1

Introduction

1 This abstract introduces the introductory chapter of the C++ Standard. This chapter describes the purpose and organization of the Standard.

1 Introduction

1 This 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.[1] In addition to the facilities provided by C, C++ provides classes, templates, exceptions, inline functions, operator overloading, function name overloading, constant types, references, free store management operators, and function argument checking and type conversion. These extensions to C are summarized in 19.1. The differences between C++ and ISO C[2] are summarized in 19.2. The extensions to C++ since the 1985 edition of this manual are summarized in 19.1.2.

1.1 Overview

1 This manual is organized like this:

1. Introduction	10. Derived Classes
2. Lexical Conventions	11. Member Access Control
3. Basic Concepts	12. Special Member Functions
4. Standard Conversions	13. Overloading
5. Expressions	14. Templates
6. Statements	15. Exception Handling
7. Declarations	16. Preprocessing
8. Declarators	Appendix A: Grammar Summary
9. Classes	Appendix B: Compatibility

1.2 Syntax notation

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

[1] "The C Programming Language" by Brian W. Kernighan and Dennis M. Ritchie, Prentice Hall, 1978 and 1988.
[2] International Standards Organization IS9899-1990.

{ $expression_{opt}$ }

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 context dependent keyword (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.

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

3 Processors shall issue a diagnostic message for programs that are syntactically incorrect.

1.3 The C++ memory model

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 comprised of one or more contiguous sequences of bytes. Each byte (except perhaps registers) has a unique address.

2 The constructs in a C++ program create, refer to, access, and manipulate *objects* in memory. Each object (except bit-fields) occupies one or more contiguous bytes. Objects are created by definitions (3.1) and *new-expressions* (5.3.3). Each object has a *type* determined by the construct that creates it. The type in turn determines the number of bytes that the object occupies and the interpretation of their contents. Objects may contain other objects, called *sub-objects* (9.2, 10). An object that is not a sub-object of any other object is called a *complete object*. The complete object containing an object is called the complete object of the object. (An object may be its own complete object),

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

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

1.4 Definitions of terms

1 In this Standard, "shall" is to be interpreted as a requirement on C++ processors, and "shall not" is to be interpreted as a prohibition.

2 The following terms are used in this document.

- Diagnostic message — a message belonging to an implementation-defined subset of the implementation's message output.

- 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.

- Undefined behavior — behavior, upon use of an erroneous program construct, of erroneous data, or of indeterminately valued objects, 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.

- Argument — an expression in the comma-separated list bounded by the parentheses in a function call expression, a sequence of prepreocessing 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."

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

- Static type — The *static type* of an expression is the type (3.6) resulting from analysis of the program without consideration of execution semantics. It depends only on the form of the program and does not change.

- Dynamic type — The *dynamic type* of an expression is determined by its current value and may change during the execution of a program. If a pointer (8.2.1) 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 (8.2.2) are treated similarly.

3 Other terms are defined at their first appearance, indicated by *italic* type. Terms explicitly defined in this standard are not to be presumed to refer implicitly to similar terms defined elsewhere. Terms not defined in this standard are to be interpreted according to the *American National Dictionary for Information Processing Systems,* Information Processing Systems Technical Report ANSI X3/TR-1-82 (1982).

2

Lexical Conventions

1 This chapter presents the lexical conventions of C++. It lists the phases of translation and describes tokens in a C++ program including comments, identifiers, keywords, and integer, character, floating point, and string literals. Operators are discussed in 5. The C++ grammar based on these token is summarized in 18.

2 Lexical conventions

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 and source files included (16.2) via the preprocessing directive `#include`, less any source lines skipped by any of the conditional inclusion (16.1) preprocessing directives, is called a *translation unit*. Previously translated translation units may be preserved individually or in libraries. The separate translation units of a program communicate (3.3) by (for example) calls to functions whose identifiers have external linkage, manipulation of objects whose identifiers have external linkage, or manipulation of data files. Translation units may be separately translated and then later linked to produce an executable program. (3.3).

2.1 Phases of translation

1 The precedence among the syntax rules of translation is specified by the following phases.[3]

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

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

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

4 Preprocessing directives are executed and macro invocations are expanded. A `#include`

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

preprocessing directive causes the named header or source file to be processed from phase 1 through phase 4, recursively.

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

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

7 White-space characters separating tokens are no longer significant. Each preprocessing token is converted into a token. (See 2.4). The resulting tokens are syntactically and semantically analyzed and translated. The result of this process starting from a single source file is called a *translation unit*.

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

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

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 the table below.

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

2 For example,

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

becomes

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

2.3 Preprocessing tokens

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

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

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

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

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

5 The program fragment `x+++++y` is parsed as `x ++ ++ + y`, which violates a constraint on increment operators, even though the parse `x ++ + ++ y` might yield a correct expression.

2.4 Digraph sequences

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

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

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

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

2.5 Tokens

token:
 identifier
 keyword
 literal
 operator
 punctuator

1 There are five kinds of tokens: identifiers, keywords, literals (which include strings and character and numeric constants), operators, and other separators. Blanks, horizontal and vertical tabs, newlines, formfeeds, and comments (collectively, “white space”), as described below, are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and

literals.

2 If the input stream has been parsed into tokens up to a given character, the next token is taken to be the longest string of characters that could possibly constitute a token.

2.6 Comments

1 The characters `/*` start a comment, which terminates with the characters `*/`. These comments do not nest. The characters `//` start a comment, which terminates at the end of the line on which they occur. 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

identifier:
  nondigit
  identifier nondigit
  identifier digit

nondigit: one of
```
_ a b c d e f g h i j k l m
  n o p q r s t u v w x y z
  A B C D E F G H I J K L M
  N O P Q R S T U V W X Y Z
```

digit: one of
```
0 1 2 3 4 5 6 7 8 9
```

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

2.8 Keywords

1 The following identifiers are reserved for use as keywords, and may not be used otherwise in phases 7 and 8:

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

2 Furthermore, the following alternate representations for certain operators and punctuators (see 2.4) are reserved and may not be used otherwise:

bitand	and	bitor	or	xor	compl
and_eq	or_eq	xor_eq	not	not_eq	

3 In addition, identifiers containing a double underscore (_ _) are reserved for use by C++ implementations and standard libraries and should be avoided by users.

4 The ASCII representation of C++ programs uses the following characters as operators or for punctuation:

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

and the following character combinations are used as operators:

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

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

5 The following character combinations are used as alternative representations for certain operators and punctuators (see 2.4):

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

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

6 In addition, the following tokens are used by the preprocessor:

```
#   ##   %%   %%%%
```

7 Certain implementation-dependent properties, such as the type of a `sizeof` (5.3.2) and the ranges of fundamental types (3.6.1), are defined in the standard header files (16.2)

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

These headers are part of the ANSI 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 ANSI C standard; `<new.h>` is C++ specific.

2.9 Literals

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

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

2.9.1 Integer literals

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

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

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

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

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

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

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

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

1 A character literal is one or more characters enclosed in single quotes, as in `'x'`. Single character literals have type `char`. The value of a single character literal is the numerical value of the character in the machine's character set. Multicharacter literals have type `int`. The value of a multicharacter literal is implementation dependent.

2 Certain nongraphic characters, the single quote `'`, the double quote `"`, the question mark `?`, and the backslash `\`, may be represented according to the following table of 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.

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

4 A character literal immediately preceded by the letter `L`, for example, `L'ab'`, is a wide-character literal. A wide-character literal is of type `wchar_t`. Wide-characters are intended for character sets where a character does not fit into a single byte.

2.9.3 Floating literals

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

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

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

sign: one of
  `+` `-`

digit-sequence:
  digit
  digit-sequence digit

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

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

2.9.4 String literals

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, as in `"..."`. A string has type "array of `char`" and storage class *static* (3.5), 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 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'`).

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

4 A string literal immediately preceded by the letter `L`, for example, `L"asdf"`, is a wide-character string. A wide-character string is of type "array of `wchar_t`." Concatenation of ordinary and wide-character string literals is undefined.

3

Basic Concepts

1 This chapter 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 class*. The mechanisms for starting and terminating a program are discussed. Finally, this chapter presents the fundamental types of the language and lists the ways of constructing derived types from these.

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

3 Basic concepts

1 A name denotes an object, a function, a set of functions, an enumerator, a type, a class member, a template, a value, or a label. A name is introduced into a program by a declaration. A name can be used only within a region of program text called its *scope*. A name has a type, which determines its use. A name used in more than one translation unit may (or may not) refer to the same object, function, type, template, or value in these translation units depending on the linkage (3.3) specified in the translation units.

2 An object is a region of storage (3.7). A named object has a storage class (3.5) that determines its lifetime. The meaning of the values found in an object is determined by the type of the expression used to access it.

3.1 Declarations and definitions

1 A declaration (7) introduces one or more names into a program. A declaration is a definition unless it declares a function without specifying the body (8.3), it contains the `extern` specifier (7.1.1) and no initializer or function body, it is the declaration of a static data member in a class declaration (9.4), it is a class name declaration (9.1), or it is a typedef declaration (7.1.3). The following, for example, are definitions:

```
int a;
extern const c = 1;
int f(int x) { return x+a; }
struct S { int a; int b; };
enum { up, down };
```

whereas these are just declarations:

```
extern int a;
extern const c;
int f(int);
struct S;
typedef int Int;
```

2 There must be exactly one definition of each object, function, class, and enumerator used in a program. If, however, a function is never called and its address is never taken, it need not be defined. Also, if a declared object is unused, or is only used as the operand of `sizeof`, it need not be defined. Similarly, if the name of a class is used only in a way that does not require its definition to be known, it need not be defined.

This needs to be made more precise.

3.2 Scopes

1 There are four kinds of scope: local, function, file, and class.

> *Local*: A name declared in a block (6.3) is local to that block and can be used only in it and in blocks enclosed by it after the point of declaration. Names of parameters for a function are treated as if they were declared in the outermost block of that function. In a function declaration, names of parameters (if supplied) have *function prototype scope*, which terminates at the end of the function declarator.
>
> *Function*: Labels (6.1) can be used anywhere in the function in which they are declared. Only labels have function scope.
>
> *File*: A name declared outside all blocks (6.3) and classes (9) has file scope and can be used in the translation unit in which it is declared after the point of declaration. Names declared with *file* scope are said to be *global*.
>
> *Class*: The name of a class member is local to its class and can be used only in a member function of that class (9.3), after the `.` operator applied to an object of its class (5.2.4) or a class derived from (10) its class, after the `->` operator applied to a pointer to an object of its class (5.2.4) or a class derived from its class, or after the `::` scope resolution operator (5.1) applied to the name of its class or a class derived from its class. A name first declared by a `friend` declaration belongs to either the global scope or a function scope; see 11.4. The name of a class first declared in a return or parameter type belongs to the global scope.

Special rules apply to names declared in function parameter declarations (8.2.5), and friend declarations (11.4).

2 A name may be hidden by an explicit declaration of that same name in an enclosed block or in a class. A hidden class member name can still be used when it is qualified by its class name using the `::` operator (5.1, 9.4, 10). A hidden name of an object, function, type, or enumerator with file scope can still be used when it is qualified by the unary `::` operator (5.1). In addition, a class name (9.1) may be hidden by the name of an object, function, or enumerator declared in the same scope. If a class and an object, function, or enumerator are declared in the same scope (in any order) with the same name the class name is hidden. A class name hidden by a name of an object, function, or enumerator in local or class scope can still be used when appropriately (7.1.6) prefixed with `class`, `struct`, or `union`, or when followed by the `::` operator. Similarly, a hidden enumeration name can be used when appropriately (7.1.6) prefixed with ‘enum.’ The scope rules are summarized in 10.4.

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

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

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

4 The point of declaration for an enumerator is immediately after the identifier that names it. For example,

```
enum { x = x };
```

Here, again, the enumerator x is initialized to its own (uninitialized) value.

5 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.

3.3 Program and linkage

1 A program consists of one or more files (2) linked together. A file consists of a sequence of declarations.

2 A name of file scope that is explicitly declared `static` has *internal* linkage. Such names are local to their translation units and can be used as names for other objects, functions, and so on, in other translation units. A name of file scope that is explicitly declared `inline` has internal linkage. A name of file scope that is explicitly declared `const` and not explicitly declared `extern` has internal linkage. So does the name of a class that has not been used in the declaration of an object, function, or class that has external linkage and has no static members (9.4) and no noninline member functions (9.3.2). Every declaration of a particular name of file scope that is not declared to have internal linkage in one of these ways in a multifile program refers to the same object (3.7), function (8.2.5), or class (9). Such names are said to have *external* linkage. In particular, since it is not possible to declare a class name `static`, every use of a particular file scope class name that has been used in the declaration of an object or function with external linkage or has a static member or a noninline member function refers to the same class.

3 Typedef names (7.1.3), enumerators (7.2), and template names (14) do not have external linkage.

4 Static class members (9.4) have external linkage.

5 Noninline class member functions have external linkage. Inline class member functions must have exactly one definition in a program.

6 Local names (3.2) explicitly declared `extern` have external linkage unless already declared `static` (7.1.1).

7 The types specified in all declarations of a particular external name must be identical except for the use of typedef names (7.1.3) and unspecified array bounds (8.2.4).

8 A function may be defined only in file or class scope.

9 Linkage to non-C++ declarations can be achieved using a *linkage-specification* (7.4).

3.4 Start and termination

1 A program must contain a function called `main`. This function is the designated start of the program. This function is not predefined by the compiler, it cannot be overloaded, and its type is implementation dependent. The two examples below are allowed on any implementation. It is recommended that any further (optional) parameters be added after `argv`. The function `main()` may be defined as

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

or

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

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

2 The function `main()` may not be called from within a program. The linkage (3.3) of `main()` is implementation dependent. The address of `main()` cannot be taken and `main()` may not be declared `inline` or `static`.

3 Calling the function

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

declared in <stdlib.h> 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.

4 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.

5 The initialization of nonlocal static objects (3.5) in a translation unit is done before the first use of any function or object defined in that translation unit. Such initializations (8.4, 9.4, 12.1, 12.6.1) may be done before the first statement of main() or deferred to any point in time before the first use of a function or object defined in that translation unit. The default initialization of all static objects to zero (8.4) is performed before any dynamic (that is, run-time) initialization. No further order is imposed on the initialization of objects from different translation units. The initialization of local static objects is described in 6.7.

6 Destructors (12.4) for initialized static objects are called when returning from main() and when calling exit(). Destruction is done in reverse order of initialization. The function atexit() from <stdlib.h> can be used to specify that a function must be called at exit. If atexit() is to be called, objects initialized before an atexit() call may not be destroyed until after the function specified in the atexit() call has been called. 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.

7 Calling the function

```
void abort();
```

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

3.5 Storage classes

1 There are two declarable storage classes: automatic and static.

Automatic objects are associated with an invocation of a block.

Static objects exist and retain their values throughout the execution of the entire program.

2 Named automatic objects are initialized (12.1) each time the control flow reaches their definition and destroyed (12.4) on exit from their block (6.6).

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

4 Similarly, a defined global object of a class with a constructor or a destructor with side effects may not be eliminated even if it appears to be unused.

5 Static objects are initialized and destroyed as described in 3.4 and 6.7. Some objects are not associated with names; see 5.3.3 and 12.2. All global objects have storage class *static*. Local objects and class members can be given static storage class by explicit use of the static storage class specifier (7.1.1).

3.6 Types

1 There are two kinds of types: fundamental types and derived types. Types may describe objects, arrays, references, or functions.

2 Arrays of unknown size and classes which have been declared but not defined are called *incomplete* types because the size of an instance of the type is unknown. Also, the void type represents an empty set of values; it is an incomplete type that cannot be completed.

3 A class type may be incomplete at one point in a compilation unit and complete later on. Also, the type of an array may be incomplete at one point in a compilation unit and complete later on. However, the type of a pointer to array of unknown size cannot be completed.

4 Variables that have incomplete type may not be used in some contexts, for example:

```
class X;              // X us 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

void bar()
{
    xp++;    // okay:  X is complete
    arrp++;  // ill-formed:  UNKA can't be completed
}
```

3.6.1 Fundamental types

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

2 Objects declared as characters (`char`) are large enough to store any member of the implementation's basic character set. If a character from this set is stored in a character variable, its value is equivalent to the integer code of that character. Characters may be explicitly declared `unsigned` or `signed.` Plain `char`, `signed char`, and `unsigned char` are three distinct types. A `char`, a `signed char`, and an `unsigned char` consume the same amount of space.

