member functions during overload resolution. It is not used in the call to the selected function. Since the member functions all have the same implicit object parameter, the contrived object will not be the cause to select or reject a function.

where *conversion-type-id* denotes the type "pointer to function with parameters of type `P1`,...,`Pn` and returning `R`" or type "reference to function with parameters of type `P1`,...,`Pn` and returning `R`", a *surrogate call function* with the unique name *call-function* and having the form

```
R call-function (conversion-type-id F, P1 a1,...,Pn an) { return F (a1,...,an); }
```

is also considered as a candidate function. Similarly, surrogate call functions are added to the set of candidate functions for each conversion function declared in an accessible base class provided the function is not hidden within `T` by another intervening declaration[71].

3 If such a surrogate call function is selected by overload resolution, its body, as defined above, will be executed to convert `E` to the appropriate function and then to invoke that function with the arguments of the call.

4 The argument list submitted to overload resolution consists of the argument expressions present in the function call syntax preceded by the implied object argument `(E)`. When comparing the call against the function call operators, the implied object argument is compared against the implicit object parameter of the function call operator. When comparing the call against a surrogate call funtion, the implied object argument is compared against the first parameter of the surrogate call function. The conversion function from which the surrogate call function was derived will be used in the conversion sequence for that parameter since it converts the implied object argument to the appropriate function pointer or reference required by that first parameter.

13.2.1.2 Operators in expressions [over.match.oper]

1 If no operand of the operator has a type that is a class or an enumeration, the operator is assumed to be a built-in operator and interpreted according to clause 5. For example,

```
class String {
public:
   String (const String&);
   String (char*);
        operator char* ();
};
String operator + (const String&, const String&);

void f(void)
{
   char* p= "one" + "two"; // ill-formed because neither
                           // operand has user defined type
   int I = 1 + 1;          // Always evaluates to 2 even if
                           // user defined types exist which
                           // would perform the operation.
}
```

2 If either operand has a type that is a class or an enumeration, a user-defined operator function might be declared that implements this operator or a user-defined conversion can be necessary to convert the operand to a type that is appropriate for a built-in operator. In this case, overload resolution is used to determine which operator function is to be invoked to implement the operator. Therefore, the operator notation is first transformed to the equivalent function-call notation as summarized in Table 8 (where @ denotes one of the operators covered in the specified subclause).

[71] Note that this construction can yield candidate call functions that cannot be differentiated one from the other by overload resolution because they have identical declarations or differ only in their return type. The call will be ambiguous if overload resolution cannot select a match to the call that is uniquely better than such undifferentiable functions.

Table 8—relationship between operator and function call notation

Subclause	Expression	As member function	As non-member function
13.4.1	`@a`	`(a).operator@ ()`	`operator@ (a)`
13.4.2	`a@b`	`(a).operator@ (b)`	`operator@ (a, b)`
13.4.3	`a=b`	`(a).operator= (b)`	
13.4.5	`a[b]`	`(a).operator[](b)`	
13.4.6	`a->`	`(a).operator-> ()`	
13.4.7	`a@`	`(a).operator@ (0)`	`operator@ (a, 0)`

3 Three sets of candidate functions are constructed as follows:

— If the first operand of the operator is an object or reference to an object of class `X`, the operator could be implemented by a member operator function of `X`. The expression is transformed to a qualified function call per column 3 of Table 8 and a set of candidate functions is constructed for the transformed call according to the rules in 13.2.1.1.1. This set is designated the *member candidates* .

— If the operator is either a unary or binary operator (13.4.1, 13.4.2, or 13.4.7), the operator could be implemented by a non-member operator function. The expression is transformed to an unqualified function call per column 4 of Table 8. The operator name is looked up in the context of the expression following the usual rules for name lookup except that all member functions are ignored. Thus, if the operator name resolves to any declaration, it will be to a non-member function declaration. That function and its overloaded declarations constitute the set of candidate functions designated the *non-member candidates*. Because of the name hiding rules, these will all be declared in the same block or namespace[72].

Box 60

A motion is expected in Valley Forge that would eliminate all name hiding when resolving non-member operator names so that the non-member candidates would include all operators of the same name with a declaration in any enclosing block or namespace.

— In any case, a set of candidate functions, called the `built-in` candidates, is constructed. For the binary operator `,` or the unary operator `&`, the built-in candidates set is empty. For all other operators, the built-in candidates include all of the built-in operators defined in 13.5 that, compared to the given operator,

  — have the same operator name, and

  — accept the same number of operands, and

[72] Note that the look up rules for operators in expressions are different than the lookup rules for operator function names in a function call as shown in the following example:

```
struct A { };
void operator + (A, A);

struct B {
  void operator + (B);
  void f ();
};

A a;

void B::f() {
  operator+ (a,a);       // ERROR - global operator hidden by member
  a + a;                 // OK - calls global operator+
}
```

— accept operand types to which the given operand or operands can be converted according to 13.2.3.1.

4 For the built-in assignment operators, conversions of the left operand are restricted as follows:

— no temporaries are introduced to hold the left operand

— no user-defined conversions are applied to achieve a type match with it

5 For all other operators, no such restrictions apply.

6 If a built-in candidate is selected by overload resolution, any class operands are first converted to the appropriate type for the operator. Then the operator is treated as the corresponding built-in operator and interpreted according to clause 5. The set of candidate functions for overload resolution is the union of the member candidates, the non-member candidates, and the built-in candidates. The argument list contains all of the operands of the operator.

7 If the operator is the binary operator ,or the unary operator & and overload resolution is unsuccessful, then the operator is assumed to be the built-in operator and interpreted according to clause 5.

13.2.1.3 Initialization by user-defined conversions [over.match.user]

1 Under the conditions specified in 8.5 and 8.5.3, a user-defined conversion can be invoked to convert the *assignment-expression* of an *initializer-clause* to the type of the object being initialized (which might be a temporary in the reference case). Overload resolution is used to select the user-defined conversion to be invoked. Assuming that "*cv1* `T`" is the type of the object being initialized, the candidate functions are selected as follows:

— When `T` is a class type, the constructors of `T` are candidate functions

— When the type of the *assignment-expression* is a class type "*cv* `S`", the conversion functions of `S` and its base classes are considered. Those that are not hidden within `S` and yield type "*cv2* `T`" or a type that can be converted to type "*cv2* `T`," for any *cv2* that is the same cv-qualification as, or lesser cv-qualification than, *cv1*, via a standard conversion sequence (13.2.3.1.1) are candidate functions

2 In both cases, the argument list has one argument, which is the *assignment-expression* of the *initializer-clause*. This argument will be compared against the first parameter of the constructors and against the implicit object parameter of the conversion functions.

3 Because only one user-defined conversion is allowed in an implicit conversion sequence, special rules apply when selecting the best user-defined conversion (13.2.3, 13.2.3.1).

13.2.1.4 Initialization by constructor [over.match.ctor]

1 When objects of classes with constructors are initialized with a parenthesized *expression-list* (12.6.1), overload resolution selects the constructor. The candidate functions are all the constructors of the class of the object being initialized. The argument list is the *expression-list* within the parentheses of the initializer.

13.2.2 Viable functions [over.match.viable]

1 From the set of candidate functions constructed for a given context (13.2.1), a set of viable functions is chosen, from which the best function will be selected by comparing argument conversion sequences for the best fit (13.2.3). The selection of viable functions considers relationships between arguments and function parameters other than the ranking of conversion sequences.

2 First, to be a viable function, a candidate function shall have enough parameters to agree in number with the arguments in the list.

— If there are *m* arguments in the list, all candidate functions having exactly *m* parameters are viable.

— A candidate function having fewer than *m* parameters is viable only if it has an ellipsis in its parameter list (8.3.5). For the purposes of overload resolution, its parameter list is extended to the right with ellipses so that there are exactly *m* parameters.

— A candidate function having more than *m* parameters is viable only if the *(m+1)*–st parameter has a default initializer (8.3.6). For the purposes of overload resolution, the parameter list is truncated on the right, so that there are exactly *m* parameters.

3 Second, for `F` to be a viable function, there shall exist for each argument an *implicit conversion sequence* (13.2.3.1) that converts that argument to the corresponding parameter of `F`. If the parameter has reference type, the implicit conversion sequence includes the operation of binding the reference, and the fact that a reference to non-`const` cannot be bound to an rvalue can affect the viability of the function (see 13.2.3.1.4).

13.2.3 Best Viable Function [over.match.best]

1 Let ICS*i*(`F`) denote the implicit conversion sequence that converts the *i*-th argument in the list to the type of the *i*-th parameter of viable function `F`. Subclause 13.2.3.1 defines the implicit conversion sequences and subclause 13.2.3.2 defines what it means for one implicit conversion sequence to be a better conversion sequence or worse conversion sequence than another. Given these definitions, a viable function `F1` is defined to be a *better* function than another viable function `F2` if for all arguments *i*, ICS*i*(`F1`) is not a worse conversion sequence than ICS*i*(`F2`), and then

— for some argument *j*, ICS*j*(`F1`) is a better conversion sequence than ICS*j*(`F2`), or, if not that,

— `F1` is a non-template function and `F2` is a template function, or, if not that,

— the context is an initialization by user-defined conversion (see 8.5 and 13.2.1.3) and the standard conversion sequence from the return type of `F1` to the destination type (i.e., the type of the entity being initialized) is a better conversion sequence than the standard conversion sequence from the return type of `F2` to the destination type. For example,

```
struct A {
    A();
    operator int();
    operator double();
} a;
int i = a;      // a.operator int() followed by no conversion is better
                // than a.operator double() followed by a conversion
                // to int
float x = a;    // ambiguous: both possibilities require conversions,
                // and neither is better than the other
```

2 If there is exactly one viable function that is a better function than all other viable functions, then it is the one selected by overload resolution; otherwise the call is ill-formed[73].

3 Examples:

[73] The algorithm for selecting the best viable function is linear in the number of viable functions. Run a simple tournament to find a function `W` that is not worse than any opponent it faced. Although another function `F` that `W` did not face might be better than `W`, `F` cannot be the best function because at some point in the tournament `F` encountered another function `G` such that `F` was not better than `G`. Hence, `W` is either the best function or there is no best function. So, make a second pass over the viable functions to verify that `W` is better than all other functions.

```
void Fcn(const int*,  short);
void Fcn(int*, int);

int i;
short s = 0;

Fcn(&i, s);       // is ambiguous because
                  // &i -> int* is better than &i -> const int*
                  // but s -> short is also better than s -> int

Fcn(&i, 1L);      // calls Fcn(int*, int), because
                  // &i -> int* is better than &i -> const int*
                  // and 1L -> short and 1L -> int are indistinguishable

Fcn(&i,'c');      // calls Fcn(int*, int), because
                  // &i -> int* is better than &i -> const int*
                  // and 'c' -> int is better than 'c' -> short
```

13.2.3.1 Implicit conversion sequences [over.best.ics]

1 An *implicit conversion sequence* is a sequence of conversions used to convert an argument in a function call to the type of the corresponding parameter of the function being called. The sequence of conversions is governed by the rules for initialization of an object or reference by a single expression (8.5 and 8.5.3).