3 An *enumeration* comprises a set of named integer constant values. Each distinct enumeration constitutes a different *enumerated type*.

4 Up to three sizes of integer, declared `short int`, `int`, and `long int`, are available. Longer integers provide no less storage than shorter ones, but the implementation may make either short integers or long integers, or both, equivalent to plain integers. Plain integers have the natural size suggested by the machine architecture; the other sizes are provided to meet special needs.

5 Type `wchar_t` is a type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locales (17.4.4). Type `wchar_t` has the same size, signedness, and alignment requirements as one of the other integral types, called its *underlying type*.

6 For each of the types `signed char`, `short`, `int`, and `long`, there exists a corresponding (but different) `unsigned` type, which occupies the same amount of storage and has the same alignment requirements. An *alignment requirement* is an implementation-dependent restriction on the value of a pointer to an object of a given type (5.4).

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

8 There are three *floating* types: `float`, `double`, and `long double`. The type `double` provides no less precision than `float`, and the type `long double` provides no less precision than `double`. An implementation will define the characteristics of the fundamental floating point types in the standard header `<float.h>`.

9 Types `char`, and the signed and unsigned integer types are collectively called *integral* types. Enumerations (7.2) are not integral, but they can be promoted (4.1) to signed or unsigned `ints`. *Integral* and *floating* types are collectively called *arithmetic* types.

10 The `void` type specifies an empty set of values. It is used as the return type for functions that do not return a value. No object of type `void` may be declared. Any expression may be explicitly converted to type `void` (5.4); the resulting expression may be used only as an expression statement (6.2), as the left operand of a comma expression (5.18), or as a second or third operand of `?:` (5.16).

3.6.2 Derived types

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

arrays of objects of a given type, 8.2.4;

functions, which have parameters of given types and return objects of a given type, 8.2.5;

pointers to objects or functions (including static members of classes) of a given type, 8.2.1;

references to objects or functions of a given type, 8.2.2;

constants, which are values of a given type, 7.1.6;

classes containing a sequence of objects of various types (9), a set of functions for manipulating these objects (9.3), and a set of restrictions on the access to these objects and functions, 11;

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

unions, which are classes capable of containing objects of different types at different times, 9.5;

pointers to non-static[4] *class members*, which identify members of a given type within objects of a given class, 8.2.3.

2 In general, these methods of constructing types can be applied recursively; restrictions are mentioned in 8.2.1, 8.2.4, 8.2.5, and 8.2.2.

3 Any type so far mentioned is an *unqualified type*. Each unqualified type has three corresponding *qualified versions* of its type:[5] a *const-qualified* version, a *volatile-qualified* version, and a version having both qualifications (see 7.1.6). The qualified or unqualified versions of a type are distinct types that belong to the same type category and have the same representation and alignment requirements.[6] A derived type is not cv-qualified (3.6.3) by the cv-qualifiers (if any) of the type from which it is derived.

4 A pointer to objects of a type `T` is referred to as a "pointer to `T`." For example, a pointer to an object of type `int` is referred to as "pointer to `int`" and a pointer to an object of class `X` is called a "pointer to `X`." Pointers to incomplete types are allowed although there are restrictions on what can be done with them (see 3.6).

5 Objects of cv-qualified (3.6.3) or unqualified type `void*` (pointer to void), can be used to point to objects of unknown type. A `void*` must have enough bits to hold any object pointer.

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

3.6.3 CV-qualifiers

1 There are two *cv-qualifier*s, `const` and `volatile`. When applied to an object, `const` means the program may not change the object, and `volatile` has an implementation-defined meaning.[7] An object may

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

[5] See _xxx.yyy_ regarding qualified array and function types.

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

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

have both cv-qualifiers.

2 There is a (non-total) ordering on cv-qualifiers, so that one object or pointer may be said to be more cv-qualified than another. The following relations constitute this ordering.

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

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

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

3.6.4 Type names

1 Fundamental and derived 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).

3.7 Lvalues

1 An *object* is a region of storage; an *lvalue* is an expression referring to an object or function. An obvious example of an lvalue expression is the name of an object. Some operators yield lvalues. For example, if `E` is an expression of pointer type, then `*E` is an lvalue expression referring to the object to which `E` points. The name "lvalue" comes from the assignment expression `E1 = E2` in which the left operand `E1` must be an lvalue expression. The discussion of each operator in 5 indicates whether it expects lvalue operands and whether it yields an lvalue.

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

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

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

In C this is undefined.

For example:

[8] An lvalue that refers to an array object is usually converted to a (non-lvalue) pointer to the first element of the array; see 4.6.

```
struct  X;
X*   xp;
xp;         // okay: pointer to incomplete type
*xp;        // error: incomplete type
```

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

3 An lvalue can also be used to modify its referent under certain circumstances. Functions cannot be modified, but pointers to functions may be modifiable. Objects of incomplete type cannot be modified, but a pointer to such an object may be modifiable and the object itself may be modifiable at some point in the program where its type is complete. Array objects cannot be modified, but their elements may be modifiable. The referent of a `const`-qualified lvalue cannot be modified (through that lvalue). Class or union objects cannot be modified if any of their elements is a reference or is `const` or cannot be modified for any of the foregoing reasons. If an lvalue can be used to modify its object, it is called a *modifiable lvalue*. A program that attempts to modify a nonmodifiable lvalue is ill-formed.

4

Standard Conversions

1 This chapter presents standard type conversions, including integral promotions, integral conversions, floating point conversions, conversions between floating and integral types, and arithmetic conversions, as well as pointer, reference, and pointer to member conversions.

4 Standard conversions

1 Some operators may, depending on their operands, cause conversion of the value of an operand from one type to another. This section summarizes the conversions demanded by most ordinary operators and explains the result to be expected from such conversions; it will be supplemented as required by the discussion of each operator. These conversions are also used in initialization (8.4, 8.4.3, 12.8, 12.1). 12.3 and 13.2 describe user-defined conversions and their interaction with standard conversions. The result of a conversion is an lvalue only if the result is a reference (8.2.2).

4.1 Integral promotions

1 A `char`, `wchar_t`, a `short int`, enumerator, object of enumeration type (7.2), or an `int` bit-field (9.6) (in both their signed and unsigned varieties) may be used wherever an integer rvalue may be used. In contexts where a constant integer is required, the `char`, `wchar_t`, `short int`, object of enumeration type (7.2), or bit-field must be constant. (Enumerators are always constant). Except for enumerators, objects of enumeration type, and type `wchar_t`, if an `int` can represent all the values of the original type, the value is converted to `int`; otherwise it is converted to `unsigned int`. For enumerators, objects of enumeration type, and type `wchar_t`, if an `int` can represent all the values of the underlying type, the value is converted to an `int`; otherwise if an `unsigned int` can represent all the values, the value is converted to an `unsigned int`; otherwise, if a `long` can represent all the values, the value is converted to a `long`; otherwise it is converted to `unsigned long`. This process is called *integral promotion*.

4.2 Integral conversions

1 An integer rvalue may be converted to any integral type. If the target type is *unsigned*, the resulting value is the least unsigned integer congruent to the source integer (modulo 2^n where n is the number of bits used to represent the unsigned type). In a two's complement representation, this conversion is conceptual and there is no change in the bit pattern.

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

4.3 Float and double

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

4.4 Floating and integral

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

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

4.5 Arithmetic conversions

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

2

> If either operand is of type `long double`, the other is converted to `long double`.
>
> Otherwise, if either operand is `double`, the other is converted to `double`.
>
> Otherwise, if either operand is `float`, the other is converted to `float`.
>
> Otherwise, the integral promotions (4.1) are performed on both operands.
>
> Then, if either operand is `unsigned long` the other is converted to `unsigned long`.
>
> Otherwise, if one operand is a `long int` and the other `unsigned int`, then if a `long int` can represent all the values of an `unsigned int`, the `unsigned int` is converted to a `long int`; otherwise both operands are converted to `unsigned long int`.
>
> Otherwise, if either operand is `long`, the other is converted to `long`.
>
> Otherwise, if either operand is `unsigned`, the other is converted to `unsigned`.
>
> Otherwise, both operands are `int`.

4.6 Pointer conversions

1 The following conversions may be performed wherever pointers (8.2.1) are assigned, initialized, compared, or otherwise used:

> A constant expression (5.19) that evaluates to zero (the null pointer constant) when assigned to, compared with, alternated with (5.16), or used as an initializer of an operand of pointer type is converted to a pointer of that type. It is guaranteed that this value will produce a pointer distinguishable from a pointer to any object or function.
>
> A pointer to a cv-qualified or unqualified object type may be converted to a pointer to the same type with greater cv-qualifications (3.6.3). That is, for any unqualified type `T`, a `T*` may be converted to a `const T*`, a `volatile T*`, or a `const volatile T*`; a `const T*` may be converted to a

`const volatile T*`; or a `volatile T*` may be converted to a `const volatile T*`.

A pointer to any object type may be converted to a `void*` with the greater or equal cv-qualifications. That is, for any unqualified type `T`. a `T*` may be converted to a `void*`, a `const void*`, a `volatile void*`, or a `const volatile void*`; a `const T*` may be converted to a `const void*` or a `const volatile void*`; a `volatile T*` may be converted to a `volatile void*` or a `const volatile void*`; and a `const volatile T*` may be converted to a `const volatile void*`.

For any unqualified type `T`, a `T**` may be converted to a `const T *const *`, and similarly for more levels of indirection, e.g, a `T***` may be converted to a `const T *const *const *`, and a `T****` may be converted to a `const T *const *const *const *`, etc. This rule may be applied using `volatile` in place of `const`.

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

A pointer to a class may be converted to a pointer to an accessible[9] base class of that class (10) provided the conversion is unambiguous (10.1); a base class is accessible if its public members are accessible (11.1). The result of the conversion is a pointer to the base class sub-object of the derived class object. The null pointer (`0`) is converted into itself.

An expression with type "array of `T`" may be converted to a pointer to the initial element of the array (5).

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

4.7 Reference conversions

1 The following conversion may be performed wherever references (8.2.2) are initialized (including argument passing (5.2.2) and function value return (6.6.3)):

A reference to a cv-qualified or unqualified object type may be converted to a reference to the same type with increased cv-qualifications.

A reference to a class may be converted to a reference to an accessible base class (10, 11.1) of that class (8.4.3) provided this conversion can be done unambiguously (10.1.1). The result of the conversion is a reference to the base class sub-object of the derived class object.

4.8 Pointers to members

1 The following conversion may be performed wherever pointers to members (8.2.3) are initialized, assigned, compared, or otherwise used:

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

A pointer to a member of a class may be converted to a pointer to member of a class derived from that class provided the (inverse) conversion from the derived class to the base class pointer is

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

accessible (11.1) and provided this conversion can be done unambiguously (10.1.1).

2 The rule for conversion of pointers to members (from pointer to member of base to pointer to member of derived) appears inverted compared to the rule for pointers to objects (from pointer to derived to pointer to base) (4.6, 10). This inversion is necessary to ensure type safety.

3 Note that a pointer to member is not a pointer to object or a pointer to function and the rules for conversions of such pointers do not apply to pointers to members. In particular, a pointer to member cannot be converted to a `void*`.

5

Expressions

1 This chapter discusses C++ expressions, the primary building blocks for computation. C++ provides the usual arithmetic operators (`+`, `-`, `*`, and so on), bit manipulation operators (`&`, `|`, `^`, and so on), operators for pointer manipulation (`*`, `&`, `[]`, `->`), storage management (`new` and `delete`), conditional evaluation (`?:`, `||`, `&&`), and the pointer to member operators (`.*` and `->*`).

2 This chapter also describes explicit type conversions (“casting”).

5 Expressions

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

2 Operators can be overloaded, that is, given meaning when applied to expressions of class type (9). Uses of overloaded operators are transformed into function calls as described in 13.4. Overloaded operators obey the rules for syntax specified in this section, 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).

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

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

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

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

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

7 If an expression has the type "reference to `T`" (8.2.2, 8.4.3), the value of the expression is the object of type "`T`" denoted by the reference. The expression is an lvalue. A reference can be thought of as a name of an object.

8 User-defined conversions of class objects to and from fundamental types, pointers, and so on, can be defined (12.3). If unambiguous (13.2), such conversions may be applied by the compiler wherever a class object appears as an operand of an operator, as an initializer (8.4), as the controlling expression in a selection (6.4) or iteration (6.5) statement, as a function return value (6.6.3), or as a function argument (5.2.2).

5.1 Primary expressions

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

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

4 The operator `::` followed by an *identifier*, a *qualified-id*, or an *operator-function-id* is a primary expression. Its type is specified by the declaration of the identifier, name, or *operator-function-id*. The result is the identifier, name, or *operator-function-id*. The result is an lvalue if the identifier is. The identifier or *operator-function-id* must be of file scope. Use of `::` allows a type, an object, a function, or an enumerator to be referred to even if its identifier has been hidden (3.2).

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:
  identifier
  operator-function-id
  conversion-function-id
  ~ *class-name*
  qualified-id

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-class-specifier `::` *id-expression*

8 A *nested-class-specifier* (9.1) followed by `::` and the name of a member of that class (9.2), or a member of a base of that class (10), is a *qualified-id*; its type is the type of the member. The result is the member. The result is an lvalue if the member is. The *class-name* may be hidden by a nontype name, in which case the *class-name* is still found and used. Where *class-name* `::` *class-name* or *class-name* `::` ~ *class-name* is used, the two *class-name*s must refer to the same class; this notation names constructors (12.1) and destructors (12.4), respectively. Multiply qualified names, such as `N1::N2::N3::n`, can be used to refer to nested types (9.7).

5.2 Postfix expressions

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-name* `> (` *expression* `)`
 `typeid (` *expression* `)`
 `typeid (` *type-name* `)`

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

5.2.1 Subscripting

1 A postfix expression followed by an expression in square brackets is a postfix expression. The intuitive meaning is that of a subscript. One of the expressions must have the type "pointer to `T`" and the other must be of enumeration or integral type. The type of the result is "`T`." The type "`T`" must be complete. The expression `E1[E2]` is identical (by definition) to `*((E1)+(E2))`. See 5.3 and 5.7 for details of `*` and `+` and 8.2.4 for details of arrays.

5.2.2 Function call

1 There are two kinds of function call: ordinary function call and member function[10] (9.3) call. A function call is a postfix expression followed by parentheses containing a possibly empty, comma-separated list of expressions which constitute the arguments to the function. For ordinary function call, the postfix expression must be a function name, or a pointer or reference to function. For member function call, the postfix expression must be a class member access (5.2.4) whose *id-expression* is a function member name, or a pointer-to-member expression (5.5) selecting a function member. The first expression in the postfix expression is then called the *object expression*, and the call is as a member of the object pointed to or referred to. If a function or member function name is used, the name may be overloaded (13), in which case the appropriate function will be selected according to the rules in 13.2. The function called in a member function call is normally selected according to the static type of the object expression (see 10), but if that function is `virtual` the function actually called will be the final overrider (10.2) of the selected function in the dynamic type of the object expression (i.e., the type of the object pointed or referred to by the current value of the object expression).

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

3 When a function is called, each parameter (8.2.5) is initialized (8.4.3, 12.8, 12.1) with its corresponding argument. Standard (4) and user-defined (12.3) conversions are performed. The value of a function call is the value returned by the called function except in a virtual function call if the return type of the final overrider is different from the return type of the statically chosen function, the value returned from the final overrider is converted to the return type of the statically chosen function. A function may change the values of its nonconstant parameters, but these changes cannot affect the values of the arguments except where a parameter is of a non-`const` reference type (8.2.2). Where a parameter is of reference type a temporary variable is introduced if needed (7.1.6, 2.9, 2.9.4, 8.2.4, 12.2). In addition, it is possible to modify the

[10] A static member function (9.4) is an ordinary function.

values of nonconstant objects through pointer parameters.

4 A function may be declared to accept fewer arguments (by declaring default parameters 8.2.6) or more arguments (by using the ellipsis, `...` 8.2.5) than the number of parameters in the function definition (8.3).