2 Implicit conversion sequences are concerned only with the type, cv-qualification, and lvalue-ness of the argument and how these are converted to match the corresponding properties of the parameter. Other properties, such as the lifetime, storage class, alignment, or accessibility of the argument and whether or not the argument is a bit-field are ignored. So, although an implicit conversion sequence can be defined for a given argument-parameter pair, the conversion from the argument to the parameter might still be ill-formed in the final analysis.

3 Except in the context of an initialization by user-defined conversion (13.2.1.3), a well-formed implicit conversion sequence is one of the following forms:

— a *standard conversion sequence* (13.2.3.1.1),

— a *user-defined conversion sequence* (13.2.3.1.2), or

— an *ellipsis conversion sequence* (13.2.3.1.3).

4 In the context of an initialization by user-defined conversion (i.e., when considering the argument of a user-defined conversion function; see 13.2.1.3), only standard conversion sequences and ellipsis conversion sequences are allowed.

5 When initializing a reference, the operation of binding the reference to an object or temporary occurs after any conversion. The binding operation is not a conversion, but it is considered to be part of a standard conversion sequence, and it can affect the rank of the conversion sequence. See 13.2.3.1.4.

6 In all contexts, when converting to the implicit object parameter or when converting to the left operand of an assignment operation only standard conversion sequences that create no temporary object for the result are allowed.

7 If no conversions are required to match an argument to a parameter type, the implicit conversion sequence is the standard conversion sequence consisting of the identity conversion (13.2.3.1.1).

8 If no sequence of conversions can be found to convert an argument to a parameter type or the conversion is otherwise ill-formed, an implicit conversion sequence cannot be formed.

9 If several different sequences of conversions exist that each convert the argument to the parameter type, the implicit conversion sequence is a sequence among these that is not worse than all the rest according to 13.2.3.2[74]. If that conversion sequence in not better than all the rest and a function that uses such an

[74] This rule prevents a function from becoming non-viable because of an ambiguous conversion sequence for one of its parameters. Consider this example,

implicit conversion sequence is selected as the best viable function, then the call will be ill-formed because the conversion of one of the arguments in the call is ambiguous.

10 The three forms of implicit conversion sequences mentioned above are defined in the following subclauses.

13.2.3.1.1 Standard conversion sequences [over.ics.scs]

1 Table 9 summarizes the conversions defined in clause 4 and partitions them into four disjoint categories: Lvalue Transformation, Qualification Adjustment, Promotion, and Conversion. Note that these categories are orthogonal with respect to lvalue-ness, cv-qualification, and data representation: the Lvalue Transformations do not change the cv-qualification or data representation of the type; the Qualification Adjustments do not change the lvalue-ness or data representation of the type; and the Promotions and Conversions do not change the lvalue-ness or cv-qualification of the type.

2 A standard conversion sequence is either the Identity conversion by itself or consists of one to four conversions from the other four categories. At most one conversion from each category is allowed in a single standard conversion sequence. If there are two or more conversions in the sequence, the conversions are applied in the canonical order: **Lvalue Transformation**, **Promotion**, **Conversion**, **Qualification Adjustment**.

3 Each conversion in Table 9 also has an associated rank (Exact Match, Promotion, or Conversion). These are used to rank standard conversion sequences (13.2.3.2). The rank of a conversion sequence is determined by considering the rank of each conversion in the sequence and the rank of any reference binding (13.2.3.1.4). If any of those has Conversion rank, the sequence has Conversion rank; otherwise, if any of those has Promotion rank, the sequence has Promotion rank; otherwise, the sequence has Exact Match rank.

```
class B;
class A { A (B&); };
class B { operator A (); };
class C { C (B&); };
f(A) { }
f(C) { }
B b;
f(b);   // ambiguous since b -> C via constructor and
        // b -> A via constructor or conversion function.
```

If it were not for this rule, `f(A)` would be eliminated as a viable function for the call `f(b)` causing overload resolution to select `f(C)` as the function to call even though it is not clearly the best choice. On the other hand, if an `f(B)` were to be declared then `f(b)` would resolved to that `f(B)` because the exact match with `f(B)` is better than any of the sequences required to match `f(A)`.

Table 9—conversions

Conversion	Category	Rank	Subclause
No conversions required	Identity	Exact Match	
Lvalue-to-rvalue conversion	Lvalue Transformation		4.1
Array-to-pointer conversion			4.2
Function-to-pointer conversion			4.3
Qualification conversions	Qualification Adjustment		4.4
Integral promotions	Promotion	Promotion	4.5
Floating point promotion			4.6
Integral conversions	Conversion	Conversion	4.7
Floating point conversions			4.8
Floating-integral conversions			4.9
Pointer conversions			4.10
Pointer to member conversions			4.11
Base class conversion			4.12
Boolean conversions			4.13

13.2.3.1.2 User-defined conversion sequences [over.ics.user]

1 A user-defined conversion sequence consists of an initial standard conversion sequence followed by a user-defined conversion (12.3) followed by a second standard conversion sequence. If the user-defined conversion is specified by a constructor (12.3.1), the initial standard conversion sequence converts the source type to the type required by the argument of the constructor. If the user-defined conversion is specified by a conversion function (12.3.2), the initial standard conversion sequence converts the source type to the implicit object parameter of the conversion function.

2 The second standard conversion sequence converts the result of the user-defined conversion to the target type for the sequence. Since an implicit conversion sequence is an initialization, the special rules for initialization by user-defined conversion apply when selecting the best user-defined conversion for a user-defined conversion sequence (see 13.2.3 and 13.2.3.1)

3 It should be noted that a conversion of an expression of class type to the same class type or to a base class of that type is a standard conversion rather than a user-defined conversion in spite of the fact that a copy constructor (i.e., a user-defined conversion function) is called.

13.2.3.1.3 Ellipsis conversion sequences [over.ics.ellipsis]

1 An ellipsis conversion sequence occurs when an argument in a function call is matched with the ellipsis parameter specification of the function called.

13.2.3.1.4 Reference binding [over.ics.ref]

1 The operation of binding a reference is not a conversion, but for the purposes of overload resolution it is considered to be part of a standard conversion sequence (specifically, it is the last step in such a sequence).

2 A standard conversion sequence cannot be formed if it requires binding a reference to non-`const` to an rvalue (except when binding an implicit object parameter; see the special rules for that case in 13.2.1). This means, for example, that a candidate function cannot be a viable function if it has a non-`const` reference parameter (other than the implicit object parameter) and the corresponding argument is a temporary or would require one to be created to initialize the reference (see 8.5.3).

3 Other restrictions on binding a reference to a particular argument do not affect the formation of a standard conversion sequence, however. For example, a function with a "reference to `int`" parameter can be a viable candidate even if the corresponding argument is an `int` bit-field. The formation of implicit conversion sequences treats the `int` bit-field as an `int` lvalue and finds an exact match with the parameter. If the function is selected by overload resolution, the call will nonetheless be ill-formed because of the prohibition on binding a non-`const` reference to a bit-field (8.5.3).

4 A reference binding in general has no effect on the rank of a standard conversion sequence, but there is one exception: the binding of a reference to a (possibly cv-qualified) class to an expression of a (possibly cv-qualified) class derived from that class gives the overall standard conversion sequence Conversion rank.

13.2.3.2 Ranking implicit conversion sequences [over.ics.rank]

1 This clause defines a partial ordering of implicit conversion sequences based on the relationships *better conversion sequence* and *better conversion*. If an implicit conversion sequence S1 is defined by these rules to be a better conversion sequence than S2, then it is also the case that S2 is a *worse conversion sequence* than S1. If conversion sequence S1 is neither better than nor worse than conversion sequence S2, S1 and S2 are said to be *indistinguishable conversion sequences*.

2 When comparing the basic forms of implicit conversion sequences (as defined in 13.2.3.1)

— A standard conversion sequence (13.2.3.1.1) is a better conversion sequence than a user-defined conversion sequence or an ellipsis conversion sequence

— A user-defined conversion sequence (13.2.3.1.2) is a better conversion sequence than an ellipsis conversion sequence (13.2.3.1.3)

3 Two implicit conversion sequences of the same form are indistinguishable conversion sequences unless one of the following rules apply:

— Standard conversion sequence `S1` is a better conversion sequence than standard conversion sequence `S2` if

  — `S1` is a proper subsequence of `S2`, or, if not that,

  — the dominant conversion of `S1` is better than the dominant conversion of `S2` (by the rules defined below), or, if not that,

  — `S1` and `S2` differ only in their qualification conversion and they yield types identical except for cv-qualifiers and `S2` adds all the qualifiers that `S1` adds (and in the same places) and `S2` adds yet more cv-qualifiers than `S1`, or the similar case with reference binding (see the definition of *reference-compatible with added qualification* in 8.5.3).

— User-defined conversion sequence `U1` is a better conversion sequence than another user-defined conversion sequence `U2` if they contain the same user-defined conversion operator or constructor and if the second standard conversion sequence of `U1` is better than the second standard conversion sequence of `U2`.

4 Standard conversions are ordered by their ranks: an Exact Match is a better conversion than a Promotion, which is a better conversion than a Conversion. Two conversions with the same rank are indistinguishable unless one of the following rules applies:

— If class `B` is derived directly or indirectly from class `A`, conversion of `B*` to `A*` is better than conversion of `B*` to `void*`.

— If class `B` is derived directly or indirectly from class `A` and class `C` is derived directly or indirectly from `B`,

  — conversion of `C*` to `B*` is better than conversion of `C*` to `A*`

  — Binding of an expression of type `C` to a reference of type `B&` is better than binding an expression of type `C` to a reference of type `A&`

— conversion of `A::*` to `B::*` is better than conversion of `A::*` to `C::*`

13.3 Address of overloaded function [over.over]

1 A use of a function name without arguments selects, among all functions of that name that are in scope, the (only) function that exactly matches the target. The target can be

— an object being initialized (8.5)

— the left side of an assignment (5.17)

— a parameter of a function (5.2.2)

— a parameter of a user-defined operator (13.4)

— the return value of a function, operator function, or conversion (6.6.3)

— an explicit type conversion (5.2.3, 5.4)

2 Non-member functions match targets of type "pointer-to-function;" member functions match targets of type "pointer-to-member-function."

3 Note that if `f()` and `g()` are both overloaded functions, the cross product of possibilities must be considered to resolve `f(&g)`, or the equivalent expression `f(g)`.

4 For example,

```
int f(double);
int f(int);
(int (*)(int))&f;            // cast expression as selector
int (*pfd)(double) = &f;     // selects f(double)
int (*pfi)(int) = &f;        // selects f (int)
int (*pfe)(...) = &f;        // error: type mismatch
```

The last initialization is ill-formed because no `f()` with type `int(...)` has been defined, and not because of any ambiguity.

5 Note also that there are no standard conversions (4) of one pointer-to-function type or pointer-to-member-function into another (4.10). In particular, even if `B` is a public base of `D` we have

```
D* f();
B* (*p1)() = &f;        // error

void g(D*);
void (*p2)(B*) = &g;    // error
```

6 Note that if the target type is a pointer to member function, the function type of the pointer to member is used to select the member function from a set of overloaded member functions. For example:

```
struct X {
    int f(int);
    static int f(long);
};

int (X::*p1)(int)  = &X::f;    // OK
int    (*p2)(int)  = &X::f;    // error: mismatch
int    (*p3)(long) = &X::f;    // OK
int (X::*p4)(long) = &X::f;    // error: mismatch
int (X::*p5)(int)  = &(X::f); // error: wrong syntax for
                               // pointer to member
int    (*p6)(long) = &(X::f); // OK
```

13.4 Overloaded operators [over.oper]

1 A function declaration having one of the following *operator-function-id*s as its name declares an *operator function*. An operator function is said to *implement* the operator named in its *operator-function-id*.

operator-function-id:
    `operator` *operator*

operator: one of
```
new  delete    new[]     delete[]
+    -    *    /    %    ^    &    |    ~
!    =    <    >    +=   -=   *=   /=   %=
^=   &=   |=   <<   >>   >>=  <<=  ==   !=
<=   >=   &&   ||   ++   --   ,    ->*  ->
()   []
```

The last two operators are function call (5.2.2) and subscripting (5.2.1).

2 Both the unary and binary forms of

```
+    -    *    &
```

can be overloaded.

3 The following operators cannot be overloaded:

```
.    .*   ::   ?:
```

nor can the preprocessing symbols `#` and `##` (16).

4 Operator functions are usually not called directly; instead they are invoked to evaluate the operators they implement (13.4.1 - 13.4.7). They can be explicitly called, though. For example,

```
complex z = a.operator+(b);  // complex z = a+b;
void* p = operator new(sizeof(int)*n);
```

5 The allocation and deallocation functions, `operator new`, `operator new[]`, `operator delete` and `operator delete[]`, are described completely in 12.5. The attributes and restrictions found in the rest of this section do not apply to them unless explicitly stated in 12.5.

6 An operator function shall either be a non-static member function or, be a non-member function and have at least one parameter whose type is a class, a reference to a class, an enumeration, or a reference to an enumeration. It is not possible to change the precedence, grouping, or number of operands of operators. The meaning of the operators =, (unary) `&`, and `,` (comma), predefined for each type, can be changed for specific types by defining operator functions that implement these operators. Operator functions are inherited the same as other functions, but because an instance of `operator=` is automatically constructed for each class (12.8, 13.4.3), `operator=` is never inherited by a class from its bases.

7 The identities among certain predefined operators applied to basic types (for example, `++a` ≡ `a+=1`) need not hold for operator functions. Some predefined operators, such as +=, require an operand to be an lvalue when applied to basic types; this is not required by operator functions.

8 An operator function cannot have default arguments (8.3.6).

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 can be implemented by a non-static member function (9.4) with no parameters or a non-member function with one parameter. Thus, for any prefix unary operator `@`, `@x` can be interpreted as either `x.operator@()` or `operator@(x)`. If both forms of the operator function have been declared, the rules in 13.2.1.2 determine which, if any, interpretation is used. See 13.4.7 for an explanation of the postfix unary operators `++` and `--`.

2 The unary and binary forms of the same operator are considered to have the same name. Consequently, a unary operator can hide a binary operator from an enclosing scope, and vice versa.

13.4.2 Binary operators [over.binary]

1 A binary operator can be implemented either by a non-static member function (9.4) with one parameter or by a non-member function with two parameters. Thus, for any binary operator `@`, `x@y` can be interpreted as either `x.operator@(y)` or `operator@(x,y)`. If both forms of the operator function have been declared, the rules in 13.2.1.2 determines which, if any, interpretation is used.

13.4.3 Assignment [over.ass]

1 An overloaded assignment operator shall be a non-static member function with exactly one parameter. Because an instance of `operator=` is constructed for each class (12.8), it is never inherited by a derived class.

2 A *copy assignment operator* `operator=` is a non-static member function of class `X` with exactly one parameter of type `X&` or `const X&`. 12.8 describes the copy assignment operator.

13.4.4 Function call [over.call]

1 `operator()` shall be a non-static member function. It implements the function call syntax

postfix-expression (*expression-list*$_{opt}$)

where the *postfix-expression* evaluates to a class object and the possibly empty *expression-list* matches the parameter list of an `operator()` member function of the class. Thus, a call `x(arg1,arg2,arg3)` is interpreted as `x.operator()(arg1,arg2,arg3)` for a class object `x` of type `T` if `T::operator()(T1, T2, T3)` exists and if the operator is selected as the best match function by the overload resolution mechanism (13.2.3).

13.4.5 Subscripting [over.sub]

1 `operator[]` shall be a non-static member function. It implements the subscripting syntax

postfix-expression [*expression*]

Thus, a subscripting expression `x[y]` is interpreted as `x.operator[](y)` for a class object `x` of type `T` if `T::operator()(T1)` exists and if the operator is selected as the best match function by the overload resolution mechanism (13.2.3).

13.4.6 Class member access [over.ref]

1 `operator->` shall be a non-static member function taking no parameters. It implements class member access using `->`

postfix-expression -> *primary-expression*

An expression `x->m` is interpreted as `(x.operator->())->m` for a class object `x` of type `T` if `T::operator->()` exists and if the operator is selected as the best match function by the overload resolution mechanism (13.2). It follows that `operator->` must return either a pointer to a class that has a member `m` or an object of or a reference to a class for which `operator->` is defined.

13.4.7 Increment and decrement [over.inc]

1 The prefix and postfix increment operators can be implemented by a function called `operator++`. If this function is a member function with no parameters, or a non-member function with one class parameter, it defines the prefix increment operator ++ for objects of that class. If the function is a member function with one parameter (which shall be of type `int`) or a non-member function with two parameters (the second shall be of type `int`), it defines the postfix increment operator ++ for objects of that class. When the postfix increment is called, the `int` argument will have value zero. For example,

```
class X {
public:
    const X&   operator++();      // prefix ++a
    const X&   operator++(int);   // postfix a++
};

class Y {
public:
};
const Y&   operator++(Y&);        // prefix ++b
const Y&   operator++(Y&, int);   // postfix b++

void f(X a, Y b)
{
    ++a;        // a.operator++();
    a++;        // a.operator++(0);
    ++b;        // operator++(b);
    b++;        // operator++(b, 0);

    a.operator++();     // explicit call: like ++a;
    a.operator++(0);    // explicit call: like a++;
    operator++(b);      // explicit call: like ++b;
    operator++(b, 0);   // explicit call: like b++;
}
```

2 The prefix and postfix decrement operators `--` are handled similarly.

13.5 Built-in operators [over.built]

1 The built-in operators (5) participate in overload resolution (13.2.1.2) as though declared as specified in this section. For `operator,` and unary `operator&`, a built-in operator is selected only if there are no user-defined operator candidates. For all other built-in operators, since they take only operands with non-class type, and operator overload resolution occurs only when an operand expression originally has class type, operator overload resolution can resolve to a built-in operator only when an operand has a class type which has a user-defined conversion to a non-class type appropriate for the operator.

2 In this section, the term *promoted integral type* is used to refer to those integral types which are preserved by integral promotion (including e.g. `int` but excluding e.g. `char`). Similarly, the term *promoted arithmetic type* refers to promoted integral types plus floating types.

3 For every pair (*T*, *VQ*), where *T* is an arithmetic type, and *VQ* is either `volatile` or empty, there exist

```
VQ T&    operator++(VQ T&);
VQ T&    operator--(VQ T&);
T        operator++(VQ T&, int);
T        operator--(VQ T&, int);
```

4 For every pair (*T*, *VQ*), where *T* is a cv-qualified or unqualified complete object type, and *VQ* is either `volatile` or empty, there exist

```
T*VQ&    operator++(T*VQ&);
T*VQ&    operator--(T*VQ&);
T*       operator++(T*VQ&, int);
T*       operator--(T*VQ&, int);
```

5 For every cv-qualified or unqualified complete object type *T*, there exists

```
T&       operator*(T*);
```

6 For every function type *T*, there exists

```
T&       operator*(T*);
```

7 For every type *T*, there exist

```
T*       operator&(T&);
T*       operator+(T*);
```

8 For every promoted arithmetic type *T*, there exist

```
T        operator+(T);
T        operator-(T);
```

9 For every promoted integral type *T*, there exists

```
T        operator~(T);
```

10 For every quadruple (*C*, *T*, *CV1*, *CV2*), where *C* is a class type, *T* is a complete object type or a function type, and *CV1* and *CV2* are *cv-qualifier-seq*s, there exists

```
CV12 T&  operator->*(CV1 C*, CV2 T C::*);
```

where *CV12* is the union of *CV1* and *CV2*.

11 For every pair of promoted arithmetic types *L* and *R*, there exist

```
LR       operator*(L, R);
LR       operator/(L, R);
LR       operator+(L, R);
LR       operator-(L, R);
bool     operator<(L, R);
bool     operator>(L, R);
bool     operator<=(L, R);
bool     operator>=(L, R);
bool     operator==(L, R);
bool     operator!=(L, R);
```

where *LR* is the result of the usual arithmetic conversions between types *L* and *R*.

12 For every pair of types *T* and *I*, where *T* is a cv-qualified or unqualified complete object type and *I* is a promoted integral type, there exist

```
T*       operator+(T*, I);
T&       operator[](T*, I);
T*       operator-(T*, I);
T*       operator+(I, T*);
T&       operator[](I, T*);
```

13 For every triple (*T*, *CV1*, *CV2*), where *T* is a complete object type, and *CV1* and *CV2* are *cv-qualifier-seq*s, there exists

```
ptrdiff_t operator-(CV1 T*, CV2 T*);
```

14 For every triple (*T*, *CV1*, *CV2*), where *T* is any type, and *CV1* and *CV2* are *cv-qualifier-seq*s, there exist

```
bool     operator<(CV1 T*, CV2 T*);
bool     operator>(CV1 T*, CV2 T*);
bool     operator<=(CV1 T*, CV2 T*);
bool     operator>=(CV1 T*, CV2 T*);
bool     operator==(CV1 T*, CV2 T*);
bool     operator!=(CV1 T*, CV2 T*);
```

15 For every quadruple (*C*, *T*, *CV1*, *CV2*), where *C* is a class type, *T* is any type, and *CV1* and *CV2* are *cv-qualifier-seq*s, there exist

```
bool    operator==(CV1 T C::*, CV2 T C::*);
bool    operator!=(CV1 T C::*, CV2 T C::*);
```

16 For every pair of promoted integral types *L* and *R*, there exist

```
LR      operator%(L, R);
LR      operator&(L, R);
LR      operator^(L, R);
LR      operator|(L, R);
L       operator<<(L, R);
L       operator>>(L, R);
```

where *LR* is the result of the usual arithmetic conversions between types *L* and *R*.