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

6 Any argument of type `float` for which there is no parameter is converted to `double` before the call; any of `char`, `short`, enumeration, or a bit-field type for which there is no parameter are converted to `int` or `unsigned` by integral promotion (4.1). An object of a class for which no parameter is declared is passed as a data structure.

What does it mean to pass a parameter as a data structure?

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

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

9 Recursive calls are permitted.

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

5.2.3 Explicit type conversion (functional notation)

1 A *simple-type-specifier* (7.1.6) followed by a parenthesized *expression-list* constructs a value of the specified type given the expression list. If the expression list specifies a single value, the expression is equivalent (in definedness, and if defined in meaning) to the corresponding cast expression (5.4). If the expression list specifies more than a single value, the type must be a class with a suitably declared constructor (8.4, 12.1).

2 A *simple-type-specifier* (7.1.6) followed by a (empty) pair of parentheses constructs a value of the specified type. If the type is a class, the class must have a default constructor (12.1) (otherwise the expression is erroneous) and that constructor will be called; otherwise (the type is not a class) the result is an unspecified value of the specified type. See also (5.4).

5.2.4 Class member access

1 A postfix expression followed by a dot (`.`) or an arrow (`->`) followed by an *id-expression* is a postfix expression. For the first option (dot) the type of the first expression (the *object expression*) must be "class object" (of a complete type). For the second option (arrow) the type of the first expression (the *pointer expression*) must be "pointer to class object" (of a complete type). The *id-expression* must name a member of that class, except that an imputed destructor may be explicitly invoked for a built-in type, see 12.4. If the *id-expression* is a *qualified-id*, the *nested-class-specifier* of the *qualified-id* is looked up as a type both in the class of the object expression (or the class pointed to by the pointer expression) and the context in which the entire *postfix-expression* occurs. If the *nested-class-specifier* contains a *template-class-id* (14.2), its *template-argument*s are evaluated in the context in which the entire *postfix-expression* occurs. For the purpose of this type lookup, the name, if any, of each class is also considered a nested class member of that class. These searches must yield a single type which may be found in either or both contexts. If the *id-expression* names a nonstatic data member, the result is the named member of the object designated by the value of the first expression, and it is an lvalue if the class object and the member are lvalues. If the *id-expression* names a static data member, the result is the named member of the class. If the *id-expression* names a (possibly overloaded) nonstatic function member, the expression can only be used as part of a member function call (5.2.2). If the *id-expression* names a (possibly overloaded) static function member, the result is the function.

2 Thus if `E1` is a pointer to a class object, the expression `E1->MOS` is the same as `(*E1).MOS`.

3 Note that "class objects" can be structures (9.2) and unions (9.5). Classes are discussed in 9.

5.2.5 Increment and decrement

1 The value obtained by applying a postfix ++ is the value of the operand. The operand must be a modifiable lvalue. The type of the operand must be an arithmetic type or a pointer to object type. After the result is noted, the object is incremented by `1`. 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.

5.2.6 Dynamic cast

1 The result of the expression `dynamic_cast<T>(v)` is of type `T`, which must be a pointer or a reference to a complete class or `void*`. The type of `v` must be a complete pointer type if `T` is a pointer, or a complete reference type if `T` is a reference.

2 If `T` is a pointer to class `B` and `v` is a pointer to class `D` such that `B` is an unambiguous accessible direct or indirect base class of `D`, the result is a pointer to the unique `B` sub-object of the `D` object pointed to by `v`. Similarly, if `T` is a reference to class `B` and `v` is a reference to class `D` such that `B` is an unambiguous accessible direct or indirect base class of `D`, the result is a reference to the unique[11] `B` sub-object of the `D` object referred to by `v`. For example,

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

Otherwise `v` must be a pointer or reference to a polymorphic type (10.2).

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

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

5 The value of a failed cast to pointer type is the null pointer. A failed cast to reference type throws `Bad_cast` (17.1.3.3.3). 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
}
```

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

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

5.2.7 Type identification

1 The result of a `typeid` expression is of type `const Type_info&` (17.1.2). The value is a reference to a `Type_info` object that represents the *type-name* or the type of the *expression* respectively.

2 If the *expression* is a reference to a polymorphic type (10.2) the `Type-info` for the complete object (12.6.2) referred to is the result. Where the *expression* is a pointer to a polymorphic type dereferenced using `*` or `[`*expression*`]` the `Type-info` for the complete object pointed to is the result. Otherwise, the result is the `Type-info` representing the (static) type of the *expression*.

5.3 Unary operators

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
  `*  &  +  -  !  ~`

The unary `*` operator means *indirection*: the expression must be a pointer, and the result is an lvalue referring to the object to which the expression points. If the type of the expression is "pointer to `T`," the type of the result is "`T`."

2 The result of the unary `&` operator is a pointer to its operand. The operand must be a function, an lvalue, or a *qualified-id*. In the first two cases, if the type of the expression is "`T`," the type of the result is "pointer to `T`." In particular, the address of an object of type "cv-qualified `T`" is "pointer to cv-qualified `T`," with the same cv-qualifiers. For example, the address of an object of type `const T` has type `const T*`; `volatile` is handled similarly. For a *qualified-id*, if the member is not static and of type "`T`" in class "`C`," the type of the result is "pointer to member of `C` of type `T`." For a static member of type `T`, the type is plain "pointer to `T`."

3 The address of an object of incomplete type may be taken, but only if the complete type of that object does not have the address-of operator (`operator&()`) overloaded.

This is (probably) an example of an error of form that need not be diagnosed.

4 The address of an overloaded function (13) can be taken only in a context that uniquely determines which version of the overloaded function is referred to (see 13.3).

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

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

7 The operand of the logical negation operator `!` must have arithmetic type or be a pointer or a pointer to member; its value is `1` if the value of its operand is zero (for arithmetic types) or null (for pointer and pointer to member types), and zero otherwise. The type of the result is `int`.

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

5.3.1 Increment and decrement

1 The operand of prefix ++ is incremented by `1`. The operand must be a modifiable lvalue. The type of the operand must be an arithmetic type or a pointer to object type. The value is the new value of the operand; it is an lvalue. 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.

5.3.2 Sizeof

1 The `sizeof` operator yields the size, in bytes, of its operand. The operand is either an expression, which is not evaluated, or a parenthesized type name. The `sizeof` operator may not be applied to a function, a bit-field, an undefined class, the type `void`, or an array with an unspecified dimension. A *byte* is unspecified by the language except in terms of the value of `sizeof`; `sizeof(char)` is `1`.

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

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

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

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

5.3.3 New

1 The *new-expression* attempts to create an object of the *type-id* (8.1) to which it is applied. This type must be a complete object type.

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

direct-new-declarator:
  direct-new-declarator$_{opt}$ `[` *expression* `]`

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

The lifetime of an object created by a *new-expression* is not restricted to the scope in which it is created. The *new-expression* returns a pointer to the object created. When that object is an array (that is, the *new-declarator*$_{opt}$[*expression*] syntax is used), a pointer to its initial element (if any) is returned. For example, both `new int` and `new int[10]` return an `int*` and the type of `new int[i][10]` is `int (*)[10]`. Where an array type (8.2.4) is specified all array dimensions but the first must be constant integral expressions (5.19) with positive values. The first array dimension can be a general integral *expression* even when the *type-id* is used (despite the general restriction of array dimensions in *type-id*s to *constant-expression*s (5.19)). If the value of the first array dimension is negative the result is undefined.

2 When the value of the first array dimension 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.

3 The *type-specifier-seq* may not contain `const`, `volatile`, class declarations, or enumeration declarations.

4 Storage for the object created by a *new-expression* is obtained from the appropriate *allocation function* (12.5) (`operator new()` for non-arrays or `operator new[]()` for arrays). When the allocation function is called, the first argument will be amount of space requested (which may be larger than the size of the object being created only if that object is an array). The *new-placement* syntax can be used to supply additional arguments. For example, `new T` results in a call of `operator new(sizeof(T))`, `new(2,f) T` results in a call of `operator new(sizeof(T),2,f)`, `new T[5]` results in a call of `operator new[](x)`, and `new(2,f) T[5]` results in a call of `operator new[](y,2,f)`, where `x` and `y` are greater than or equal to `sizeof(T[5])`.

5 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). The allocation function may indicate failure by throwing an `xalloc` exception (15, 17.1.3.3.2). In this case no initialization is done.

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

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

8 No initializers can be specified for arrays. Arrays of objects of a class with constructors can be created by a *new-expression* only if the class has a default constructor. In that case, the default constructor will be called for each element of the array.

9 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.

10 In a *new-type-id* used as the operand for `new`, parentheses may not be used. This implies that

```
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 derived types. For example,

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

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

11 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.

5.3.4 Delete

1 The *delete-expression* operator destroys a complete object (1.3) 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` must 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 representing a base class of such an object. In the second alternative (*delete array*), the value of the operand of `delete` must be a pointer to an array created by a *new-expression* without a *new-placement* specification. Otherwise, the result is undefined.

3 In the first alternative (*delete object*), if the static type of the operand is different from its dynamic type and the class of the complete object has a destructor (12.4), the static type must have a virtual destructor or the result is undefined. In the second alternative (*delete array*) if the dynamic type of the object to be deleted is a class that has a destructor and its static type is different from its dynamic type, the result is undefined.

4 The effect of attempting to access a deleted object is undefined and the deletion of an object may change its value. Furthermore, 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.

5 A program that applies `delete` to a pointer to constant is ill formed.

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

7 The *delete-expression* will invoke the destructor (if any) for the object or the elements of the array being deleted.

8 To free the storage pointed to, the *delete-expression* will call a *deallocation function* (`operator delete()` for non-arrays or `operator delete[]()` for arrays); see 12.5.

5.4 Explicit type conversion (cast notation)

1 An explicit type conversion can be expressed using either functional notation (5.2.3) or the *cast* notation.

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

The *cast* notation is needed to express conversion to a type that does not have a *simple-type-specifier*.

2 Types may not be defined in casts.

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

4 Any type that can be converted to another by a standard conversion (4) can also be converted by explicit conversion and the meaning is the same.

5 A value of integral type may be explicitly converted to an enumeration type. If the integral value is not equal to the value of one of the enumerators, the result is undefined.

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

7 A value of integral type may 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 dependent.

8 A pointer to one object type may be explicitly converted to a pointer to another object type (subject to the restrictions mentioned in this section). The resulting pointer may be invalid if the subject pointer does not refer to an object suitably aligned in storage. It is guaranteed that a pointer to an object of a given alignment may be converted to a pointer to an object of the same or less strict alignment and back again; the result shall compare equal to the original pointer. (An object that has character type has the least strict alignment). Different machines may differ in the number of bits in pointers and in alignment requirements for objects. Aggregates are aligned on the strictest boundary required by any of their constituents. A `void*` is considered a pointer to object type. A pointer to any object type may be converted to `void*` and back again without change.

9 A pointer to a complete class `B` may be explicitly converted to a pointer to a complete class `D` that has `B` as a direct or indirect base class if an unambiguous conversion from `D` to `B` exists (4.6, 10.1.1) and if `B` is not a virtual base class (10.1). Such a cast from a base to a derived class is only valid if the pointer points to an object of the base class that is actually a sub-object of an object of the derived class; the resulting pointer points to the enclosing object of the derived class. Otherwise (the object of the base class is not a sub-object of an object of the derived class) the result of the cast is undefined.

10 A pointer to an object of a derived class (10) may be explicitly converted to a pointer to one of its base classes regardless of accessibility restrictions (11.2), provided the conversion is unambiguous (10.1.1). The resulting pointer will refer to the contained object of the base class.

11 The null pointer (`0`) is converted into itself.

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

13 An object may be explicitly converted to a reference type `X&` if a pointer to that object may be explicitly converted to an `X*`. Constructors or conversion functions are not called as the result of a cast to a reference. Conversion of a reference to a base class to a reference to a derived class is handled similarly to the conversion of a pointer to a base class to a pointer to a derived class with respect to ambiguity, virtual classes, and so on.

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

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

16 A pointer to a function may be explicitly converted to a pointer to a function of a different type. The effect of calling a function through a pointer to a function type that differs from the type used in the definition of the function is undefined. See also 4.6.

17 An object or a value may be converted to a class object (only) if an appropriate constructor or conversion operator has been declared (12.3).

18 A pointer to member may be explicitly converted into a different pointer to member type when the two types are both pointers to members of the same class or when the two types are pointers to member of classes one of which is unambiguously derived from the other (4.6).

19 A pointer to `const` can be cast into a pointer to non-`const` with otherwise identical type. The resulting pointer will refer to the original object. A `const` object or a reference to `const` can be cast into a reference to non-`const` with otherwise identical type. The resulting reference will refer to the original object. Depending on the type of the referenced object, a write operation through the resulting pointer or reference may be undefined; see 7.1.6.

20 A pointer to `volatile` can be cast into a pointer to a non-`volatile` with otherwise identical type. The resulting pointer will refer to the original object. An object of a `volatile` type or a reference to `volatile` can be cast into a reference to a non-`volatile` with otherwise identical type.

21 Any expression may be explicitly converted to type `void`.

5.5 Pointer-to-member operators

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

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

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

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

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

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

calls the member function denoted by `ptr_to_mfct` for the object pointed to by `ptr_to_obj`. The result of an `.*` expression or a `->*` expression is an lvalue only if its first operand is an lvalue and its second operand refers to an lvalue.

5.6 Multiplicative operators

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

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

2 The operands of `*` and `/` must have arithmetic type; the operands of `%` must have integral type. The usual arithmetic conversions (4.5) are performed on the operands and determine the type of the result.

3 The binary `*` operator indicates multiplication.

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

5.7 Additive operators

1 The additive operators `+` and `-` group left-to-right. The usual arithmetic conversions (4.5) are performed for operands of arithmetic type.

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

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

2 For subtraction, one of the following shall hold:

- both operands have arithmetic type;
- both operands are pointers to qualified or unqualified versions of the same completely defined object type; or
- the left operand is a pointer to a completely defined object type and the right operand has integral type.

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

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

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

6 When two pointers to elements of the same array object are subtracted, the result is the difference of the subscripts of the two array elements. The type of the result is an implementation-defined signed integral type, `ptrdiff_t`, defined in the `<stddef.h>` header.

Is this a typedef for some signed integral type or a different type?

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.[12]

5.8 Shift operators

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

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

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

---

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

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

5.9 Relational operators

1 The relational operators group left-to-right, but this fact is not very useful; `a<b<c` means `(a<b)<c` and *not* `(a<b)&&(b<c)`.

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

The operands must have arithmetic or pointer type. The operators < (less than), > (greater than), <= (less than or equal to), and >= (greater than or equal to) all yield zero if the specified relation is false and `1` if it is true. The type of the result is `int`.

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

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

5.10 Equality operators

1

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

The == (equal to) and the `!=` (not equal to) operators are exactly analogous to the relational operators except for their lower precedence. (Thus `a<b == c<d` is `1` whenever `a<b` and `c<d` have the same truth-value.)

2 In addition, pointers to members of the same type may be compared. Pointer to member conversions (4.8) are performed. A pointer to member may be compared to a constant expression that evaluates to zero.

5.11 Bitwise AND operator

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

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

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

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

The `&&` operator groups left-to-right. The operands need not have the same type, but each must have arithmetic type or be a pointer or pointer to member. It returns `1` if both its operands are nonzero (for arithmetic types) or non-null (for pointer or pointer to member types), zero otherwise. Unlike `&`, `&&` guarantees left-to-right evaluation; moreover the second operand is not evaluated if the first operand evaluates to zero or the null pointer or the null pointer to member.

2 The result is an `int`. All side effects of the first expression happen before the second expression is evaluated.

5.15 Logical OR operator

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

The `||` operator groups left-to-right. The operands need not have the same type, but each must have arithmetic type or be a pointer or a pointer to member. It returns `1` if either of its operands is nonzero or non-null, and zero otherwise. Unlike `|`, `||` guarantees left-to-right evaluation; moreover, the second operand is not evaluated if the first operand evaluates to nonzero or non-null.

2 The result is an `int`. All side effects of the first expression happen before the second expression is evaluated.

5.16 Conditional operator

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

Conditional expressions group right-to-left. The first expression must have arithmetic or pointer or pointer to member type. It is evaluated and if it is nonzero or nonnull, 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 happen before the second or third expression is evaluated.