17 For every triple (*L*, *VQ*, *R*), where *L* is an arithmetic type, *VQ* is either `volatile` or empty, and *R* is a promoted arithmetic type, there exist

```
VQ L&   operator=(VQ L&, R);
VQ L&   operator*=(VQ L&, R);
VQ L&   operator/=(VQ L&, R);
VQ L&   operator+=(VQ L&, R);
VQ L&   operator-=(VQ L&, R);
```

18 For every pair (*T*, *VQ*), where *T* is any type and *VQ* is either `volatile` or empty, there exists

```
T*VQ&   operator=(T*VQ&, T*);
```

19 For every triple (*T*, *VQ*, *I*), where *T* is a cv-qualified or unqualified complete object type, *VQ* is either `volatile` or empty, and *I* is a promoted integral type, there exist

```
T*VQ&   operator+=(T*VQ&, I);
T*VQ&   operator-=(T*VQ&, I);
```

20 For every triple (*L*, *VQ*, *R*), where *L* is an integral type, *VQ* is either `volatile` or empty, and *R* is a promoted integral type, there exist

```
VQ L&   operator%=(VQ L&, R);
VQ L&   operator<<=(VQ L&, R);
VQ L&   operator>>=(VQ L&, R);
VQ L&   operator&=(VQ L&, R);
VQ L&   operator^=(VQ L&, R);
VQ L&   operator|=(VQ L&, R);
```

21 For every pair of types *L* and *R*, there exists

```
R       operator,(L, R);
```

22 There also exist

```
bool    operator!(bool);
bool    operator&&(bool, bool);
bool    operator||(bool, bool);
```

14 Templates [temp]

1 A class *template* defines the layout and operations for an unbounded set of related types. For example, a single class template `List` might provide a common definition for list of `int`, list of `float`, and list of pointers to `Shape`s. A function *template* defines an unbounded set of related functions. For example, a single function template `sort()` might provide a common definition for sorting all the types defined by the `List` class template.

2 A *template* defines a family of types or functions.

template-declaration:
  `template <` *template-parameter-list* `>` *declaration*

template-parameter-list:
  template-parameter
  template-parameter-list `,` *template-parameter*

The *declaration* in a *template-declaration* shall declare or define a function or a class, define a static data member of a template class, or define a template member of a class. A *template-declaration* is a *declaration*. A *template-declaration* is a definition (also) if its *declaration* defines a function, a class, or a static data member of a template class. There shall be exactly one definition for each template in a program. There can be many declarations. Multiple definitions of a template in a single compilation unit is a required diagnostic. Multiple definitions of a template in different compilation units is a nonrequired diagnostic.

Box 61
This − and all other requirements for unique definitions of templates in this clause − will have to be rephrased to take the ODR into account when the ODR is completely defined.

3 The name of a template obeys the usual scope and access control rules. A *template-declaration* can appear only as a global declaration, as a member of a namespace, as a member of a class, or as a member of a class template. A member template shall not be `virtual`. A destructor shall not be a template. A local class shall not have a member template.

4 A template shall not have C linkage. If the linkage of a template is something other than C or C++, the behavior is implementation-defined.

5 A vector class template might be declared like this:

```
template<class T> class vector {
    T* v;
    int sz;
public:
    vector(int);
    T& operator[](int);
    T& elem(int i) { return v[i]; }
    // ...
};
```

The prefix `template <class T>` specifies that a template is being declared and that a *type-name* `T` will be used in the declaration. In other words, `vector` is a parameterized type with `T` as its parameter. A class template definition specifies how individual classes can be constructed much as a class definition specifies how individual objects can be constructed.

6 A member template can be defined within its class or separately. For example:

```
template<class T> class string {
public:
        template<class T2> compare(const T2&);
        template<class T2> string(const string<T2>& s) { /* ... */ }
        // ...
};

template<class T> template<class T2> string<T>::compare(const T2& s)
{
        // ...
}
```

14.1 Template names [temp.names]

1 A template can be referred to by a *template-id*:

> *template-id:*
>   *template-name* < *template-argument-list* >
>
> *template-name:*
>   *identifier*
>
> *template-argument-list:*
>   *template-argument*
>   *template-argument-list* , *template-argument*
>
> *template-argument:*
>   *assignment-expression*
>   *type-id*
>   *template-name*

2 A *template-id* that names a template class is a *class-name* (9).

3 A *template-id* that names a defined template class can be used exactly like the names of other defined classes. For example:

```
vector<int> v(10);
vector<int>* p = &v;
```

*Template-id*s that name functions are discussed in 14.9.

4 A *template-id* that names a template class that has been declared but not defined can be used exactly like the names of other declared but undefined classes. For example:

```
template<class T> class X; // X is a class template

X<int>* p; // ok: pointer to declared class X<int>
X<int> x;  // error: object of undefined class X<int>
```

5 The name of a template followed by a < is always taken as the beginning of a *template-id* and never as a name followed by the less-than operator. Similarly, the first non-nested > is taken as the end of the *template-argument-list* rather than a greater-than operator. For example:

```
template<int i> class X { /* ... */ }

X< 1>2 >x1; // syntax error
X<(1>2)>x2; // ok

template<class T> class Y { /* ... */ }
Y< X<1> > x3; // ok
```

6 The name of a class template shall not be declared to refer to any other template, class, function, object, namespace, value, or type in the same scope. Unless explicitly specified to have internal linkage, a template in namespace scope has external linkage (3.5). A global template name shall be unique in a program.

7 In a *template-argument*, an ambiguity between a *type-id* and an *expression* is resolved to a *type-id*. For example:

```
template<class T> void f();
template<int I> void f();

void g()
{
        f<int()>(); // ``int()'' is a type-id: call the first f()
}
```

14.2 Name resolution [temp.res]

1 A name used in a template is assumed not to name a type unless it has been explicitly declared to refer to a type in the context enclosing the template declaration or is qualified by the keyword `typename`. For example:

```
// no B declared here

class X;

template<class T> class Y {
        class Z; // forward declaration of member class

        void f() {
                X* a1;    // declare pointer to X
                T* a2;    // declare pointer to T
                Y* a3;    // declare pointer to Y
                Z* a4;    // declare pointer to Z
                typedef typename T::A TA;
                TA* a5;   // declare pointer to T's A
                typename T::A* a6;   // declare pointer to T's A
                T::A* a7; // T::A is not a type name:
                          // multiply T::A by a7
                B* a8;    // B is not a type name:
                          // multiply B by a8
        }
};
```

2 In a template, any use of a *qualified-name* where the qualifier depends on a *template-parameter* can be prefixed by the keyword `typename` to indicate that the *qualified-name* denotes a type.

elaborated-type-specifier:
  ...
  `typename ::`$_{opt}$ *nested-name-specifier identifier full-template-argument-list*$_{opt}$ `;`

full-template-argument-list:
  < *template-argument-list* >

3 If a specialization of that template is generated for a *template-argument* such that the *qualified-name* does not denote a type, the specialization is ill-formed. is a *declaration* that states that *qualified-name* names a type, but gives no clue to what that type might be. The *qualified-name* shall include a qualifier containing a template parameter or a template class name.

4 Knowing which names are type names allows the syntax of every template declaration to be checked. Syntax errors in a template declaration can therefore be diagnosed at the point of the declaration exactly as errors for non-template constructs. Other errors, such as type errors involving template parameters, cannot be diagnosed until later; such errors shall be diagnosed at the point of instantiation or at the point where member functions are generated (14.3). Errors that can be diagnosed at the point of a template declaration, shall be diagnosed there or later together with the dependent type errors. For example:

```
template<class T> class X {
        // ...
        void f(T t, int i, char* p)
        {
                t = i;  // typecheck at point of instantiation,
                        //        or at function generation
                p = i;  // typecheck immediately at template declaration,
                        //        at point of instantiation,
                        //        or at function generation
        }
};
```

No diagnostics shall be issued for a template definition for which a valid specialization can be generated.

5 Three kinds of names can be used within a template definition:

— The name of the template itself, the names of the *template-parameter*s (14.6), and names declared within the template itself.

— Names from the scope of the template definition.

— Names dependent on a *template-argument* (14.7) from the scope of a template instantiation.

6 For example:

```
#include <iostream>
using namespace std;

template<class T> class Set {
        T* p;
        int cnt;
public:
        Set();
        Set<T>(const Set<T>&);
        void printall()
        {
                for (int i = 0; i<cnt; i++)
                        cout << p[i] << '\n';
        }
        // ...
};
```

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 even though the name `operator<<` is known within the definition of `sum()` an a declaration of it can be found in `<iostream>`, the actual declaration of `operator<<` needed to print `p[i]` cannot be known until it is known what type `T` is (14.2.3).

7 If a name can be bound at the point of the template definition and it is not a function called in a way that depends on a *template-parameter* (as defined in 14.2.3), it will be bound at the template definition point and the binding is not affected by later declarations. For example:

```
void f(int);

template<class T> void g(T t)
{
        f(1);      // f(int)
        f(T(1));   // dependent
        f(t);      // dependent
}

void f(char);

void h()
{
f('a'); // will cause two calls of f(int) followed
        // by a call of f(char)
}
```

14.2.1 Locally declared names [temp.local]

1 Within the scope of a class template or a specialization of a template the name of the template is equivalent to the name of the template qualified by the *template-parameter*. Thus, the constructor for `Set` can be referred to as `Set()` or `Set<T>()`. Other specializations (14.5) of the class can be referred to by explicitly qualifying the template name with appropriate *template-argument*s. For example:

```
template<class T> class X {
        X* p;           // meaning X<T>
        X<T>* p2;
        X<int>* p3;
};

template<class T> class Y;

class Y<int> {
        Y* p;           // meaning Y<int>
};
```

See 14.6 for the scope of *template-parameter*s.