2 If either the second or third expression is a *throw-expression* (15.2), the result is of the type of the other.

3 If both the second and the third expressions are of arithmetic type, then if they are of the same type the result is of that type; otherwise the usual arithmetic conversions are performed to bring them to a common type. Otherwise, if both the second and the third expressions are either a pointer or a constant expression

that evaluates to zero, pointer conversions (4.6) are performed to bring them to a common type which must be a qualified or unqualified version of the type of either the second or the third expression. Otherwise, if both the second and the third expressions are either a pointer to member or a constant expression that evaluates to zero, pointer to member conversions (4.8) are performed to bring them to a common type[13] which must be a qualified or unqualified version of the type of either the second or the third expression. Otherwise, if both the second and the third expressions are references, reference conversions (4.7) are performed to bring them to a common type which must be a qualified or unqualified version of the type of either the second or the third expression. Otherwise, if both the second and the third expressions are `void`, the common type is `void`. Otherwise, if both the second and the third expressions are of the same class `T`, the common type is `T`. Otherwise the expression is ill formed. The result has the common type; only one of the second and third expressions is evaluated. The result is an lvalue if the second and the third operands are of the same type and both are lvalues.

5.17 Assignment operators

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

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

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

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

3 A pointer of type `T*const` can be assigned to a pointer of type `T*`, but the reverse assignment is not allowed (7.1.6). Objects of types `const T` and `volatile T` can be assigned to plain `T` lvalues and to lvalues of type `volatile T`; see also (8.4).

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

5 Assignment to objects of a class (9) `X` is defined by the function `X::operator=()` (13.4.3). Unless the user defines an `X::operator=()`, the default version is used for assignment (12.8). This implies that an object of a class derived from `X` (directly or indirectly) by unambiguous public derivation (4.6) can be assigned to an `X`.

6 A pointer to a member of class `B` may be assigned to a pointer to a member of class `D` of the same type provided `D` is derived from `B` (directly or indirectly) by unambiguous public derivation (10.1.1).

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

8 The behavior of an expression of the form `E1` *op*= `E2` is equivalent to `E1 = E1` *op* `(E2)`; except that `E1` is evaluated only once. In += and −=, the left operand may be a pointer to completely defined object type, in which case the (integral) right operand is converted as explained in 5.7; all right operands and all nonpointer left operands must have arithmetic type.

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

9 For class objects, assignment is not in general the same as initialization (8.4, 12.1, 12.6, 12.8).

10 See 15.2 for throw expressions.

5.18 Comma operator

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 (8.4), the comma operator as described in this section 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

1 In several places, C++ requires expressions that evaluate to an integral constant: as array bounds (8.2.4), as `case` expressions (6.4.2), as bit-field lengths (9.6), and as enumerator initializers (7.2).

constant-expression:
  conditional-expression

A *constant-expression* can involve only literals (2.9), enumerators, `const` values of integral types initialized with constant expressions (8.4), and `sizeof` expressions. Floating constants (2.9.3) must be cast to integral types. Only type conversions to integral types may be used. In particular, except in `sizeof` expressions, functions, class objects, pointers, and references cannot be used. The comma operator and *assignment-operator*s may not be used in a constant expression.

6

Statements

1 This chapter discusses statements, which control the execution sequence of programs.

2 C++ provides statements for conditional execution (`if` and `switch`) and iteration (`do`, `for`, and `while`). The `break`, `continue`, `return`, and `goto` statements transfer control in a C++ program. Other statements evaluate an expression (the expression statement) or do nothing (the null statement). Statements may be grouped in `{}` pairs to form compound statements.

3 A declaration is a statement in C++; declarations are introduced in this chapter and discussed in detail in the following two chapters.

6 Statements

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

1 A statement may be labeled.

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

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

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

6.2 Expression statement

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

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 (3.2).

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

6.4 Selection statements

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 declarator = *expression*

The *statement* in a *selection-statement* (both statements, in the `else` form of the `if` statement) implicitly defines a local scope (3.2). This can be expressed as a rewriting rule in which the statement is replaced by a compound statement containing the original statement. For example,

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

may be equivalently rewritten as

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

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

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

3 A name introduced by a declaration in a *condition* is in scope from its point of declaration until the end of the statements controlled by the condition. The value of a *condition* that is an initialized declaration is the value of the initialized variable.

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

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

6.4.1 The `if` statement

1 The expression must be of arithmetic or pointer or pointer to member type or of a class type for which an unambiguous conversion to arithmetic or pointer or pointer to member type exists (12.3).

2 The expression is evaluated and if it is nonzero (for arithmetic types) or non-null (for pointer or pointer to member types), the first substatement is executed. If `else` is used, the second substatement is executed if the expression is zero or null. The `else` ambiguity is resolved by connecting an `else` with the last encountered `else`-less `if`.

6.4.2 The `switch` statement

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

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

`case` *constant-expression* `:`

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

3 There may be at most one label of the form

`default :`

within a `switch` statement.

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

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

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

7 Usually, the statement that is the subject of a switch is compound. Declarations may appear in the *statement* of a switch-statement. However, a program that jumps past a declaration with an explicit or implicit initializer is ill formed unless the declaration is in an inner block that is not entered (that is, completely bypassed by the transfer of control; 6.7). This implies that declarations that contain explicit or implicit initializers must be contained in an inner block.

6.5 Iteration statements

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 (3.2) which is entered and exited each time through the loop. This can be expressed as a rewriting rule in which the statement is replaced by a compound statement containing the original statement. For example,

```
while (x)
    for (int i;;) {
        // ...
    }
```

may be equivalently rewritten as

```
while (x) {
    for (int i;;) {
        // ...
    }
}
```

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

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

6.5.1 The `while` statement

1 In the `while` statement the substatement is executed repeatedly until the value of the expression becomes zero or null. The test takes place before each execution of the statement.

2 The expression must be of arithmetic or pointer or pointer to member type or of a class type for which an unambiguous conversion to arithmetic or pointer or pointer to member type exists (12.3).

6.5.2 The `do` statement

1 In the `do` statement the substatement is executed repeatedly until the value of the expression becomes zero or null. The test takes place after each execution of the statement.

2 The expression must be of arithmetic or pointer or pointer to member type or of a class type for which an unambiguous conversion to arithmetic or pointer or pointer to member type exists (12.3).

6.5.3 The `for` statement

1 The `for` statement

`for (` *for-init-statement expression-1*$_{opt}$ `;` *expression-2*$_{opt}$ `)` *statement*

is equivalent to

for-init-statement
`while (` *expression-1* `) {`
        statement
        expression-2 `;`
`}`

except that a `continue` in *statement* (not enclosed in another iteration statement) will execute *expression-2* before re-evaluating *expression-1*. Thus the first statement specifies initialization for the loop; the first expression specifies a test, made before each iteration, such that the loop is exited when the expression becomes zero or null; the second expression often specifies incrementing that is done after each iteration. The first expression must be of arithmetic or pointer or pointer to member type or of a class type for which an unambiguous conversion to arithmetic or pointer or pointer to member type exists (12.3).

2 Either or both of the expressions may be dropped. A missing *expression-1* makes the implied `while` clause equivalent to `while(1)`.

3 If the *for-init-statement* is a declaration, the scope of the names declared extends to the end of the block enclosing the *for-statement*.

6.6 Jump statements

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 named `auto` objects 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 `auto` variable involves the destruction of `auto` variables declared at the point transferred from but not at the point transferred to. (See 6.7 for transfers into blocks). However, the program may be terminated (by calling `exit()` or `abort()`, for example) without destroying automatic class objects.

6.6.1 The `break` statement

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

6.6.2 The `continue` statement

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

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

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

6.6.3 The `return` statement

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, to the return type of the function in which it appears. This may involve the construction and copy of a temporary object (12.2). Flowing off the end of a function is equivalent to a `return` with no value; this results in undefined behavior in a value-returning function.

6.6.4 The `goto` statement

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

6.7 Declaration statement

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 Any initializations of `auto` or `register` variables are done each time their *declaration-statement* is executed. Destruction of local variables declared in the block is done on exit from the block (6.6).

3 It is possible to transfer into a block, but not in a way that causes initializations not to be done. A program that jumps past a declaration with an explicit or implicit initializer is ill formed unless the declaration is in an inner block that is not entered (that is, completely bypassed by the transfer of control) or unless the jump is from a point where the variable has already been initialized. For example,

```
void f()
{
    // ...
    goto lx;    // error: jump past initializer
    // ...
ly:
    X a = 1;
    // ...
lx:
    goto ly;    // ok, jump implies destructor
                // call for 'a'
}
```

4 Initialization of a local object with storage class `static` (7.1.1) is done the first time control passes through its declaration (only). Where a `static` variable is initialized with an expression that is not a *constant-expression*, default initialization to zero of the appropriate type (8.4) happens before its block is first entered.

5 The destructor for a local `static` object will be executed if and only if the variable was constructed. The destructor must be called either immediately before or as part of the calls of the `atexit()` functions (3.4). Exactly when is unspecified.

6.8 Ambiguity resolution

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

2 To disambiguate, the whole *statement* may have to be examined to determine if it is an *expression-statement* or a *declaration*. This disambiguates many examples. For example, assuming `T` is a *simple-type-specifier* (7.1.6),

```
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

1 A declaration introduces one or more names into a program and specifies how those names are to be interpreted. A declaration can specify a storage class, type, and linkage for an object or function. It can also provide the definition of a function or an initial value for an object. A declaration can give a name to a constant (enumeration declaration), declare a new type, or specify a synonym for a type. Inline functions, `const`, `volatile`, and the provision of type-safe linkage are discussed.

7 Declarations

1 Declarations specify the interpretation given to each identifier; they do not necessarily reserve storage associated with the identifier (3.1). Declarations have the form

declaration:
    decl-specifier-seq$_{opt}$ *init-declarator-list*$_{opt}$ **;**
    asm-definition
    function-definition
    template-declaration
    linkage-specification

The declarators in the *init-declarator-list* (8) contain the identifiers being declared. Only in function definitions (8.3) and function declarations may the *decl-specifier-seq* be omitted. Only when declaring a class (9) or enumeration (7.2), that is, when the *decl-specifier* is a *class-specifier* or *enum-specifier,* may the *init-declarator-list* be empty. *asm-definition*s are described in 7.3, and *linkage-specification*s in 7.4. A declaration occurs in a scope (3.2); the scope rules are summarized in 10.4.

7.1 Specifiers

1 The specifiers that can be used in a declaration are

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

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

2 The longest sequence of *decl-specifier*s that could possibly be a type name is taken as the *decl-specifier-seq* of a *declaration*. The sequence must be self-consistent as described below. For example,

```
typedef char* Pc;
static Pc;              // error: name missing
```

Here, the declaration `static Pc` is undefined because no name was specified for the static variable of type `Pc`. To get a variable of type `int` called `Pc`, the *type-specifier* `int` must be present to indicate that the *typedef-name* `Pc` is the name being (re)declared, rather than being part of the *decl-specifier* sequence. For example,

```
void f(const Pc);       // void f(char*const)
void g(const int Pc);   // void g(const int)
```

3 Note that since `signed`, `unsigned`, `long`, and `short` by default imply `int`, a *typedef-name* appearing after one of those specifiers must be 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

1 The storage class specifiers are

storage-class-specifier:
```
        auto
        register
        static
        extern
```

2 The `auto` or `register` specifiers can be applied only to names of objects declared in a block (6.3) and function parameters (8.3). The `auto` declarator 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` declaration is an `auto` declaration, together with a hint to the compiler that the variables declared will be heavily used. The hint may be ignored and in most implementations it will be ignored if the address of the variable is taken.

4 An object declaration is a definition unless it contains the `extern` specifier and has no initializer (3.1).

5 A definition causes the appropriate amount of storage to be reserved and any appropriate initialization (8.4) to be done.

6 The `extern` specifier can be applied only to names of objects and functions. The `static` specifier can be applied only to names of objects and functions and to anonymous unions (9.5). There can be no `static` function declarations within a block, nor any `static` or `extern` function parameters. Static class members are described in (9.4); `extern` cannot be used for class members.

7 A name specified `static` has internal linkage. Objects declared `const` have internal linkage unless they have previously been given external linkage. A name specified `extern` has external linkage unless it has previously been given internal linkage. A file scope name without a *storage-class-specifier* has external linkage unless it has previously been given internal linkage and provided it is not declared `const`. For a nonmember function an `inline` specifier is equivalent to a `static` specifier for linkage purposes (3.3). All linkage specifications for a name must agree. For example,

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

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

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

```
static int b;      // ‘b’ has internal linkage
extern int b;      // ‘b’ still has internal linkage

int c;             // ‘c’ has external linkage
static int c;      // error: inconsistent linkage

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

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
}
```

7.1.2 Function specifiers

1 Some specifiers can be used only in function declarations.

function-specifier:
        `inline`
        `virtual`

2 The `inline` specifier is a hint to the compiler that inline substitution of the function body is to be preferred to the usual function call implementation. The hint may be ignored. For a nonmember function `inline` specifier also gives the function default internal linkage (3.3). A function (5.2.2, 8.2.5) defined within the declaration of a class is `inline` by default.

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

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

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

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

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

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

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

5 The `virtual` specifier may be used only in declarations of nonstatic class member functions within a class declaration; see 10.2.

7.1.3 The `typedef` specifier

1 Declarations containing the *decl-specifier* `typedef` declare identifiers that can be used later for naming fundamental or derived types. The `typedef` specifier may not be used in a *function-definition* (8.3).

typedef-name:
    identifier

Within the scope (3.2) of a `typedef` declaration, each identifier appearing as part of any declarator therein becomes syntactically equivalent to a keyword and names the type associated with the identifier in the way described in 8. A *typedef-name* is thus a synonym for another type. A *typedef-name* does not introduce a new type the way a class declaration (9.1) 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 A `typedef` may be used to redefine a name to refer to the type to which it already refers − even in the scope where the type was originally declared. For example,

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

3 A `typedef` may not redefine a name of a type declared in the same scope to refer to a different type. For example,

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

Similarly, a class may not be declared with the name of a type declared in the same scope to refer to a different type. For example,

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

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

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

typedef struct S T;

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

5 An unnamed class defined in a typedef gets a dummy name and the typedef name for linkage (3.3) and as a synonym for its true name. Such a class cannot have constructors or destructors. For example,

```
typedef struct {
    S();    // an ordinary member function, not a constructor
} S;
```

7.1.4 The `template` specifier

1 The `template` specifier is used to specify families of types or functions; see 14.

7.1.5 The `friend` specifier

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

7.1.6 Type specifiers

1 The type-specifiers are

type-specifier:
    simple-type-specifier
    class-specifier
    enum-specifier
    elaborated-type-specifier
    `::` *typedef-name*
    `const`
    `volatile`

The words `const` and `volatile` may be added to any *type-specifier* in the declaration of an object. Otherwise, at most one *type-specifier* may be given in a declaration. A `const` object may be initialized, but its value may not be changed thereafter without an explicit cast. Unless explicitly declared `extern`, a `const` object does not have external linkage and must be initialized (8.4; 12.1). An integer `const` initialized by a constant expression may be used in constant expressions (5.19). Each element of a `const` array is `const` and each nonfunction, nonstatic member of a `const` class object is `const` (9.3.1). A type which has no user-defined constructors or destructor and no base classes or members with user-defined constructors or destructors is called *ROMable* (but no objects are ever required to be placed in read-only memory). The effect of a write operation on any part of a `const` object of a non-ROMable type is the same as if the object was not `const`. The effect of a write operation on any part of a `const` object of a ROMable type (which is not a sub-object of an object of a non-ROMable type) is undefined. Such an object may be placed in readonly memory.

2 There are no implementation-independent semantics for `volatile` objects; `volatile` is a hint to the compiler to avoid aggressive optimization involving the object because the value of the object may be changed by means undetectable by a compiler. Each element of a `volatile` array is `volatile` and each nonfunction, nonstatic member of a `volatile` class object is `volatile` (9.3.1). An object may be both `const` and `volatile`, with the *type-specifier*s appearing in either order.

3 If the *type-specifier* is missing from a declaration, it is taken to be `int`.

simple-type-specifier:
  qualified-class-specifier
  qualified-type-specifier
  `char`
  `wchar_t`
  `short`
  `int`
  `long`
  `signed`
  `unsigned`
  `float`
  `double`
  `void`

At most one of the words `long` or `short` may be specified together with `int`. Either may appear alone, in which case `int` is understood. The word `long` may appear together with `double`. At most one of the words `signed` and `unsigned` may be specified together with `char`, `short`, `int`, or `long`. Either may appear alone, in which case `int` is understood. The `signed` specifier forces `char` objects and bit-fields to be signed; it is redundant with other integral types.

4 *class-specifier*s and *enum-specifier*s are discussed in 9 and 7.2, respectively.

5

elaborated-type-specifier:
  class-key identifier
  class-key qualified-class-specifier `::` *identifier*
  `enum` *identifier*
  `enum` *qualified-class-specifier* `::` *identifier*

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

6 If an *identifier* is specified, the *elaborated-type-specifier* declares it to be a *class-name* (9.1) or *enum-name* (7.2).

7 If defined, a name declared using the `union` specifier must be defined as a union. If defined, a name declared using either the `class` or `struct` specifier must be defined using either the `class` or `struct` specifier. When a *qualified-class-specifier* is used, the *identifier* must already have been declared as a *class-name*. Names of nested types (9.7) can be qualified by the name of their enclosing class:

qualified-type-specifier:
  typedef-name
  class-name `::` *qualified-type-specifier*

qualified-class-specifier:
  nested-class-specifier
  `::` *nested-class-specifier*

nested-class-specifier:
  class-name
  class-name `::` *nested-class-specifier*

A name qualified by a *class-name* must be a type defined in that class or in a base class of that class. As usual, a name declared in a derived class hides members of that name declared in base classes; see 3.2.

7.2 Enumeration declarations

1 An enumeration is a distinct type (3.6.1) 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
  enumerator-list `,` *enumerator*

enumerator:
  identifier
  identifier = *constant-expression*

The identifiers in an *enumerator-list* are declared as constants, and may appear wherever constants are required. If no enumerators with = appear, then the values of the corresponding constants begin at zero and increase by one as the declaration is read from left to right. An enumerator with = gives the associated identifier the value indicated by the *constant-expression*; subsequent identifiers without initializers continue the progression from the assigned value. The *constant-expression* must be of integral type.

2 The names of enumerators must be distinct from those of ordinary variables and other enumerators in the same scope. The values of the enumerators need not be distinct. An enumerator is considered defined immediately after it and its initializer, if any, has been seen. 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 Each enumeration defines a type that is different from all other types. The type of an enumerator is its enumeration.

4 The *underlying type* of an enumeration is an integral type, not gratuitously larger than `int`,[14] that can represent all enumberator 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.

5 For an enumeration where $e_{\min}$ is the smallest enumerator and $e_{\max}$ is the largest, the values of the enumeration are the values of the underlying type in the range $b_{\min}$ to $b_{\max}$, where $b_{\min}$ and $b_{\max}$ are, respectively, the smallest and largest values of the smallest bit-field that can store $e_{\min}$ and $e_{\max}$. On a two's-complement machine, $b_{\max}$ is the smallest value greater than or equal to $\max(abs(e_{\min}), abs(e_{\max}))$ of the form $2^M - 1$; $b_{\min}$ is zero if $e_{\min}$ is non-negative and $-(b_{\max} + 1)$ otherwise. It is possible to define an enumeration that has values not defined by any of its enumerators.

6 The value of an enumerator or an object of an enumeration type is converted to an integer by integral promotion (4.1). For example,

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

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

```
color c = 1;      // error: type mismatch,
                  // no conversion from int to color

int i = yellow;   // ok: yellow converted to integral value 1
                  // integral promotion
```

See also 19.3.

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

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

This means the program does not crash.

8 Enumerators defined in a class (9) are in the scope of that class and can be referred to outside member functions of that class only by explicit qualification with the class name (5.1). The name of the enumeration itself is also local to the class (9.7). 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
    // ...
}
```

7.3 Asm declarations

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.

7.4 Linkage specifications

1 Linkage (3.3) 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. Linkage to a function written in the C programming language, `"C"`, and linkage to a C++ function, `"C++"`, must be provided by every implementation. Default linkage is `"C++"`. For example,

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

2 Linkage specifications nest. A linkage specification does not establish a scope. A *linkage-specification* may occur only in *file* scope (3.2). A *linkage-specification* for a class applies to nonmember functions and objects declared within it. A *linkage-specification* for a function also applies to functions and objects declared within it. A linkage declaration with a string that is unknown to the implementation is ill-formed.

3 If a function has more than one *linkage-specification*, they must agree; that is, they must specify the same *string-literal*. A function declaration without a linkage specification may not precede the first linkage specification for that function. A function may be declared without a linkage specification after an explicit linkage specification has been seen; the linkage explicitly specified in the earlier declaration is not affected by such a function declaration.

4 At most one of a set of overloaded functions (13) with a particular name can have C linkage. See 7.4.

5 Linkage can be specified for objects. For example,

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

Functions and objects may be declared `static` within the `{}` of a linkage specification. The linkage directive is ignored for such a function or object. Otherwise, a function declared in a linkage specification behaves as if it was explicitly declared `extern`. For example,

```
extern "C" double f();
static double f();     // error
```

is ill-formed (7.1.1). An object defined within an

```
extern "C" { /* ... */ }
```

construct is still defined (and not just declared).

6 Linkage from C++ to objects defined in other languages and to objects defined in C++ from other languages is implementation and language dependent. Only where the object layout strategies of two language implementations are similar enough can such linkage be achieved.

7 When the name of a programming language is used to name a style of linkage in the *string-literal* in a *linkage-specification*, it is recommended that the spelling be taken from the document defining that language, for example, `Ada` (not `ADA`) and `FORTRAN` (not `Fortran`).

8

Declarators

1 A declarator declares a single object, function, or type, within a declaration. The syntax for declarators, including pointers, references, pointers to members, arrays, functions, and types, is explained, as well as how to initialize a declarator in a declaration.

8 Declarators

1 The *init-declarator-list* appearing in a declaration is a comma-separated sequence of declarators, each of which may have an initializer.

> *init-declarator-list:*
> > *init-declarator*
> > *init-declarator-list* `,` *init-declarator*
>
> *init-declarator:*
> > *declarator initializer*$_{opt}$

2 The two components of a *declaration* are the specifiers (*decl-specifier-seq*; 7.1) and the declarators (*init-declarator-list*). The specifiers indicate the fundamental type, storage class, or other properties of the objects and functions being declared. The declarators specify the names of these objects and functions and (optionally) modify the type with operators such as `*` (pointer to) and `()` (function returning). Initial values can also be specified in a declarator; initializers are discussed in 8.4 and 12.6.

3 Each *init-declarator* in a declaration is analyzed separately as if it was in a declaration by itself.[15]

4 Declarators have the syntax

[15] 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
```

declarator:
  direct-declarator
  ptr-operator declarator

direct-declarator:
  declarator-id
  direct-declarator (*parameter-declaration-clause*) *cv-qualifier-seq$_{opt}$* *exception-specification$_{opt}$*
  direct-declarator [*constant-expression$_{opt}$*]
  (*declarator*)

ptr-operator:
  * *cv-qualifier-seq$_{opt}$*
  & *cv-qualifier-seq$_{opt}$*
  qualified-class-specifier :: * *cv-qualifier-seq$_{opt}$*

cv-qualifier-seq:
  cv-qualifier cv-qualifier-seq$_{opt}$

cv-qualifier:
  `const`
  `volatile`

declarator-id:
  id-expression
  qualified-type-specifier

A *class-name* has special meaning in a declaration of the class of that name and when qualified by that name using the scope resolution operator `::` (12.1, 12.4).

8.1 Type names

1 To specify type conversions explicitly, and as an argument of `sizeof` or `new`, the name of a type must be specified. This can be done with a *type-id*, which is syntactically a declaration for an object or function of that type that omits the name of the object or function.

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

type-specifier-seq:
  type-specifier type-specifier-seq$_{opt}$

abstract-declarator:
  ptr-operator abstract-declarator$_{opt}$
  direct-abstract-declarator

direct-abstract-declarator:
  direct-abstract-declarator$_{opt}$ (*parameter-declaration-clause*) *cv-qualifier-seq$_{opt}$* *exception-specification$_{opt}$*
  direct-abstract-declarator$_{opt}$ [*constant-expression$_{opt}$*]
  (*abstract-declarator*)

It is possible to identify uniquely the location in the *abstract-declarator* where the identifier would appear if the construction were a declarator in a declaration. The named type is then the same as the type of the hypothetical identifier. For example,

```
int                 // int i
int *               // int *pi
int *[3]            // int *p[3]
int (*)[3]          // int (*p3i)[3]
int *()             // int *f()
int (*)(double)     // int (*pf)(double)
```

name respectively the types “integer,” “pointer to integer,” “array of 3 pointers to integers,” “pointer to array of 3 integers," "function having no parameters and returning pointer to integer," and "pointer to function of `double` returning an integer."

2 A type can also be named (often more easily) by using a *typedef* (7.1.3).

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

8.1.1 Ambiguity resolution

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
}
```

8.2 Meaning of declarators

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. Except for the declarations of some special functions (12.3, 13.4) a *declarator-id* will be a simple *identifier*. An `auto`, `static`, `extern`, `register`, `friend`, `inline`, `virtual`, or `typedef` specifier applies directly to each *declarator-id* in a *init-declarator-list*; the type specified for each *declarator-id* depends on both the *decl-specifier-seq* and its *declarator*.

2 Thus, a declaration of a particular identifier has the form

```
T D
```

where `T` is a *decl-specifier-seq* and `D` is a declarator. The following subsections give an inductive procedure for determining the type specified for the contained *declarator-id* by such a declaration.

3 First, the *decl-specifier-seq* determines a type. For example, in the declaration

```
int unsigned i;
```

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

```
T D
```

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

4 In a declaration `T D` where `D` is an unadorned identifier the type of this identifier is "`T`."

5 In a declaration `T D` where `D` has the form

```
( D1 )
```

the type of the contained *declarator-id* is the same as that of the contained *declarator-id* in the declaration

```
T D1
```

Parentheses do not alter the type of the embedded *declarator-id*, but they may alter the binding of complex declarators.

8.2.1 Pointers

1 In a declaration `T D` where `D` has the form

`*` *cv-qualifier-seq*$_{opt}$ `D1`

the type of the contained *declarator-id* is "... *cv-qualifier-seq* pointer to `T1`," where `T1` is the type assigned to the contained *declarator-id* in the declaration `T D1`. 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 8.4.

5 There can be no pointers to references (8.2.2) or pointers to bit-fields (9.6).

8.2.2 References

1 In a declaration `T D` where `D` has the form

`&` *cv-qualifier-seq*$_{opt}$ `D1`

the type of the contained *declarator-id* is "... *cv-qualifier-seq* reference to `T1`," where `T1` is the type assigned to the contained *declarator-id* in the declaration `T D1`. The type `void&` is not permitted.

Should cv-qualifiers be allowed here? What does

```
int& const i=0;
```

mean?

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 8.4.3.

3 There can be no references to references, no references to bit-fields (9.6), no arrays of references, and no pointers to references. The declaration of a reference must contain an *initializer* (8.4.3) 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 (8.2.5); see 3.1. A reference must be initialized to refer to a valid object. In particular, null references are prohibited.

8.2.3 Pointers to members

1 In a declaration `T D` where `D` has the form

qualified-class-specifier `:: *` *cv-qualifier-seq*$_{opt}$ `D1`

the type of the contained *declarator-id* is "... *cv-qualifier-seq* pointer to member of class *class-name* of type `T1`," where `T1` is the type assigned to the contained *declarator-id* in the declaration `T D1`.

2 For example,

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

int X::* pmi = &X::a;
void (X::* pmf)(int) = &X::f;
```

declares `pmi` and `pmf` to be a pointer to a member of `X` of type `int` and a pointer to a member of `X` of type `void(int)`, respectively. They 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 (9.4). There are no references to members. See also 5.5 and 5.3.

8.2.4 Arrays

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 "*type-modifier* array of `T`." If the *constant-expression* (5.19) is present, it must be of enumeration or integral type and have a value greater than zero. The constant expression specifies the number of elements in the array. If the constant expression is `N`, the array has `N` elements numbered zero to `N-1`.

2 An array may be constructed from one of the fundamental types (except `void`), from a pointer, from a pointer to member, from a class, from an enumeration, or from another array.

3 When several "array of" specifications are adjacent, a multidimensional array is created; the constant expressions that specify the bounds of the arrays may be omitted only for the first member of the sequence. This elision is useful for function parameters of array types, and when the array is external and the definition, which allocates storage, is given elsewhere. The first *constant-expression* may also be omitted when the declarator is followed by an *initializer-clause* (8.4). In this case the size is calculated from the number of initial elements supplied (8.4.1).

4 The declaration

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

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

```
static int x3d[3][5][7];
```

declares a static three-dimensional array of integers, with rank 3×5×7. In complete detail, `x3d` is an array of three items; each item is an array of five arrays; each of the latter arrays is an array of seven integers. Any of the expressions `x3d`, `x3d[i]`, `x3d[i][j]`, `x3d[i][j][k]` may reasonably appear in an expression.

5 When an identifier of array type appears in an expression, except as the operand of `sizeof` or `&` or used to initialize a reference (8.4.3), it is converted into a pointer to the first member of the array. Because of this conversion, arrays are not modifiable lvalues. Except where it has been declared for a class (13.4.5), the subscript operator `[]` is interpreted in such a way that `E1[E2]` is identical to `*((E1)+(E2))`. Because of the conversion rules that apply to `+`, if `E1` is an array and `E2` an integer, then `E1[E2]` refers to the `E2`-th member of `E1`. Therefore, despite its asymmetric appearance, subscripting is a commutative operation.

6 A consistent rule is followed for multidimensional arrays. If `E` is an n-dimensional array of rank $i \times j \times \cdots \times k$, then `E` appearing in an expression is converted to a pointer to an $(n-1)$-dimensional array with rank $j \times \cdots \times k$. If the `*` operator, either explicitly or implicitly as a result of subscripting, is applied to this pointer, the result is the pointed-to $(n-1)$-dimensional array, which itself is immediately converted into a pointer.

7 For example, consider

```
int x[3][5];
```

Here `x` is a 3×5 array of integers. When `x` appears in an expression, it is converted to a pointer to (the first of three) five-membered arrays of integers. In the expression `x[i]`, which is equivalent to `*(x+i)`, `x` is first converted to a pointer as described; then `x+i` is converted to the type of `x`, which involves multiplying `i` by the length of the object to which the pointer points, namely five integer objects. The results are added and indirection applied to yield an array (of five integers), which in turn is converted to a pointer to the first of the integers. If there is another subscript the same argument applies again; this time the result is an integer.

8 It follows from all this that arrays in C++ are stored row-wise (last subscript varies fastest) and that the first subscript in the declaration helps determine the amount of storage consumed by an array but plays no other part in subscript calculations.

8.2.5 Functions

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`."

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 = *expression*
  decl-specifier-seq abstract-declarator$_{opt}$
  decl-specifier-seq abstract-declarator$_{opt}$ = *expression*

2 The *parameter-declaration-clause* determines the arguments that can be specified, and their processing, when the function is called. If the *parameter-declaration-clause* terminates with an ellipsis, the number of arguments is known only to be equal to or greater than the number of parameters specified; if it is empty, the function takes no arguments. The parameter list `(void)` is equivalent to the empty parameter list. Except for this special case `void` may not be a parameter type (though types derived from `void`, such as `void*`, may). Where syntactically correct, "`, ...`" is synonymous with "`...`". The standard header `<stdarg.h>` contains a mechanism for accessing arguments passed using the ellipsis, see 17.4.8. See 12.1 for the treatment of array arguments. *

3 A single name may be used for several different functions in a single scope; this is function overloading (13). All declarations for a function with a given parameter list must agree exactly both in the type of the value returned and in the number and type of parameters; the presence or absence of the ellipsis is considered part of the function type. Parameter types that differ only in the use of typedef (7.1.3) names, the `register` *storage-class-specifier*, or unspecified array bounds agree exactly.

This needs to be made more precise.

A parameter that is declared as "array of *type*" or "function returning *type*" is adjusted to "pointer to *type*" or "pointer to function returning *type*," respectively. The return type and the parameter types, but not the default parameters (8.2.6), are part of the function type. 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 9.3.1. It is part of the function type.

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

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

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

7 An identifier can optionally be provided as a parameter name; if present in a function declaration, it cannot be used since it goes out of scope at the end of the function declarator (3.2); if present in a function definition (8.3), 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.

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` (7.1.6). The declaration

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

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

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

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

8.2.6 Default parameters

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

2 The declaration

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

declares a function that can be called with zero, one, or two arguments of type `int`. It may be called in any of these ways:

```
point(1,2);  point(1);  point();
```

The last two calls are equivalent to `point(1,4)` and `point(3,4)`, respectively.

3 Default parameter 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 parameter expressions are bound at the end of the class declaration, like names in inline member function bodies (9.3.2). In the following example, `g` will be called with the value `f(2)`:

```
int a = 1;
int f(int);
int g(int x = f(a)); // default parameter: f(::a)

void h() {
    a = 2;
    {
        int a = 3;
        g();        // g(f(::a))
    }
}
```

Local variables may not be used in default parameter expressions. For example,

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

4 Note that default parameters are evaluated before entry into a function and that the order of evaluation of function arguments is implementation dependent. Consequently, parameters of a function may not be used in default parameter expressions. Paramaters of a function declared before a default parameter expression are in scope and may hide global and class member names. For example,

```
int a;
int f(int a, int b = a);    // error: parameter 'a'
                            // used as default parameter
typedef int I;
int g(float I, int b = I(2)); // error: 'float' called
```

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

```
int b;
class X {
    int a;
    mem1(int i = a); // error: nonstatic member 'a'
                     // used as default parameter
    mem2(int i = b); // ok;  use X::b
    static b;
};
```

The declaration of `X::mem2()` is meaningful, however, since no object is needed to access the static member `X::b`. Classes, objects, and members are described in 9.

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

```
int f(int = 0);

void h()
{
    int j = f(1);
    int k = f();          // fine, means f(0)
}

int (*p1)(int) = &f;
int (*p2)() = &f;     // error: type mismatch
```

7 An overloaded operator (13.4) cannot have default parameters.

8.3 Function definitions

1 Function definitions have the form

function-definition:
  decl-specifier-seq$_{opt}$ declarator ctor-initializer$_{opt}$ function-body

function-body:
  compound-statement

The *declarator* in a *function-definition* must contain a declarator with the form

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

as described in 8.2.5.

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

3 A simple example of a complete function definition is

```
int max(int a, int b, int c)
{
    int m = (a > b) ? a : b;
    return (m > c) ? m : c;
}
```

Here `int` is the *decl-specifier-seq*; `max(int a, int b, int c)` is the *declarator*; `{ /* ... */ }` is the *function-body*.

4 A *ctor-initializer* is used only in a constructor; see 12.1 and 12.6.

5 A *cv-qualifier-seq* can be part of a non-static member function declaration, non-static member function definition, or pointer to member function only; see 9.3.1. It is part of the function type.

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

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

8.4 Initializers

1 A declarator may specify an initial value for the identifier being declared.

initializer:
  = *initializer-clause*
  (*expression-list*)

initializer-clause:
  assignment-expression
  { *initializer-list* ,$_{opt}$ }

initializer-list:
  initializer-clause
  initializer-list , *initializer-clause*

2 Automatic, register, static, and external variables at file scope may be initialized by arbitrary expressions involving constants and previously declared variables and functions.

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

3 A pointer of type `const T*` or `volatile T*`, or `const volatile T*`, that is, a pointer to constant, volatile, or constant volatile `T`, can be initialized with a pointer of type `T*`, but none of the reverse initializations are allowed. Objects of type `T` can be initialized with objects of type `T` independently of `const` and `volatile` modifiers on both the initialized variable and on the initializer. For example,

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

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

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

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

4 Default parameter expressions are more restricted; see 8.2.6.

5 Initialization of objects of classes with constructors is described in 12.6.1. Copying of class objects is described in 12.8. The order of initialization of static objects is described in 3.4 and 6.7.

6 Variables with storage class static (3.5) that are not initialized and do not have a constructor are guaranteed to start off as zero converted to the appropriate type. If the object is a `class` or `struct`, its data members start off as zero converted to the appropriate type. If the object is a `union`, its first data member starts off as zero converted to the appropriate type. The initial values of automatic and register variables that are not initialized are indeterminate.

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

8 Note that since `()` is not an initializer,

```
X a();
```

is not the declaration of an object of class `X`, but the declaration of a function taking no argument and returning an `X`.

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

```
int a;

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

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

See 8.2.6 for initializers used as default parameters.

8.4.1 Aggregates

1 An *aggregate* is an array or an object of a class (9) with no constructors (12.1), no private or protected members (11), no base classes (10), and no virtual functions (10.2). When an aggregate is initialized the *initializer* may be an *initializer-clause* consisting of a brace-enclosed, comma-separated list of initializers for the members of the aggregate, written in increasing subscript or member order. If the aggregate contains subaggregates, this rule applies recursively to the members of the subaggregate. If there are fewer initializers in the list than there are members of the aggregate, then the aggregate is padded with zeros of the

appropriate types.

2 For example,

```
struct S { int a; char* b; int c; };
S ss = { 1, "asdf" };
```

initializes `ss.a` with `1`, `ss.b` with `"asdf"`, and `ss.c` with zero.

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

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

5 For example,

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

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

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

is a completely-bracketed initialization: 1, 3, and 5 initialize the first row of the array `y[0]`, namely `y[0][0]`, `y[0][1]`, and `y[0][2]`. Likewise the next two lines initialize `y[1]` and `y[2]`. The initializer ends early and therefore `y[3]` is initialized with zeros. Precisely the same effect could have been achieved by

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

The last (rightmost) index varies fastest (8.2.4).

6 The initializer for `y` begins with a left brace, but the one for `y[0]` does not, therefore three elements from the list are used. Likewise the next three are taken successively for `y[1]` and `y[2]`. Also,

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

initializes the first column of `y` (regarded as a two-dimensional array) and leaves the rest zero.

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

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

```
union u { int a; char* b; };

u a = { 1 };
u b = a;
u c = 1;               // error
u d = { 0, "asdf" };   // error
u e = { "asdf" };      // error
```

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

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

is ill-formed.

10 A *POD-struct*[16] is an aggregate structure that contains neither references nor pointers to members. Similarly, a *POD-union* is an aggregate union that contains neither references nor pointers to members.

8.4.2 Character arrays

1 A `char` array (whether signed or unsigned) may be initialized by a *string-literal*; successive characters of the string initialize the members of the array. For example,

```
char msg[] = "Syntax error on line %s\n";
```

shows a character array whose members are initialized with a string. Note that because `'\n'` is a single character and because a trailing `'\0'` is appended, `sizeof(msg)` is `25`.

2 There may not be more initializers than there are array elements. For example,

```
char cv[4] = "asdf";  // error
```

is ill-formed since there is no space for the implied trailing `'\0'`.

8.4.3 References

1 A variable declared to be a `T&`, that is "reference to type `T`" (8.2.2), must be initialized by an object of type `T` or by an object that can be converted into a `T`. For example,

```
void f()
{
    int i;
    int& r = i;  // `r' refers to `i'
    r = 1;       // the value of `i' becomes 1
    int* p = &r; // `p' points to `i'
    int& rr = r; // `rr' refers to what `r' refers to,
                 // that is, to `i'
}
```

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

3 The initializer may be omitted for a reference only in a parameter declaration (8.2.5), in the declaration of a function return type, in the declaration of a class member within its class declaration (9.2), and where the `extern` specifier is explicitly used. For example,

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

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

```
double d = 2.0;

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

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

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

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

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

6 The lifetime of a temporary object created in this way is the scope in which it is created (3.5). Note that a reference to a class `B` can be initialized by an object of a class `D` provided `B` is an accessible and unambiguous base class of `D` (in that case a `D` is a `B`); see 4.7.

9

Classes

1 A *class* is a user-defined type. A class definition specifies the representation of objects of the class and the set of operations that can be applied to such objects. This chapter presents the syntax and semantics for simple classes.

2 The definition of both `static` and non-`static` members is discussed, and the scope rules involving classes and functions − including local and nested classes containing member functions − are described. The mechanisms for controlling the layout of class objects, for conforming to externally imposed formats, and for maintaining compatibility with C layouts (`struct`s, `union`s and bit-fields) are presented.

3 Derived classes (that is, inheritance), access control, and special member functions are discussed in the next three chapters.

9 Classes

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

> *class-name:*
>   *identifier*
>   *template-class-id*

*Class-specifier*s and *elaborated-type-specifier*s (7.1.6) are used to make *class-name*s. An object of a class consists of a (possibly empty) sequence of members.

> *class-specifier:*
>   *class-head* `{` *member-specification*$_{opt}$ `}`
>
> *class-head:*
>   *class-key identifier*$_{opt}$ *base-clause*$_{opt}$
>   *class-key nested-class-specifier base-clause*$_{opt}$
>
> *class-key:*
>   `class`
>   `struct`
>   `union`

2 The name of a class can be used as a *class-name* even within the *member-specification* of the class specifier itself. A *class-specifier* is commonly referred to as a class definition. A class is considered defined when its *class-specifier* has been seen even though its member functions are in general not yet defined.

3 Objects of an empty class have a nonzero size.

4 Class objects may be assigned, passed as arguments to functions, and returned by functions (except objects of classes for which copying has been restricted; see 12.8). Other plausible operators, such as equality comparison, can be defined by the user; see 13.4.

5 A *structure* is a class declared with the *class-key* `struct`; its members and base classes (10) are public by default (11). A *union* is a class declared with the *class-key* `union`; its members are public by default and it holds only one member at a time (9.5).

9.1 Class names

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 (3.2). If a class name is declared in a scope where an object, function, or enumerator of the same name is also declared the class can be referred to only using an *elaborated-type-specifier* (7.1.6). 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()
    // ...
}
```

An *elaborated-type-specifier* with a *class-key* used without declaring an object or function introduces a class name exactly like a class definition but without defining a class. For example,

```
struct s { int a; };

void g()
{
    struct s;  // hide global struct 's'
    s* p;      // refer to local struct 's'
    struct s { char* p; };  // declare local struct 's'
}
```

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

```
class vector;

class matrix {
    // ...
    friend vector operator*(matrix&, vector&);
};

class vector {
    // ...
    friend vector operator*(matrix&, vector&);
};
```

Declaration of `friends` is described in 11.4, operator functions in 13.4.

3 An *elaborated-type-specifier* (7.1.6) can also be used in the declarations of objects and functions. It differs from a class declaration in that if a class of the elaborated name is in scope the elaborated name will refer to it. For example,

```
struct s { int a; };

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

4 A name declaration takes effect immediately after the *identifier* is seen. For example,

```
class A * A;
```

first specifies `A` to be the name of a class and then redefines it as the name of a pointer to an object of that class. This means that the elaborated form `class A` must be used to refer to the class. Such artistry with names can be confusing and is best avoided.

5 A *typedef-name* (7.1.3) that names a class is a *class-name*; see also 7.1.3.

9.2 Class members

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 `;`

member-declarator-list:
    member-declarator
    member-declarator-list `,` *member-declarator*

member-declarator:
    declarator pure-specifier$_{opt}$
    identifier$_{opt}$ `:` *constant-expression*

pure-specifier:
    `= 0`

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 (9.3), nested types, and member constants. Data members and member functions are static or nonstatic; see 9.4. Nested types are classes (9.1, 9.7) and enumerations (7.2) defined in the class, and arbitrary types declared as members by use of a typedef declaration (7.1.3). The enumerators of an enumeration (7.2) defined in the class are

member constants of the class. Except when used to declare friends (11.4) or to adjust the access to a member of a base class (11.3), *member-declaration*s declare members of the class, and each such *member-declaration* must declare at least one member name of the class. A member may not be declared twice in the *member-specification*, except that a nested class may be declared and then later defined.

2 Note that a single name can denote several function members provided their types are sufficiently different (13). Note that a *member-declarator* cannot contain an *initializer* (8.4). A member can be initialized using a constructor; see 12.1.

3 A member may not be `auto`, `extern`, or `register`.

4 The *decl-specifier-seq* can be omitted in function declarations only. The *member-declarator-list* can be omitted only after a *class-specifier*, an *enum-specifier*, or a *decl-specifier-seq* of the form `friend` *elaborated-type-specifier*. A *pure-specifier* may be used only in the declaration of a virtual function (10.2).

5 Non-`static` (9.4) members that are class objects must be objects of previously declared classes. In particular, a class `cl` may not contain an object of class `cl`, but it may contain a pointer or reference to an object of class `cl`. When an array is used as the type of a nonstatic member all dimensions must be specified.

6 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`.

7 Nonstatic data members of a class declared without an intervening *access-specifier* are allocated so that later members have higher addresses within a class object. The order of allocation of nonstatic data members separated by an *access-specifier* is implementation dependent (11.1). Implementation alignment requirements may cause two adjacent members not to be allocated immediately after each other; so may requirements for space for managing virtual functions (10.2) and virtual base classes (10.1); see also 5.4.

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

9 Two POD-struct (8.4.1) types are layout-compatible if they have the same number of members, and corresponding members (in order) have layout-compatible types.

10 Two POD-union (8.4.1) types are layout-compatible if they have the same number of members, and corresponding members (in any order) have layout-compatible types.

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

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

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

12 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.

13 A pointer to a POD-struct object, suitably converted, points to its initial member (or if that member is a bit-field, then to the unit in which it resides) and vice versa. There may therefore be unnamed padding within a POD-struct object, but not at its beginning, as necessary to achieve appropriate alignment.

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

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

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

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

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

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

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

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

- the declared type of the object,
- a qualified version of the declared type of the object,
- a type that is the signed or unsigned type corresponding to the declared type of the object,
- a type that is the signed or unsigned type corresponding to a qualified version of the declared type of the object,
- an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union), or
- a character type.[17]

the result is undefined.

21 A function member (9.3) with the same name as its class is a constructor (12.1). A static data member, enumerator, member of an anonymous union, or nested type may not have the same name as its class.

9.2.1 Scope rules for classes

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 parameters, and constructor initializers in that class (including such things in nested classes).
2. A name `N` used in a class `S` must refer to the same declaration when re-evaluated in its context and in the completed scope of S.
3. If reordering member declarations in a class yields an alternate valid program under (1) and (2), the program's meaning is undefined.
4. A declaration in a nested declarative region hides a declaration whose declarative region contains the nested declarative region.

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

5. A declaration within a member function hides a declaration whose scope extends to or past the end of the member function's class.

6. The scope of a declaration that extends to or past the end of a class definition also extends to the regions defined by its member definitions, even if defined lexically outside the class (this includes both function member bodies and static data member initializations).

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;
};
```

9.3 Member functions

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

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

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

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

2 The definition of a member function is considered to be within the scope of its class. This means that (provided it is nonstatic 9.4) it can use names of members of its class directly. Such names then refer to the members of the object for which the function was called.

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

the class name using the `::` operator. For example,

```
void tnode::set(char* w, tnode* l, tnode* r)
{
    count = strlen(w+1);
    if (sizeof(tword)<=count)
        error("tnode string too long");
    strcpy(tword,w);
    left = l;
    right = r;
}
```

The notation `tnode::set` specifies that the function `set` is a member of and in the scope of class `tnode`. The member names `tword`, `count`, `left`, and `right` refer to members of the object for which the function was called. Thus, in the call `n1.set("abc",&n2,0)`, `tword` refers to `n1.tword`, and in the call `n2.set("def",0,0)` it refers to `n2.tword`. The functions `strlen`, `error`, and `strcpy` must be declared elsewhere.

4 Members may be defined (3.1) outside their class definition if they have already been declared but not defined in the class definition; they may not be redeclared. See also 3.3. Function members may be mentioned in friend declarations after their class has been defined. Each member function that is called must have exactly one definition in a program.

5 The effect of calling a nonstatic member function (9.4) of a class `X` for something that is not an object of class `X` is undefined.