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

```
template<class T> class A {
        friend class T;
        class T* p;
        class T;        // error: redeclaration of template parameter T
                        // (a name declaration, not an elaboration)
        // ...
}
```

3 However, a specialization of a template for which a *type-parameter* used this way is not in agreement with the *elaborated-type-specifier* (7.1.5) is ill-formed. For example:

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

A<C> ac;         // ok
A<S> as;         // ok
A<U> au;         // error: parameter T elaborated as a class,
                 // but the argument supplied for T is a union
A<int> ai;       // error: parameter T elaborated as a class,
                 // but the argument supplied for T is an int
A<E> ae;         // error: parameter T elaborated as a class,
                 // but the argument supplied for T is an enumeration
```

14.2.2 Names from the template's enclosing scope [temp.encl]

1 If a name used in a template isn't defined in the template definition itself, names declared in the scope enclosing the template are considered. If the name used is found there, the name used refers to the name in the enclosing context. For example:

```
void g(double);
void h();

template<class T> class Z {
public:
        void f() {
                g(1); // calls g(double)
                h++;  // error: cannot increment function
        }
};

void g(int); // not in scope at the point of the template
             // definition, not considered for the call g(1)
```

In this, a template definition behaves exactly like other definitions. For example:

```
void g(double);
void h();

class ZZ {
public:
        void f() {
                g(1); // calls g(double)
                h++;  // error: cannot increment function
        }
};

void g(int); // not in scope at the point of class ZZ
             // definition, not considered for the call g(1)
```

14.2.3 Dependent names [temp.dep]

1 Some names used in a template are neither known at the point of the template definition nor declared within the template definition. Such names shall depend on a *template-argument* and shall be in scope at the point of the template instantiation (14.3). For example:

```
class Horse { /* ... */ };

ostream& operator<<(ostream&,const Horse&);

void hh(Set<Horse>& h)
{
        h.printall();
}
```

In the call of `Set<Horse>::printall()`, the meaning of the `<<` operator used to print `p[i]` in the definition of `Set<T>::printall()` (14.2), is

```
operator<<(ostream&,const Horse&);
```

This function takes an argument of type `Horse` and is called from a template with a *template-parameter* `T` for which the *template-argument* is `Horse`. Because this function depends on a *template-argument* the call is well-formed.

2 A function call *depends on* a *template-argument* if the call would have a different resolution or no resolution if a type, template, or named constant mentioned in the *template-argument* 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` has the type `T`.

3) A call is resolved by the use of a conversion to `T` without either an argument or a parameter of the called function being of a type that depended on `T` as specified in (1) and (2). For example:

```
struct B { };
struct T : B { };
struct X { operator T(); };

void f(B);

void g(X x)
{
        f(x);  // meaning f( B( x.operator T() ) )
               // so the call f(x) depends on T
}
```

3 This ill-formed template instantiation uses a function that does not depend on a *template-argument*: *

```
template<class T> class Z {
public:
        void f() {
                g(1); // g() not found in Z's context.
                      // Look again at point of instantiation
        }
};

void g(int);

void h(const Z<Horse>& x)
{
        x.f(); // error: g(int) called by g(1) does not depend
               // on template-parameter ``Horse''
}
```

The call `x.f()` gives raise to the specialization:

```
Z<Horse>::f() { g(1); }
```

The call `g(1)` would call `g(int)`, but since that call in no way depends on the *template-argument* `Horse` and because `g(int)` wasn't in scope at the point of the definition of the template, the call `x.f()` is ill-formed.

4 On the other hand:

```
void h(const Z<int>& y)
{
        y.f(); // fine: g(int) called by g(1) depends
               // on template-parameter ``int''
}
```

Here, the call `y.f()` gives raise to the specialization:

```
Z<int>::f() { g(1); }
```

The call `g(1)` calls `g(int)`, and since that call depends on the *template-argument* `int`, the call `y.f()` is acceptable even though `g(int)` wasn't in scope at the point of the template definition.

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

```
struct A {
        struct B { /* ... */ };
        int a;
        int Y;
};

template<class B, class a> struct X : A {
        B b;  // A's B
        a b;  // error: A's a isn't a type name
};
```

6 However, a name from a *template-argument* cannot hide a name declared within a template, a *template-parameter*, or a name from the template's enclosing scopes. For example:

```
int a;

template<class T> struct Y : T {
        struct B { /* ... */ };
        B b;                      // The B defined in Y
        void f(int i) { a = i; } // the global a;
        Y* p;                     // Y<T>
};

Y<A> ya;
```

The members `A::B`, `A::a`, and `A::Y` of the template argument `A` do not affect the binding of names in `Y<A>`.

7 A name of a member can hide the name of a *template-parameter*. 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. When a template is specialized, the names declared in it are declared as if the specialization had been explicitly declared at its point of instantiation. If a template is first specialized as the result of use within a block or class, names declared within the template shall be used only after the template use that caused the specialization. For example:

```
// Assume that Y is not yet declared

template<class T> class X {
        friend class Y;
};

Y* py1;             // ill-formed: Y is not in scope

// Here is the point of instantiation for X<C>
void g()
{
        X<C>* pc;   // does not cause instantiation
        Y* py2;     // ill-formed: Y is not in scope
        X<C> c;     // causes instantiation of X<C>, so
                    // names from X<C> can be used
                    // here on
        Y* py3;     // ok
}
Y* py4;             // ok
```

14.3 Template instantiation [temp.inst]

1 A class generated from a class template is called a generated class. A function generated from a function template is called a generated function. A static data member generated from a static data member template is called a generated static data member. A class defined with a *template-id* as its name is called an explicitly specialized class. A function defined with a *template-id* as its name is called an explicitly specialized function. A static data member defined with a *template-id* as its name is called an explicitly specialized static data member. A specialization is a class, function, or static data member that is either generated or explicitly specialized.

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

14.3.1 Template linkage [temp.linkage]

1 A function template has external linkage, as does a static member of a class template. Every function template shall have the same definition in every translation unit in which it appears.

14.3.2 Point of instantiation [temp.point]

1 The point of instantiation of a template is the point where names dependent on the *template-argument* are bound. That point is immediately before the declaration in the nearest enclosing global or namespace scope containing the first use of the template requiring its definition. This implies that names used in a template definition cannot be bound to local names or class member names from the scope of the template use. They can, however, be bound to names of namespace members. For example:

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

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

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

class C {
        void g(int);

        void m(const Y<int>& h)
        {
                h.f(); // error: g(int) called by g(1) not found
                       //        C::g() not considered
        }
};

namespace N {
        void g(int);

        void n(const Y<int>& h)
        {
                h.f(); // N::g(int) called by g(1)
        }
}
```

2 Names from both the namespace of the template itself and of the namespace containing the point of instantiation of a specialization are used to resolve names for the specialization. Overload resolution is used to chose between functions with the same name in these two namespaces. For example: *

```
namespace NN {
        void g(int);
        void h(int);
        template<class T> void f(T t)
        {
                g(t);
                h(t);
                k(t);
        }
}

namespace MM {
        void g(double);
        void k(double);

        // instantiation point for NN:f(int) and NN::f(double)

        void m()
        {
                NN:f(1);    // indirectly calls NN::g(int),
                            //                  NN::h, and MM::k.
                NN:f(1.0);  // indirectly calls MM::g(double),
                            //                  NN::h, and MM::k.
        }
}
```

If a name is found in both namespaces and overload resolution cannot resolve a use, the program is ill-formed.

3 Each compilation unit in which the definition of a template is used in a way that require definition of a specialization has a point of instantiation for the template. If this causes names used in the template definition to bind to different names in different 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 can be instantiated for implementation purposes.

5 If a virtual function is instantiated, its point of instantiation is immediately following the point of instantiation for its class. *

6 The point of instantiation for a template used inside another template and not instantiated previous to an instantiation of the enclosing template is immediately before the point of instantiation of the enclosing template.

```
namespace N {
        template<class T> class List {
        public:
                T* get();
                // ...
        };
}

template<class K, class V> class Map {
        List<V> lt;
        V get(K);
        //  ...
};

void g(Map<char*,int>& m)
{
        int i = m.get("Nicholas");
        // ...
}
```

This allows instantiation of a used template to be done before instantiation of its user.

7 Implicitly generated template classes, functions, and static data members are placed in the namespace where the template was defined. For example, a call of `lt.get()` from `Map<char*,int>::get()` would place `List<int>::get()` in `N` rather than in the global space. *

Box 62

Name injection from an implicitly generated template function specialization are under debate. That is, it might be banned.

8 If a template for which a definition is in scope is used in a way that involves overload resolution or conversion to a base class, the definition of a template specialization is required. For example:

```
template<class T> class B { /* ... */ };
template<class T> class D : public B<T> { /* ... */ };

void f(void*);
void f(B<int>*);

void g(D<int>* p, D<char>* pp)
{
        f(p); // instantiation of D<int> required: call f(B<int>*)

        B<char>* q = pp; // instantiation of D<char> required:
                         // convert D<char>* to B<char>*
}
```

9 If an instantiation of a class template is required and the template is declared but not defined, the program is ill-formed. For example:

```
template<class T> class X;

X<char> ch; // error: definition of X required
```

10 Recursive instantiation is possible. For example:

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

int fac<0>() { return 1; }

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

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

12 The result of an infinite recursion in instantiation is undefined. In particular, an implementation is allowed to report an infinite recursion as being ill-formed. For example:

```
template<class T> class X {
        X<T>* p; // ok
        X<T*> a; // instantiation of X<T> requires
                 // the instantiation of X<T*> which requires
                 // the instantiation of X<T**> which ...
};
```

13 No program shall explicitly instantiate any template more than once, both explicitly instantiate and explicitly specialize a template, or specialize a template more than once for a given set of *template-arguments*. An implementation is not required to diagnose a violation of this rule.

14 An explicit specialization or explicit instantiation of a template shall be in the namespace in which the template was defined. For example:

```
namespace N {
        template<class T> class X { /* ... */ };
        template<class T> class Y { /* ... */ };
        template<class T> class Z {
                void f(int i) { g(i); }
                // ...
        };

        class X<int> { /* ... */ }; // ok: specialization
                                    //     in same namespace
}

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

template class N::Y<char*>; // ok: explicit instantiation
                            //     in same namespace

class N::Y<double> { /* ... */ }; // ok: specialization
                                  //     in same namespace
```

15 A member function of an explicitly specialized class shall not be implicitly generated from the general template. Instead, the member function shall itself be explicitly specialized. For example:

```
template<class T> struct A {
        void f() { /* ... */ }
};

struct A<int> {
        void f();
};

void h()
{
        A<int> a;
        a.f();  // A<int>::f must be defined somewhere
}

void A<int>::f() { /* ... */ };
```

Thus, an explicit specialization of a class implies the declaration of specializations of all of its members. The definition of each such specialized member which is used shall be provided in some translation unit.

14.3.3 Instantiation of `operator->`

1 If a template class has an `operator->`, that `operator->` can have a return type that cannot be dereferenced by `->` as long as that `operator->` is neither invoked, nor has its address taken, isn't virtual, nor is explicitly instantiated. For example:

```
template<class T> class Ptr {
        // ...
        T* operator->();
};

Ptr<int> pi; // ok
Ptr<Rec> pr; // ok

void f()
{
        pi->m = 7; // error: Ptr<int>::operator->() returns a type
                   //        that cannot be dereference by ->
        pr->m = 7; // ok if Rec has an accessible member m
                   // of suitable type
}
```

14.4 Explicit instantiation [temp.explicit]

1 A class or function specialization can be explicitly instantiated from its template.

2 The syntax for explicit instantiation is:

explicit-instantiation:
  `template` *inst* `;`

inst:
  class-key template-id
  type-specifier-seq template-id `(` *parameter-declaration-clause* `)`

> **Box 63**
> Syntax WG: please check this grammar. It ought to allow any declaration that is not a definition of a class or function with a *template-id* as the name being declared.

For example:

```
template class vector<char>;

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

3 A declaration of the template shall be in scope at the point of explicit instantiation. *

4 A trailing *template-argument* can be left unspecified in an explicit instantiation or explicit specialization of a template function provided it can be deduced from the function argument type. For example:

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

5 The explicit instantiation of a class implies the instantiation of all of its members not previously explicitly specialized in the compilation unit containing the explicit instantiation.

14.5 Template specialization [temp.spec]

1 A specialized template function, template class, or static member of a template can be declared by a declaration where the declared name is a *template-id*, that is:

specialization:
  declaration

For example: *

```
template<class T> class stream;

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

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

void sort<char*>(vector<char*>&) ;
```

Given these declarations, `stream<char>` will be used as the definition of streams of `char`s; other streams will be handled by template classes generated from the class template. Similarly, `sort<char*>` will be used as the sort function for arguments of type `vector<char*>`; other `vector` types will be sorted by functions generated from the template.

2 A declaration of the template being specialized shall be in scope at the point of declaration of a specialization. For example:

```
class X<int> { /* ... */ }; // error: X not a template

template<class T> class X { /* ... */ };

class X<char*> { /* ... */ }; // fine: X is a template
```

3 If a template is explicitly specialized then that specialization shall be declared before the first use of that specialization in every translation unit in which it is used. For example:

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

void f(vector<String>& v)
{
        sort(v); // use general template
                 // sort(vector<T>&), T is String
}

void sort<String>(vector<String>& v); // error: specialize after use
void sort<>(vector<char*>& v); // fine sort<char*> not yet used
```

If a function or class template has been explicitly specialized for a *template-argument* list no specialization will be implicitly generated for that *template-argument* list.

4 Note that a function with the same name as a template and a type that exactly matches that of a template is not a specialization (14.9.4).

14.6 Template parameters [temp.param]

1 The syntax for *template-parameter*s is:

template-parameter:
  type-parameter
  parameter-declaration

type-parameter:
  `class` *identifier*$_{opt}$
  `class` *identifier*$_{opt}$ = *type-id*
  `typename` *identifier*$_{opt}$
  `typename` *identifier*$_{opt}$ = *type-id*
  `template` < *template-parameter-list* > `class` *identifier*$_{opt}$
  `template` < *template-parameter-list* > `class` *identifier*$_{opt}$ = *template-name*

For example:

```
template<class T> class myarray { /* ... */ };

template<class K, class V, template<class T> class C = myarray>
class Map {
        C<K> key;
        C<V> value;
        // ...
};
```

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

3 A *type-parameter* defines its *identifier* to be a *type-name* in the scope of the template declaration. A *type-parameter* shall not be redeclared within its scope (including nested scopes). A non-type *template-parameter* shall not be assigned to or in any other way have its value changed. For example:

```
template<class T, int i> class Y {
        int T;  // error: template-parameter redefined
        void f() {
                char T; // error: template-parameter redefined
                i++;    // error: change of template-argument value
        }
};

template<class X> class X; // error: template-parameter redefined
```

4 A *template-parameter* that could be interpreted as either an *parameter-declaration* or a *type-parameter* (because its *identifier* is the name of an already existing class) is taken as a *type-parameter*. A *template-parameter* hides a variable, type, constant, etc. of the same name in the enclosing scope. For example:

```
class T { /* ... */ };
int i;

template<class T, T i> void f(T t)
{
        T t1 = i;      // template-arguments T and i
        ::T t2 = ::i;  // globals T and i
}
```

Here, the template `f` has a *type-parameter* called `T`, rather than an unnamed non-type parameter of class `T`. There is no semantic difference between `class` and `typename` in a *template-parameter*.

5 There are no restrictions on what can be a *template-argument* type beyond the constraints imposed by the set of argument types (14.7). In particular, reference types and types containing *cv-qualifiers* are allowed. A non-reference *template-argument* cannot have its address taken. When a non-reference *template-argument* is used as an initializer for a reference a temporary is always used. For example:

```
template<const X& x, int i> void f()
{
        &x; // ok
        &i; // error: address of non-reference template-argument

        int& ri = i; // error: non-const reference bound to temporary
        const int& cri = i; // ok: reference bound to temporary
}
```

6 A non-type *template-parameter* shall not 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, value, or template specified after = in a *template-parameter*. A default *template-argument* can be specified in a template declaration or a template definition. The set of default *template-argument*s available for use with a template in a translation unit shall be provided by the first declaration of the template in that unit.

8 If a *template-parameter* has a default argument, all subsequent *template-parameter*s shall have a default argument supplied. For example:

```
template<class T1 = int, class T2> class B; // error
```

9 The scope of a *template-argument* extends from its point of declaration until the end of its template. In particular, a *template-parameter* can be used in the declaration of subsequent *template-parameter*s and their default arguments. 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.

10 A *template-parameter* can be used in the specification of base classes. For example:

```
template<class T> class X : public vector<T> { /* ... */ };
template<class T> class Y : public T { /* ... */ };
```

Note that the use of a *template-parameter* as a base class implies that a class used as a *template-argument* must be defined and not just declared.

14.7 Template arguments [temp.arg]

1 The types of the *template-argument*s specified in a *template-id* shall 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 A non-type non-reference *template-argument* shall be a *constant-expression* of non-floating type, the address of an object or a function with external linkage, or a non-overloaded pointer to member. The address of an object or function shall be expressed as `&f`, plain `f` (for function only), or `&X::f` where `f` is the function or object name. In the case of `&X::f`, `X` shall be a (possibly qualified) name of a class and `f` the name of a static member of `X`. A pointer to member shall be expressed as `&X::m` where `X` is a (possibly qualified) name of a class and `m` is the member name. In particular, a string literal (2.9.4) is *not* an acceptable *template-argument* because a string literal is the address of an object with static linkage. For example:

```
template<class T, char* p> class X {
        // ...
        X(const char* q) { /* ... */ }
};

X<int,"Studebaker"> x1; // error: string literal as template-argument

char* p = "Vivisectionist";
X<int,p> x2; // ok
```

3 Similarly, addresses of array elements and non-static class members are not acceptable as `template-arguments`. For example:

```
int a[10];
struct S { int m; static int s; } s;

X<&a[2],p> x3; // error: address of element
X<&s.m,p> x4;  // error: address of member
X<&s.s,p> x5;  // error: address of member (dot operator used)
X<&S::s,p> x6; // ok: address of static member
```

4 Nor is a local type or an type with no linkage name an acceptable *template-argument*. For example:

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

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

5 Similarly, a reference *template-parameter* cannot be bound to a temporary:

```
template<const int& CRI) struct B { /* ... */ };

B<1> b2; // error: temporary required for template argument

int c = 1;
B<c> b1; // ok
```

6 An argument to a *template-parameter* of pointer to function type shall have exactly the type specified by the *template* parameter. This allows selection from a set of overloaded functions. For example:

```
void f(char);
void f(int);

template<void (*pf)(int)> struct A { /* ... */ };

A<&f> a; // selects f(int)
```

7 A template has no special access rights to its *template-argument* types. A *template-argument* shall be accessible at the point where it is used as a *template-argument*. For example:

```
template<class T> class X { /* ... */ };

class Y {
private:
        struct S { /* ... */ };
        X<S> x;  // ok: S is accessible
};

X<Y::S> y; // error: S not accessible
```

8 In addition to the rules for non-reference *template-arguments*, an argument for a *template-parameter* of reference type shall not be a *constant-expression*. In particular, a temporary object is not an acceptable argument to a *template-parameter* of reference type. *

9 When default *template-arguments* are used, a *template-argument* list can be empty. In that case the empty <> brackets shall 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 in the same scope and their *template-argument*s have identical values. For example,

```
template<class E, int size> class buffer;

buffer<char,2*512> x;
buffer<char,1024> y;
```

declares `x` and `y` to be of the same type, and

```
template<class T, void(*err_fct)()> class list { /* ... */ };

list<int,&error_handler1> x1;
list<int,&error_handler2> x2;
list<int,&error_handler2> x3;
list<char,&error_handler2> x4;
```

declares `x2` and `x3` to be of the same type. Their type differs from the types of `x1` and `x4`.

14.9 Function templates [temp.fct]

1 A function template specifies how individual functions can be constructed. A family of sort functions, for example, might be declared like this:

```
template<class T> void sort(vector<T>&);
```

A function template specifies an unbounded set of (overloaded) functions. A function generated from a function template is called a template function, so is an explicit specialization of a function template. Template arguments can either be explicitly specified in a call or be deduced from the function arguments.