9.3.1 The `this` pointer

1 In a nonstatic (9.3) member function, the keyword `this` is a non-lvalue expression whose value is the address of the object for which the function is called. The type of `this` in a member function of a class `X` is `X*` unless the member function is declared `const` or `volatile`; in those cases, the type of `this` is `const X*` or `volatile X*`, respectively. A function declared `const` and `volatile` has a `this` with the type `const volatile X*`. See also 19.3.3. For example,

```
struct s {
    int a;
    int f() const;
    int g() { return a++; }
    int h() const { return a++; } // error
};

int s::f() const { return a; }
```

The `a++` in the body of `s::h` is ill-formed because it tries to modify (a part of) the object for which `s::h()` is called. This is not allowed in a `const` member function where `this` is a pointer to `const`, that is, `*this` is a `const`.

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

```
void k(s& x, const s& y)
{
    x.f();
    x.g();
    y.f();
    y.g();      // error
}
```

The call `y.g()` is ill-formed because `y` is `const` and `s::g()` is a non-`const` member function that could (and does) modify the object for which it was called.

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

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

9.3.2 Inline member functions

1 A member function may be defined (8.3) in the class definition, in which case it is `inline` (7.1.2). Defining a function within a class definition is equivalent to declaring it `inline` and defining it immediately after the class definition; this rewriting is considered to be done after preprocessing but before syntax analysis and type checking of the function definition. Thus

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

is equivalent to

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

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

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

2 Member functions can be defined even in local or nested class definitions where this rewriting would be syntactically incorrect. See 9.8 for a discussion of local classes and 9.7 for a discussion of nested classes.

9.4 Static members

1 A data or function member of a class may be declared `static` in the class definition. There is only one copy of a static data member, shared by all objects of the class and any derived classes in a program. A static member is not part of objects of a class. Static members of a global class have external linkage (3.3). The declaration of a static data member in its class definition is *not* a definition and may be of an incomplete type. A definition is required elsewhere; see also 19.3.

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

3 Static members of a local class (9.8) have no linkage and cannot be defined outside the class definition. It follows that a local class cannot have static data members.

4 A static member `mem` of class `cl` can be referred to as `cl::mem` (5.1), that is, independently of any object. It can also be referred to using the `.` and `->` member access operators (5.2.4). When a static member is accessed through a member access operator, the expression on the left side of the `.` or `->` is not evaluated. The static member `mem` exists even if no objects of class `cl` have been created. For example, in the following, `run_chain`, `idle`, and so on exist even if no `process` objects have been created:

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

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

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

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

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

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

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

9.5 Unions

1 A union may be thought of as a class whose member objects all begin at offset zero and whose size is sufficient to contain any of its member objects. At most one of the member objects can be stored in a union at any time. A union may have member functions (including constructors and destructors), but not virtual (10.2) functions. A union may not have base classes. A union may not be used as a base class. An object of a class with a constructor or a destructor or a user-defined assignment operator (13.4.3) cannot be a member of a union. A union can have no `static` data members.

Shouldn't we prohibit references in unions?

2 A union of the form

`union {` *member-specification* `} ;`

is called an anonymous union; it defines an unnamed object (and not a type). The names of the members of an anonymous union must be distinct from other names in the scope in which the union is declared; they are used directly in that scope without the usual member access syntax (5.2.4). For example,

```
void f()
{
    union { int a; char* p; };
    a = 1;
    // ...
    p = "Jennifer";
    // ...
}
```

Here `a` and `p` are used like ordinary (nonmember) variables, but since they are union members they have the same address.

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

4 A union for which objects or pointers are declared is not an anonymous union. For example,

```
union { int aa; char* p; } obj, *ptr = &obj;
aa = 1;        // error
ptr->aa = 1;   // ok
```

The assignment to plain `aa` is ill formed since the member name is not associated with any particular object.

5 Initialization of unions that do not have constructors is described in 8.4.1.

9.6 Bit-fields

1 A *member-declarator* of the form

identifier$_{opt}$ **:** *constant-expression*

specifies a bit-field; its length is set off from the bit-field name by a colon. Allocation of bit-fields within a class object is implementation dependent. Fields are packed into some addressable allocation unit. Fields straddle allocation units on some machines and not on others. Alignment of bit-fields is implementation dependent. Fields are assigned right-to-left on some machines, left-to-right on others.

2 An unnamed bit-field is useful for padding to conform to externally-imposed layouts. Unnamed fields are not members and cannot be initialized. As a special case, an unnamed bit-field with a width of zero specifies alignment of the next bit-field at an allocation unit boundary.

3 A bit-field may not be a static member. A bit-field must have integral or enumeration type (3.6.1). It is implementation dependent whether a plain (neither explicitly signed nor unsigned) `int` field is signed or unsigned. The address-of operator `&` may not be applied to a bit-field, so there are no pointers to bit-fields. Nor are there references to bit-fields.

9.7 Nested class declarations

1 A class may be defined within another class. A class defined within another is called a *nested* class. The name of a nested class is local to its enclosing class. The nested class is in the scope of its enclosing class. Except by using explicit pointers, references, and object names, declarations in a nested class can use only type names, static members, and enumerators from the enclosing class.

```
int x;
int y;

class enclose {
public:
    int x;
    static int s;

    class inner {

        void f(int i)
        {
            x = i;   // error: assign to enclose::x
            s = i;   // ok: assign to enclose::s
            ::x = i; // ok: assign to global x
            y = i;       // ok: assign to global y
        }

        void g(enclose* p, int i)
        {
            p->x = i;   // ok: assign to enclose::x
        }

    };
};

inner* p = 0;   // error 'inner' not in scope
```

Member functions of a nested class have no special access to members of an enclosing class; they obey the usual access rules (11). Member functions of an enclosing class have no special access to members of a nested class; they obey the usual access rules. For example,

```
class E {
    int x;

    class I {
        int y;
        void f(E* p, int i)
        {
            p->x = i;   // error: E::x is private
        }
    };

    int g(I* p)
    {
        return p->y;    // error: I::y is private
    }
};
```

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

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

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

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

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

```
class E {
    class I1;       // forward declaration of nested class
    class I2;
    class I1 {};  // definition of nested class
};
class E::I2 {};   // definition of nested class
```

Like a member function, a friend function defined within a class is in the lexical scope of that class; it obeys the same rules for name binding as the member functions (described above and in 10.4) and like them has no special access rights to members of an enclosing class or local variables of an enclosing function (11).

9.8 Local class declarations

1 A class can be defined within a function definition; such a class is called a *local* class. The name of a local class is local to its enclosing scope. The local class is in the scope of the enclosing scope. Declarations in a local class can use only type names, static variables, `extern` variables and functions, and enumerators from the enclosing scope. For example,

```
int x;
void f()
{
    static int s ;
    int x;
    extern int g();
```

```
    struct local {
        int g() { return x; }    // error: 'x' is auto
        int h() { return s; }    // ok
        int k() { return ::x; }  // ok
        int l() { return g(); }  // ok
    };
    // ...
}

local* p = 0;   // error: 'local' not in scope
```

2 An enclosing function has no special access to members of the local class; it obeys the usual access rules (11). Member functions of a local class must be defined within their class definition. A local class may not have static data members.

9.9 Nested type names

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

1 This chapter explains *inheritance*. A class can be *derived* from one or more other classes, which are then called *base* classes of the derived class. The derived class inherits the properties of its base classes, including its data members and member functions. In addition, the derived class can override *virtual* functions of its bases and declare additional data members, functions, and so on. Access to class members is checked for ambiguity.

2 Sharing among the (base) classes that make up a class can be expressed using *virtual base classes*. Classes can be declared *abstract* to ensure that they are used only as base classes.

3 The final section of this chapter (10.4) is a summary of the C++ scope rules.

10 Derived classes

1 A list of base classes may be specified in a class declaration using the notation:

base-clause:
  `:` *base-specifier-list*

base-specifier-list:
  base-specifier
  base-specifier-list `,` *base-specifier*

base-specifier:
  qualified-class-specifier
  `virtual` *access-specifier*$_{opt}$ *qualified-class-specifier*
  access-specifier `virtual`$_{opt}$ *qualified-class-specifier*

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

The *class-name* in a *base-specifier* must denote a previously declared class (9), which is called a *direct base class* for the class being declared. A class `B` is a base class of a class `D` if it is a direct base class of `D` or a direct base class of one of `D`'s base classes. A class is an *indirect* base class of another if it is a base class but not a direct base class. A class is said to be (directly or indirectly) *derived* from its (direct or indirect) base classes. For the meaning of *access-specifier* see 11. Unless redefined in the derived class, members of a base class can be referred to as if they were members of the derived class. The base class members are said to be *inherited* by the derived class. The scope resolution operator `::` (5.1) may be used to refer to a base member explicitly. This allows access to a name that has been redefined in the derived class. A derived class can itself serve as a base class subject to access control; see 11.2. A pointer to a derived class may be implicitly converted to a pointer to an accessible unambiguous base class (4.6). A reference to a derived class may be implicitly converted to a reference to an accessible unambiguous base class (4.7).

2 For example,

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

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

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

3 Here, an object of class `Derived2` will have a sub-object of class `Derived` which in turn will have a sub-object of class `Base`. A derived class and its base classes can be represented by a directed acyclic graph (DAG) where an arrow means "directly derived from." A DAG of classes is often referred to as a "class lattice." For example,

```
   Base
    ↑
 Derived
    ↑
 Derived2
```

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 Name lookup proceeds from the original class (the named class in the case of a *qualified-id*) along the edges of the lattice until the name is found. If a name is found in more than one class in the lattice, the access is ambiguous (see 10.1.1) unless one occurrence of the name hides[18] all the others. A name `B::f` *hides* a name `A::f` if its class `B` has `A` as a base and the instance of `B` containing `B::f` has the instance of `A` containing `A::f` as a sub-object. The second part of this definition is trivially satisfied when multiple inheritance is not used. For example,

```
void f()
{
    Derived2 x;
    x.a = 1;            // Base::a
    x.b = 2;            // Derived::b
    x.c = 3;            // Derived2::c
    x.Base::b = 4;      // Base::b
    x.Derived::c = 5;   // Base::c
    Base* bp = &x;      // standard conversion:
                        // Derived2* to Base*
}
```

assigns to the five members of `x` and makes `bp` point to `x`.

5 Note that in the *class-name* `::` *id-expression* notation, *id-expression* need not be a member of *class-name*; the notation simply specifies a class in which to start looking for *id-expression*.

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

[18] This criterion is called "dominance" in the ARM.

10.1 Multiple base classes

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

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

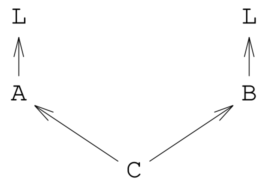
The use of more than one direct base class is often called multiple inheritance.

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

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

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

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

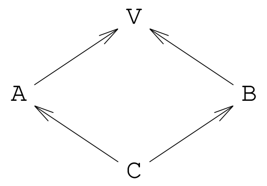
Here, an object of class `C` will have two sub-objects of class `L` as shown below.



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

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

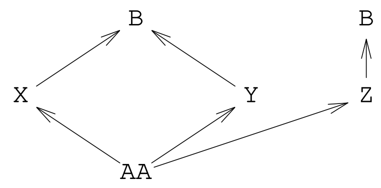
Here class `C` has only one sub-object of class `V`, as shown below.



5 A class may have both virtual and nonvirtual base classes of a given type.

```
class B { /* ... */ };
class X : virtual public B { /* ... */ };
class Y : virtual public B { /* ... */ };
class Z : public B { /* ... */ };
class AA : public X, public Y, public Z { /* ... */ };
```

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



10.1.1 Ambiguities

1 Access to base class members must be unambiguous. Access to a base class member is ambiguous if the *id-expression* or *qualified-id* used does not refer to a unique function, object, type, or enumerator. The check for ambiguity takes place before access control (11). For example,

```
class A {
public:
    int a;
    int (*b)();
    int f();
    int f(int);
    int g();
};

class B {
    int a;
    int b();
public:
    int f();
    int g;
    int h();
    int h(int);
};

class C : public A, public B {};

void g(C* pc)
{
    pc->a = 1;  // error: ambiguous: A::a or B::a
    pc->b();    // error: ambiguous: A::b or B::b
    pc->f();    // error: ambiguous: A::f or B::f
    pc->f(1);   // error: ambiguous: A::f or B::f
    pc->g();    // error: ambiguous: A::g or B::g
    pc->g = 1;  // error: ambiguous: A::g or B::g
    pc->h();    // ok
    pc->h(1);   // ok
}
```

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

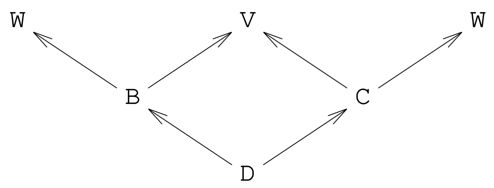
```
class A {
public:
    int f();
};

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

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

A single function, object, type, or enumerator may be reached through more than one path through the directed acyclic graph of base classes. This is not an ambiguity. For example,

```
class V { public: int v; };
class A {
public:
    int a;
    static int   s;
    enum { e };
};
class B : public A, public virtual V {};
class C : public A, public virtual V {};

class D : public B, public C { };

void f(D* pd)
{
    pd->v++;          // ok: only one 'v' (virtual)
    pd->s++;          // ok: only one 's' (static)
    int i = pd->e;    // ok: only one 'e' (enumerator)
    pd->a++;          // error, ambiguous: two 'a's in 'D'
}
```

When virtual base classes are used, a hidden function, object, or enumerator may be reached along a path through the inheritance DAG that does not pass through the hiding function, object, or enumerator. This is not an ambiguity. The identical use with nonvirtual base classes is an ambiguity; in that case there is no unique instance of the name that hides all the others. For example,

```
class V { public: int f();  int x; };
class W { public: int g();  int y; };
class B : public virtual V, public W
{
public:
    int f();  int x;
    int g();  int y;
};
class C : public virtual V, public W { };

class D : public B, public C { void g(); };
```



The names defined in `V` and the left hand instance of `W` are hidden by those in `B`, but the names defined in the right hand instance of `W` are not hidden at all.

```
void D::g()
{
    x++;        // ok: B::x hides V::x
    f();        // ok: B::f() hides V::f()
    y++;        // error: B::y and C's W::y
    g();        // error: B::g() and C's W::g()
}
```

An explicit or implicit conversion from a pointer or reference to a derived class to a pointer or reference to one of its base classes must unambiguously refer to a unique object representing the base class. For example,

```
class V { };
class A { };
class B : public A, public virtual V { };
class C : public A, public virtual V { };
class D : public B, public C { };

void g()
{
    D d;
    B* pb = &d;
    A* pa = &d;  // error, ambiguous: C's A or B's A ?
    V* pv = &d;  // fine: only one V sub-object
}
```

10.2 Virtual functions

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*[19] `Base::vf`. For convenience we say that any virtual function overrides itself. Then in any well-formed class, for each virtual function declared in that class or any of its direct or indirect base classes there is a unique *final overrider* that overrides that function and every other overrider of that function.

3 A program is ill-formed if the return type of any overriding function differs from the return type of the overridden function unless the return type of the latter is pointer or reference (possibly cv-qualified) to a class `B`, and the return type of the former is pointer or reference (respectively) to a class `D` such that `B` is an unambiguous direct or indirect base class of `D`, accessible in the class of the overriding function, and the cv-qualification in the return type of the overriding function is less than or equal to the cv-qualification in the return type of the overridden function. In that case when the overriding function is called as the final overrider of the overridden function, its result is converted to the type returned by the (statically chosen) overridden function. See 5.2.2. For example,

```
class B {};
class D : private B { friend class Derived; };
struct Base {
    virtual void vf1();
    virtual void vf2();
    virtual void vf3();
    virtual B*   vf4();
    void f();
};

struct No_good : public Base {
    D*  vf4();        // error: B (base class of D) inaccessible
};
```

[19] 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 Derived : public Base {
    void vf1();        // virtual and overrides Base::vf1()
    void vf2(int);     // not virtual, hides Base::vf2()
    char vf3();        // error: invalid difference in return type only
    D*  vf4();         // okay: returns pointer to derived class
    void f();
};

void g()
{
    Derived d;
    Base* bp = &d;      // standard conversion:
                        // Derived* to Base*
    bp->vf1();          // calls Derived::vf1()
    bp->vf2();          // calls Base::vf2()
    bp->f();            // calls Base::f() (not virtual)
    B*  p = bp->vf4(); // calls Derived::pf() and converts the
                        //  result to B*
    Derived*  dp = &d;
    D*  q = dp->vf4(); // calls Derived::pf() and does not
                        //  convert the result to B*
    dp->vf2();          // ill formed: argument mismatch
}
```

4 That is, the interpretation of the call of a virtual function depends on the type of the object for which it is called (the dynamic type), whereas the interpretation of a call of a nonvirtual member function depends only on the type of the pointer or reference denoting that object (the static type). See 5.2.2.