14.9.1 Explicit template argument specification [temp.arg.explicit]

1 Template arguments can be specified in a call by qualifying the template function name by the list of *template-argument*s exactly as *template-argument*s are specified in uses of a class template. For example:

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

and

```
template<class U, class V> U convert(V v);

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

Implicit conversions (4) are accepted for a function argument for which the parameter has been fixed by explicit specification of *template-argument*s. For example:

```
template<class T> void f(T);

class complex {
        // ...
        complex(double);
};

void g()
{
        f<complex>(1); // ok, means f<complex>((complex(1))
}
```

Box 64

There is a problem with the explicit qualification of member template functions. Consider:

```
class X {
public:
        template<size_t> X* malloc();
        // ...
};

void f(X* p)
{
        X* pi = p->malloc<200>();
}
```

There is no way of knowing that `X::malloc` is a template name until after type checking. Consequently, this example cannot be syntax analysed.

One solution is ‘‘then do not do that.’’ Another is to provide some form of explicit qualification. For example:

```
X* pi = p-> templatename malloc<200>();
```

or

```
X* pi = p-> template malloc<200>();
```

The latter, use of the keyword `template`, in general clashes with the use of `template` for explicit instantiation (14.4).

14.9.2 Template argument deduction [temp.deduct]

1 Template arguments that can be deduced from the function arguments of a call need not be explicitly specified. For example,

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

and

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

2 A template type argument `T` or a template non-type argument `i` can be deduced from a function argument composed from these elements:

> `T`
> *cv-list* `T`
> `T*`
> `T&`
> `T[`*integer-constant*`]`
> *class-template-name*`<T>`
> *type*`(*)(T)`
> *type* `T::*`
> `T(*)()`
> `T(*)(T)`
> *type*`[i]`
> *class-template-name*`<i>`

where `(T)` includes argument lists with more than one argument where at least one argument contains a `T`, and where `()` includes argument lists with arguments that do not contain a `T`. Also, these forms can be used in the same way as `T` is for further composition of types. For example,

```
X<int>(*)(char[6])
```

is of the form

> *class-template-name*`<T> (*)(`*type*`[i])`

which is a variant of

> *type* `(*)(T)`

where *type* is `X<int>` and `T` is `char[6]`.

3 In addition, a *template-parameter* can be deduced from a function or pointer to member function argument if at most one of a set of overloaded functions provides a unique match. For example:

```
template<class T> void f(void(*)(T,int));

void g(int,int);
void g(char,int);

void h(int,int,int);
void h(char,int);

int m()
{
        f(&g);  // error: ambiguous
        f(&h);  // ok: void h(char,int) is a unique match
}
```

Template arguments shall not be deduced from function arguments involving constructs other than the ones specified in here (14.9.2).

> **Box 65**
> Can a template *template-parameter* be deduced? and if so how? Spicer issue 3.19.

4 Template arguments of an explicit instantiation or explicit specialization are deduced (14.4, 14.5) according to these rules specified for deducing function arguments.

5 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][20]);
```

```
void g(int v[10][20])
{
        f1(v);     // ok: i deduced to be 20
        f1<10>(v); // ok
        f2(v);     // error: cannot deduce template-argument i
        f2<10>(v); // ok
}
```

6 Nontype parameters shall not be used in expressions in the function declaration. The type of the function *template-parameter* shall match the type of the *template-argument* exactly. 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
```

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

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

void g()
{
        f<int,char*,double>("aa",3.0);
        f<int,char*>("aa",3.0); // Z is deduced to be double
        f<int>("aa",3.0); // Y is deduced to be char*, and
                          // Z is deduced to be double
        f("aa",3.0); // error X cannot be deduced

}
```

8 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)
}
```

9 If a template parameter can be deduced from more than one function argument the deduced template parameter shall the same in each case. For example:

```
template<class T> void f(T x, T y) { /* ... */ }

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

int g(A a, B b)
{
        f(a,a);  // ok: T is A
        f(b,b);  // ok: T is B
        f(a,b);  // error T could be A or B
        f(b,a);  // error: T could be A or B
}
```

14.9.3 Overload resolution [temp.over]

1 A template function can be overloaded either by (other) functions of its name or by (other) template functions of that same name. Overloading resolution for template functions and other functions of the same name is done in the following three steps:

1) Look for an exact match (13.2) on functions; if found, call it.

2) Look for a function template from which a function that can be called with an exact match can be generated; if found, call it.

3) Look for match with conversions. For arguments to ordinary functions and for arguments to a template function that corresponds to parameters whose type does not depend on a deduced *template-parameter*, the ordinary best match rules apply. For template functions, only the following conversions listed below applies. After the best matches are found for individual arguments, the intersection rule (_over.match.args_) is used to look for a best match; if found, call it.

Box 66
Rephrase to match Clause 13.

2 For arguments that correspond to parameters whose type depends on a deduced template parameter, the following conversions are allowed:

— For a parameter of the form `B<params>`, where `params` is a template parameter list containing one or more deduced parameters, an argument of type ‘‘class derived from `B<params>`’’ can be converted to `B<params>`. Additionally, for a parameter of the form `B<params>*`, an argument of type ‘‘pointer to class derived from `B<params>`’’ can be converted to `B<params>*`. Similarly for references.[75)]

— A pointer (reference) can be converted to a more qualified pointer (reference) type, according to the rules in 4.10 (_conv.ref_).

— ‘‘array of `T`’’ to ‘‘pointer to `T`.’’

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

3 If no match is found the call is ill-formed. In each case, if there is more than one alternative in the first step that finds a match, the call is ambiguous and is ill-formed.

4 A match on a template (step (2)) implies that a specific template function with parameters that exactly match the types of the arguments will be generated (14.3). Not even trivial conversions (13.2) will be applied in this case.

Box 67
This maybe too strict. See the proposal for a more general overloaded mechanism in N0407/94−0020 (issue 3.9).

5 The same process is used for type matching for pointers to functions (13.3) and pointers to members.

6 Here is an example:

[75)] It would be nice if an argument of type ‘‘`T B::*` where `B` is a base of `D<params>`’’ could be converted to `T D<params>::*`. Unfortunately this would require an unbounded search of possible instantiations.

```
template<class T> T max(T a, T b) { return a>b?a:b; };

void f(int a, int b, char c, char d)
{
    int m1 = max(a,b);  // max(int a, int b)
    char m2 = max(c,d); // max(char a, char b)
    int m3 = max(a,c);  // error: cannot generate max(int,char)
}
```

7 For example, adding

```
int max(int,int);
```

to the example above would resolve the third call, by providing a function that could be called for `max(a,c)` after using the standard conversion of `char` to `int` for `c`.

8 Here is an example involving conversions on a function argument involved in *template-parameter* deduction:

```
template<class T> struct B { /* ... */ };
template<class T> struct D : public B<T> { /* ... */ };
template<class T> void f(B<T>&);

void g(B<int>& bi, D<int>& di)
{
        f(bi);  // f(bi)
        f(di);  // f( (B<int>&)di )
}
```

9 Here is an example involving conversions on a function argument not involved in *template-parameter* deduction:

```
template<class T> void f(T*,int);  // #1
template<class T> void f(T,char);  // #2

void h(int* pi, int i, char c)
{
        f(pi,i);  // #1: f<int>(pi,i)
        f(pi,c);  // #2: f<int*>(pi,c)

        f(i,c);   // #2: f<int>(i,c);
        f(i,i);   // #2: f<int>(i,char(i))
}
```

10 The template definition is needed to generate specializations of a template. However, only a function template declaration is needed to call a specialization. For example,

```
template<class T> void f(T);    // declaration

void g()
{
        f("Annemarie"); // call of f<char*>
}
```

The call of `f` is well formed because of the the declaration of `f`, and the program will be ill-formed unless a definition of `f` is present in some translations unit.

11 In case a call has explicitly qualified *template-argument*s and requires overload resolution, the explicit qualification is used first to determine the set of overloaded functions to be considered and overload resolution then takes place for the remaining arguments. For example:

```
template<class X, class Y> void f(X,Y*);  // #1
template<class X, class Y> void f(X*,Y);  // #2

void g(char* pc, int* pi)
{
        f(0,0); // error: ambiguous: f<int,int>(int,int*)
                   //                 or f<int,int>(int*,int) ?
        f<char*>(pc,pi); // #1: f<char*,int>(char*,int*)
        f<char>(pc,pi);  // #2: f<char,int*>(char*,int*)
}
```

14.9.4 Overloading and specialization [temp.over.spec]

1 A template function can be overloaded by a function with the same type as a potentially generated function. For example:

```
template<class T> T max(T a, T b) { return a>b?a:b; }
int max(int a, int b);

int min(int a, int b);
template<class T> T min(T a, T b) { return a<b?a:b; }
```

Such an overloaded function is a specialization but not an explicit specialization. The declaration simply guides the overload resolution. This implies that a definition of `max(int,int)` and `min(int,int)` will be implicitly generated from the templates. If such implicit instantiation is not wanted, the explicit specialization syntax should be used instead:

```
template<class T> T max(T a, T b) { return a>b?a:b; }
int max<int>(int a, int b);
```

2 Defining a function with the same type as a template specialization that is called is ill-formed. For example:

```
template<class T> T max(T a, T b) { return a>b?a:b; }
int max(int a, int b) { return a>b?a:b; }

void f(int x, int y)
{
        max(x,y); // error: duplicate definition of max()
}
```

If the two definitions of `max()` are not in the same translation unit the diagnostic is not required. If a separate definition of a function `max(int,int)` is needed, the specialization syntax can be used. If the conversions enabled by an ordinary declaration are also needed, both can be used. For example:

```
template<class T> T max(T a, T b) { return a>b?a:b; }
int max<>(int a, int b) { /* ... */ }

void g(char x, int y)
{
        max(x,y); // error: no exact match, and no conversions allowed
}

int max(int,int);

void f(char x, int y)
{
        max(x,y); // max<int>(int(x),y)
}
```

3 An explicit specialization of a function template shall be `inline` or `static` only if it is explicitly declared to be, and independently of whether its function template is. For example:

```
template<class T> void f(T) { /* ... */ }
template<class T> inline T g(T) { /* ... */ }

inline void f<>(int) { /* ... */ } // ok: inline
int g<>(int) { /* ... */ } // ok: not inline
```

14.10 Member function templates [temp.mem.func]

1 A member function of a template class is implicitly a template function with the *template-parameter*s of its class as its *template-parameter*s. For example,

```
template<class T> class vector {
    T* v;
    int sz;
public:
    vector(int);
    T& operator[](int);
    T& elem(int i) { return v[i]; }
    // ...
};
```

declares three function templates. The subscript function might be defined like this:

```
template<class T> T& vector<T>::operator[](int i)
{
    if (i<0 || sz<=i) error("vector: range error");
    return v[i];
}
```

2 The *template-argument* for `vector<T>::operator[]()` will be determined by the vector to which the subscripting operation is applied.

```
vector<int> v1(20);
vector<complex> v2(30);

v1[3] = 7;              // vector<int>::operator[]()
v2[3] = complex(7,8);   // vector<complex>::operator[]()
```

14.11 Friends [temp.friend]

1 A friend function of a template can be a template function or a non-template function. For example,

```
template<class T> class task {
    // ...
    friend void next_time();
    friend task<T>* preempt(task<T>*);
    friend task* prmt(task*);          // task is task<T>
    friend class task<int>;
    // ...
};
```

Here, `next_time()` and `task<int>` become friends of all `task` classes, and each `task` has appropriately typed functions `preempt()` and `prmt()` as friends. The `preempt` functions might be defined as a template.

```
template<class T> task<T>* preempt(task<T>* t) { /* ... */ }
```

2 A friend template shall not be defined within a class. For example:

```
class A {
        friend template<class T> B;    // ok
        friend friend template<class T> f(T); // ok

        friend template<class T> BB { /* ... /* }; // error
        friend template<class T> ff(T){ /* ... /* } // error
};
```

Note that a `friend` declaration can add a name to an enclosing scope (14.2.4).

Box 68

The syntax above isn't allowed by the grammar. The grammar allows only:

```
template<class T> friend B;
```

Is what has been used in the examples up until now a better syntax? I think so, because the template parameter specification is part of the type of what is being defined. However, allowing that requires a minor grammar change. Making

```
template<class T>
```

a *type-specifier* might simplify the grammar while achieving the desired effect.

Box 69

There is no way of declaring a specialization of a static member without also defining it. For example:

```
template<class T> class X {
        static T s;
};

X<int> s; // definition, can't just declare
```

One answer to this is to do nothing and hope there is little real need for a solution. Another answer is to introduce a separate keyword to indicate specialization; see Spicer 6.18 .

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);  // call f<int>(int*)
    f(&b);  // call f<char*>(char**)
}
```

Here `f<int>(int*)` has a static member `s` of type `int` and `f<char*>(char**)` has a static member `s` of type `char*`.

15 Exception handling [except]

1 Exception handling provides a way of transferring control and information from a point in the execution of a program to an *exception handler* associated with a point previously passed by the execution. A handler will be invoked only by a *throw-expression* invoked in code executed in the handler's *try-block* or in functions called from the handler's *try-block*.

try-block:
    `try` *compound-statement handler-seq*

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.[76)]

4 A *throw-expression* initializes a temporary object of the static type of the operand of `throw`, ignoring the top-level *cv-qualifier*s of the operand's type, and uses that temporary to initialize the appropriately-typed variable named in the handler. 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 for the 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 can be initialized directly with the argument of the throw expression.

---

76) If the value thrown has no base classes or is not of class type, this condition is vacuously satisfied.

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:

```
try {
    // ...
}
catch (...) {  // catch all exceptions

    // respond (partially) to exception

    throw;      // pass the exception to some
                // other handler
}
```

7 The exception thrown is the one most recently caught and not finished. An exception is considered caught when initialization is complete for the formal parameter of the corresponding catch clause, or when `terminate()` or `unexpected()` is entered due to a throw. An exception is considered finished when the corresponding catch clause exits.

8 If no exception is presently being handled, executing a *throw-expression* with no operand calls `terminate()` (15.5.1).

15.2 Constructors and destructors [except.ctor]

1 As control passes from a throw-point to a handler, destructors are invoked for all automatic objects constructed since the *try-block* was entered.

2 An object that is partially constructed will have destructors executed only for its fully constructed subobjects. Should a constructor for an element of an automatic array throw an exception, only the constructed elements of that array will be destroyed. If the object or array was allocated in a *new-expression*, the storage occupied by that object is sometimes deleted also (5.3.4).

3 The process of calling destructors for automatic objects constructed on the path from a *try-block* to a *throw-expression* is called "*stack unwinding*."

15.3 Handling an exception [except.handle]

1 The *exception-declaration* in a *handler* describes the type(s) of exceptions that can cause that handler to be executed. The *exception-declaration* shall not denote an incomplete type.

2 A *handler* with type `T`, `const T`, `T&`, or `const T&` is a match for a *throw-expression* with an object of type `E` if

[1] `T` and `E` are the same type, or

[2] `T` is an accessible (4.10) base class of `E` at the throw point, or

[3] `T` is a pointer type and `E` is a pointer type that can be converted to `T` by a standard pointer conversion (4.10) at the throw point.

Box 70

The intent was and is to require no run-time access or ambiguity checking.

Paragraph 3 of 15.1 says that we can't throw an object that would require the handler mechanism to do ambiguity checks.

Point [2] above says, in particular, that an object with a private class can be thrown if and only if the thrower has access to that base. This implies no violation of access, because the thrower could have thrown the private class directly.

This implies that an exception can be caught by a private class (the access check, like the ambiguity check is done at the throw point). This does not require a run-time access check. It does, however, require that a object of a class with a private base class is transmitted to the catch point together with an indication if it can be caught by its private base class.

It has been suggested that this should be simplified by prohibiting the throw of an object of a class with a private base class. It has also been suggested that run-time access checks should be required. In the absence of a proposal for change, the text will be clarified along the lines in this box.

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`.

3 The handlers for a *try-block* are tried in order of appearance. That makes it possible to write handlers that can never be executed, for example by placing a handler for a derived class after a handler for a corresponding base class.

4 A `...` in a handler's *exception-declaration* functions similarly to `...` in a function parameter declaration; it specifies a match for any exception. If present, a `...` handler shall be the last handler for its *try-block*.

5 If no match is found among the handlers for a *try-block*, the search for a matching handler continues in a dynamically surrounding *try-block*. *

6 An exception is considered handled upon entry to a handler. The stack will have been unwound at that point.

7 If no matching handler is found in a program, the function `terminate()` (15.5.1) is called. Whether or not the stack is unwound before calling `terminate()` is implementation-defined.

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 71
Should it be possible to use more general types than *type-id*s in *exception-specification*s? In the absence of a proposal for change, this box will be removed.

exception-specification:
  `throw ( `*type-id-list*$_{opt}$` )`

type-id-list:
  type-id
  type-id-list `,` *type-id*

An *exception-specification* shall appear only on a function declarator in a declaration or definition. An *exception-specification* shall not appear in a typedef declaration. For example:

```
void f() throw(int);              // OK
void (*fp) throw (int);           // OK
void g(void pfa() throw(int));    // OK
typedef int (*pf)() throw(int);   // ill-formed
```

2 If any declaration of a function has an *exception-specification*, all declarations, including the definition, of that function shall have an *exception-specification* with the same set of *type-id*s. If a virtual function has an *exception-specification*, all declarations, including the definition, of any function that overrides that virtual function in any derived class shall have an *exception-specification* at least as restrictive as that in the base class. 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`. Similarly, any function or pointer to function assigned to, or initializing, a pointer to function shall have an *exception-specification* at least as restrictive as that of the pointer or function being assigned to or initialized. For example:

```
void (*pf1)();    // no exception specification
void (*pf2) throw(A);

void f()
{
        pf1 = pf2;  // ok: pf1 is less restrictive
        pf2 = pf1;  // error: pf2 is more restrictive
}
```

3 In such an assignment or initialization, *exception-specification*s on return types and parameter types shall match exactly.

Box 72
This is needlessly restrictive. We can safely relax this restriction if needed.

4 In other assignments or initializations, *exception-specification*s shall match exactly.

Box 73
This is needlessly restrictive. We can safely relax this restriction if needed.

5 Calling a function through a declaration whose *exception-specification* is less restrictive that that of the function's definition is ill-formed. No diagnostic is required.

6 Types shall not be defined in *exception-specification*s.

7 An *exception-specification* can include the same class more than once and can include classes related by inheritance, even though doing so is redundant. An *exception-specification* can include classes with ambiguous base classes, even though throwing objects of such classes is ill-formed (15.1). An exception specification can include identifiers that represent incomplete types. An exception can also include the name of the predefined class `Xunexpected`.

Box 74
The name `Xunexpected` is under discussion and will change. The exact meaning of ''predefined'' and a possible standard library specification of class `Xunexpected` is also being defined.

8 If a class `X` is in the *type-id-list* of the *exception-specification* of a function, that function is said to *allow* exception objects of class `X` or any class publicly 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*`.

9 Whenever an exception is thrown and the search for a handler (15.3) encounters the outermost block of a function with an *exception-specification*, the function `unexpected()` is called (15.5.2) if the *exception-specification* does not allow the exception. For example,

```
class X { };
class Y { };
class Z: public X { };
class W { };

void f() throw (X, Y)
{
    int n = 0;
    if (n) throw X();         // OK
    if (n) throw Z();         // also OK
    throw W();                // will call unexpected()
}
```

10 The function `unexpected()` may throw an exception that will satisfy the *exception-specification* for which it was invoked, and in this case the search for another handler will continue at the call of the function with this *exception-specification* (see 15.5.2), or it may call terminate.

11 An implementation shall not reject an expression merely because when executed it throws or might throw an exception that the containing function does not allow. 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.

12 A function with no *exception-specification* allows all exceptions. A function with an empty *exception-specification*, `throw()`, does not allow any exceptions.

13 An *exception-specification* is not considered part of a function's type.

15.5 Special functions [except.special]

1 The exception handling mechanism relies on two functions, `terminate()` and `unexpected()`, for coping with errors related to the exception handling mechanism itself (18.6).

15.5.1 The `terminate()` function [except.terminate]

1 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,[77)]

— when the exception handling mechanism cannot find a handler for a thrown exception (see 15.3),

— 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()`(_lib.exception.terminate_).

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()`(_lib.exception.unexpected_).

2 The `unexpected()` function shall not return, but it can throw (or re-throw) an exception. If it throws a new exception which is allowed by the exception specification which previously was violated, then the search for another handler will continue at the call of the function whose exception specification was violated. If it throws or rethrows an exception an exception which is not allowed by the *exception-specification* then the following happens: if the *exception-specification* does not include the name of the predefined exception `Xunexpected` then the function `terminate()` is called, otherwise the thrown exception is replaced by an implementation-defined object of the type `Xunexpected` and the search for another handler will continue at the call of the function whose *exception-specification* was violated.

---

77) 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`.

3 Thus, an *exception-specification* guarantees that only the listed exceptions will be thrown. If the *exception-specification* includes the name `Xunexpected` then any exception not on the list may be replaced by `Xunexpected` within the function `unexpected()`.

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 can be thrown if it can be copied and destroyed in the context of the function in which the throw occurs.

16 Preprocessing directives [cpp]

1 A preprocessing directive consists of a sequence of preprocessing tokens that begins with a # preprocessing token that is either the first character in the source file (optionally after white space containing no new-line characters) or that follows white space containing at least one new-line character, and is ended by the next new-line character.[78)]

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*

78) 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;[79] 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.

[79] Because the controlling constant expression is evaluated during translation phase 4, all identifiers either are or are not macro names — there simply are no keywords, enumeration constants, and so on.

4 Prior to evaluation, macro invocations in the list of preprocessing tokens that will become the controlling constant expression are replaced (except for those macro names modified by the `defined` unary operator), just as in normal text. If the token `defined` is generated as a result of this replacement process or use of the `defined` unary operator does not match one of the two specified forms prior to macro replacement, the behavior is undefined. After all replacements due to macro expansion and the `defined` unary operator have been performed, all remaining identifiers are replaced with the pp-number `0`, and then each preprocessing token is converted into a token. The resulting tokens comprise the controlling constant expression which is evaluated according to the rules of 5.19 using arithmetic that has at least the ranges specified in 18.2, except that `int` and `unsigned int` act as if they have the same representation as, respectively, `long` and `unsigned long`. This includes interpreting character 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.[80] 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.[81]

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.

---

[80] 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)
```

[81] 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.[82] 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 75
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.

[82] 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[83)] 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

83) 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.