5 The `virtual` specifier implies membership, so a virtual function cannot be a global (nonmember) (7.1.2) function. Nor can a virtual function be a static member, since a virtual function call relies on a specific object for determining which function to invoke. A virtual function can be declared a `friend` in another class. A virtual function declared in a class must be defined or declared pure (10.3) in that class.

6 Following are some examples of virtual functions used with multiple base classes:

```
struct A {
    virtual void f();
};

struct B1 : A {    // note non-virtual derivation
    void f();
};

struct B2 : A {
    void f();
};

struct D : B1, B2 {  // D has two separate A sub-objects
};

void foo()
{
    D   d;
    // A*  ap = &d; // would be ill formed: ambiguous
    B1*  b1p = &d;
    A*   ap = b1p;
    ap->f();  // calls D::B1::f
    dp->f();  // ill formed: ambiguous
}
```

In class `D` above there are two occurrences of class `A` and hence two occurrences of the virtual member function `A::f`. The final overrider of `B1::A::f` is `B1::f` and the final overrider of `B2::A::f` is `B2::f`.

7 The following example shows a function that does not have a unique final overrider:

```
struct A {
    virtual void f();
};

struct VB1 : virtual A {   // note virtual derivation
    void f();
};

struct VB2 : virtual A {
    void f();
};

struct Error : VB1, VB2 {  // ill-formed
};

struct Okay : VB1, VB2 {
    void f();
};
```

Both `VB1::f` and `VB2::f` override `A::f` but there is no overrider of both of them in class `Error`. This example is therefore ill-formed. Class `Okay` is well formed, however, because `Okay::f` is a final overrider.

8 The following example uses the well-formed classes from above.

```
struct VB1a : virtual A {  // does not declare f
};

struct Da : VB1a, VB2 {
};

void foe()
{
    VB1a*  vb1ap = new Da;
    vb1ap->f();  // calls VB2:f
}
```

9 Explicit qualification with the scope operator (5.1) suppresses the virtual call mechanism. For example,

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

void D::f() { /* ... */ B::f(); }
```

Here, the function call in `D::f` really does call `B::f` and not `D::f`.

10.3 Abstract classes

1 The abstract class mechanism supports the notion of a general concept, such as a `shape`, of which only more concrete variants, such as `circle` and `square`, can actually be used. An abstract class can also be used to define an interface for which derived classes provide a variety of implementations.

2 An *abstract class* is a class that can be used only as a base class of some other class; no objects of an abstract class may be created except as sub-objects of a class derived from it. A class is abstract if it has at least one *pure virtual function* (which may be inherited: see below). A virtual function is specified *pure* by using a *pure-specifier* (9.2) in the function declaration in the class declaration. A pure virtual function need be defined only if explicitly called with the *qualified-id* syntax (5.1). For example,

```
class point { /* ... */ };
class shape {           // abstract class
    point center;
    // ...
public:
    point where() { return center; }
    void move(point p) { center=p; draw(); }
    virtual void rotate(int) = 0;  // pure virtual
    virtual void draw() = 0;       // pure virtual
    // ...
};
```

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

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

3 Pure virtual functions are inherited as pure virtual functions. For example,

```
class ab_circle : public shape {
    int radius;
public:
    void rotate(int) {}
    // ab_circle::draw() is a pure virtual
};
```

Since `shape::draw()` is a pure virtual function `ab_circle::draw()` is a pure virtual by default. The alternative declaration,

```
class circle : public shape {
    int radius;
public:
    void rotate(int) {}
    void draw(); // must be defined somewhere
};
```

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

4 An abstract class may be derived from a class that is not abstract, and a pure virtual function may override a virtual function which is not pure.

5 Member functions can be called from a constructor of an abstract class; the effect of calling a pure virtual function directly or indirectly for the object being created from such a constructor is undefined.

10.4 Summary of scope rules

1 The scope rules for C++ programs can now be summarized. These rules apply uniformly for all names (including *typedef-names* (7.1.3) and *class-names* (9.1)) wherever the grammar allows such names in the context discussed by a particular rule. This section discusses lexical scope only; see 3.3 for an explanation of linkage issues. The notion of point of declaration is discussed in (3.2).

2 Any use of a name must be unambiguous (up to overloading) in its scope (10.1.1). Only if the name is found to be unambiguous in its scope are access rules considered (11). Only if no access control errors are found is the type of the object, function, or enumerator named considered.

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

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

`ptr->operator->()` eventually resolves to a pointer to `X` (13.4.6).

5 A name that is not qualified in any of the ways described above and that is used in a function that is not a class member must be declared before its use in the block in which it occurs or in an enclosing block or globally. The declaration of a local name hides previous declarations of the same name in enclosing blocks and at file scope. In particular, no overloading occurs of names in different scopes (13.4).

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

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

8 A function parameter name in a function definition (8.3) is in the scope of the outermost block of the function (in particular, it is a local name). A function parameter name in a function declaration (8.2.5) that is not a function definition is in a local scope that disappears immediately after the function declaration. A default parameter is in the scope determined by the point of declaration (3.2) of its parameter, but may not access local variables or nonstatic class members; it is evaluated at each point of call (8.2.6).

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

11

Member Access Control

1 This chapter explains mechanisms for control of access to class members. Access control is based on the use of the keywords `public`, `private`, and `protected` to control access to individual members of a class and on the use of `private`, `protected`, and `public` specifiers to control access to base class members in a derived class object. The `friend` mechanism provides a way of granting individual functions and classes access to members of a class.

2 Access control applies uniformly to function members, data members, member constants, and nested types.

11 Member access control

1 A member of a class can be

`private`; that is, its name can be used only by member functions and friends of the class in which it is declared.

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

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

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

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

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

11.1 Access specifiers

1 Member declarations may be labeled by an *access-specifier* (10):

access-specifier **:** *member-specification*$_{opt}$

An *access-specifier* specifies the access rules for members following it until the end of the class or until another *access-specifier* is encountered. For example,

```
class X {
    int a;  // X::a is private by default: 'class' used
public:
    int b;  // X::b is public
    int c;  // X::c is public
};
```

Any number of access specifiers is allowed and no particular order is required. For example,

```
struct S {
    int a;  // S::a is public by default: 'struct' used
protected:
    int b;  // S::b is protected
private:
    int c;  // S::c is private
public:
    int d;  // S::d is public
};
```

2 The order of allocation of data members with separate *access-specifier* labels is implementation dependent (9.2).

11.2 Access specifiers for base classes

1 If a class is declared to be a base class (10) for another class using the `public` access specifier, the `public` members of the base class are accessible as `public` members of the derived class and `protected` members of the base class are accessible as `protected` members of the derived class (but see 13.1). If a class is declared to be a base class for another class using the `protected` access specifier, the `public` and `protected` members of the base class are accessible as `protected` members of the derived class. If a class is declared to be a base class for another class using the `private` access specifier, the `public` and `protected` members of the base class are accessible as `private` members of the derived class. Private members of a base class remain inaccessible even to derived classes unless `friend` declarations within the base class declaration are used to grant access explicitly.

2 In the absence of an *access-specifier* for a base class, `public` is assumed when the derived class is declared `struct` and `private` is assumed when the class is declared `class`. For example,

```
class B { /* ... */ };
class D1 : private B { /* ... */ };
class D2 : public B { /* ... */ };
class D3 : B { /* ... */ };    // 'B' private by default
struct D4 : public B { /* ... */ };
struct D5 : private B { /* ... */ };
struct D6 : B { /* ... */ };   // 'B' public by default
class D7 : protected B { /* ... */ };
struct D8 : protected B { /* ... */ };
```

Here `B` is a public base of `D2`, `D4`, and `D6`, a private base of `D1`, `D3`, and `D5`, and a protected base of `D7` and `D8`.

3 Because of the rules on pointer conversion (4.6), a static member of a private base class may be inaccessible as an inherited name, but accessible directly. For example,

```
class B {
public:
        int mi;          // nonstatic member
        static int si;   // static member
};
class D : private B {
};
class DD : public D {
        void f();
};

void DD::f() {
        mi = 3;         // error: mi is private in D
        si = 3;         // error: si is private in D
        B  b;
        b.mi = 3;       // okay (b.mi is different from this->mi)
        b.si = 3;       // okay (b.si is the same as this->si)
        B::si = 3;      // okay
        B* bp1 = this; // error: B is a private base class
        B* bp2 = (B*)this;  // okay with cast
        bp2->mi = 3;    // okay and bp2->mi is the same as this->mi
}
```

4 Members and friends of a class `X` can implicitly convert an `X*` to a pointer to a private or protected immediate base class of `X`.

11.3 Access declarations

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

2 For example,

```
class A {
public:
    int z;
    int z1;
};

class B : public A {
    int a;
public:
    int b, c;
    int bf();
protected:
    int x;
    int y;
};
```

```
class D : private B {
    int d;
public:
    B::c;  // adjust access to ‘B::c’
    B::z;  // adjust access to ‘A::z’
    A::z1; // adjust access to ‘A::z1’
    int e;
    int df();
protected:
    B::x;  // adjust access to ‘B::x’
    int g;
};

class X : public D {
    int xf();
};

int ef(D&);
int ff(X&);
```

The external function `ef` can use only the names `c`, `z`, `z1`, `e`, and `df`. Being a member of `D`, the function `df` can use the names `b`, `c`, `z`, `z1`, `bf`, `x`, `y`, `d`, `e`, `df`, and `g`, but not `a`. Being a member of `B`, the function `bf` can use the members `a`, `b`, `c`, `z`, `z1`, `bf`, `x`, and `y`. The function `xf` can use the public and protected names from `D`, that is, `c`, `z`, `z1`, `e`, and `df` (public), and `x`, and `g` (protected). Thus the external function `ff` has access only to `c`, `z`, `z1`, `e`, and `df`. If `D` were a protected or private base class of `X`, `xf` would have the same privileges as before, but `ff` would have no access at all.

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

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

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

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

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

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

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

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

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

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

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

11.4 Friends

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[20] (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 If a class or function mentioned as a friend has not been declared, its name is entered in the smallest non-class scope that encloses the friend declaration.

4 A function first declared in a friend declaration is equivalent to an `extern` declaration (3.3, 7.1.1).

5 A global (but not a member) `friend` function may be defined in a class definition other than a local class definition (9.8). The function is then `inline` and the rewriting rule specified for member functions (9.3.2) is applied. A `friend` function defined in a class is in the (lexical) scope of the class in which it is defined. A friend function defined outside the class is not.

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

7 Friendship is neither inherited nor transitive. For example,

```
class A {
    friend class B;
    int a;
};

class B {
    friend class C;
};

class C  {
    void f(A* p)
    {
        p->a++;  // error: C is not a friend of A
                 // despite being a friend of a friend
    }
};

class D : public B  {
    void f(A* p)
    {
        p->a++;  // error: D is not a friend of A
                 // despite being derived from a friend
    }
};
```

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

11.5 Protected member access

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

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

class D1 : public B {
};

class D2 : public B {
    friend void fr(B*,D1*,D2*);
    void mem(B*,D1*);
};

void fr(B* pb, D1* p1, D2* p2)
{
    pb->i = 1;  // illegal
    p1->i = 2;  // illegal
    p2->i = 3;  // ok (access through a D2)
    int B::*   pmi_B = &B::i;    // illegal
    int D2::*  pmi_D2 = &D2::i;  // ok
}

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

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

11.6 Access to virtual functions

1 The access rules (11) for a virtual function are determined by its declaration and are not affected by the rules for a function that later overrides it. For example,

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

class D : public B {
private:
    f();
};
```

```
void f()
{
    D d;
    B* pb = &d;
    D* pd = &d;

    pb->f();  // ok: B::f() is public,
              // D::f() is invoked
    pd->f();  // error: D::f() is private
}
```

Access is checked at the call point using the type of the expression used to denote the object for which the member function is called (`B*` in the example above). The access of the member function in the class in which it was defined (`D` in the example above) is in general not known.

11.7 Multiple access

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

1 Some member functions have special meaning in the sense that they affect the way a compiler treats objects of their class; that is, they affect the semantics even when they are not explicitly used.

2 This chapter describes *constructors*, *destructors*, and *conversions*, and the free store management operators. Constructors initialize class objects. Destructors are invoked when class objects are destroyed; they are useful for cleaning up. A conversion function specifies a conversion between a class object and another type. The free store management operators allocate and deallocate memory for dynamic objects.

3 Copying of class objects and the use of temporaries are also covered in this chapter.

12 Special member functions

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

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

12.1 Constructors

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

2 A constructor can be invoked for a `const` or `volatile` object. A constructor may not be declared `const` or `volatile` (9.3.1). A constructor may not be `virtual`. A constructor may not be `static`.

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

4 A *default constructor* for a class `X` is a constructor of class `X` that can be called without an argument. A default constructor will be generated for a class `X` only if no constructor has been declared for class `X`.

5 A *copy constructor* for a class `X` is a constructor whose first parameter is of type `X&` or `const X&` and whose other parameters, if any, all have defaults, so that it can be called with a single argument of type `X`. For example, `X::X(const X&)` and `X::X(X&,int=0)` are copy constructors. A copy constructor is generated if and only if no copy constructor is declared in the class definition.[21]

[21] Thus the class definition

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

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

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

7 Constructors for array elements are called in order of increasing addresses (8.2.4).

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

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

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

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

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

For example,

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

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

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

12.2 Temporary objects

1 In some circumstances it may be necessary or convenient for the compiler to generate a temporary object. Precisely when such temporaries are introduced is implementation dependent. When a compiler introduces a temporary object of a class that has a constructor it must ensure that a constructor is called for the temporary object. Similarly, the destructor must be called for a temporary object of a class where a destructor is declared. For example,

```
class X {
    // ...
public:
    // ...
    X(int);
    X(const X&);
    ~X();
};

X f(X);

void g()
{
    X a(1);
    X b = f(X(2));
    a = f(a);
}
```

Here, one might use a temporary in which to construct `X(2)` before passing it to `f()` by `X(X&)`; alternatively, `X(2)` might be constructed in the space used to hold the argument for the first call of `f()`. Also, a temporary might be used to hold the result of `f(X(2))` before copying it to `b` by `X(X&)`; alternatively, `f()`'s result might be constructed in `b`. On the other hand, for many functions `f()`, the expression `a=f(a)` requires a temporary for either the argument `a` or the result of `f(a)` to avoid undesired aliasing of `a`.

22

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

is ill-formed because of ambiguity.

2 The compiler must ensure that every temporary object is destroyed. Ordinarily, temporary objects are destroyed as the last step in evaluating the (unique) expression that (lexically) contains the point where they were created and is not a subexpression of another expression. This is true even if that evaulation ends in throwing an exception. Temporaries created while evaluating default parameter expressions (8.2.6) are considered to be created in the expression that calls the function, not the expression that defines the default parameter.

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

Another form of temporaries is discussed in 8.4.3.

12.3 Conversions

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 (8.4), function arguments (5.2.2, 8.2.5), function return values (6.6.3, 8.2.5), 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 (10.1.1, 12.3.2). Conversions obey the access control rules (11). As ever access control is applied after ambiguity resolution (10.4).

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

1 A constructor with a single parameter specifies a conversion from its parameter type to the type of its class. For example,

```
class X {
    // ...
public:
    X(int);
    X(const char*, int =0);
};

void f(X arg) {
    X a = 1;          // a = X(1)
    X b = "Jessie";   // b = X("Jessie",0)
    a = 2;            // a = X(2)
    f(3);             // f(X(3))
}
```

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

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

12.3.2 Conversion functions

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 ptr-operator$_{opt}$

specifies a conversion from X to the type specified by the *conversion-type-id*. Such member functions are called conversion functions. Classes, enumerations, and *typedef-name*s may not be declared in the *type-specifier-seq*. Neither parameter types nor return type may be specified. A conversion operator is never used to convert a (possibly qualified) object (or reference to an object) to the (possibly qualified) same object type (or a reference to it), or to a (possibly qualified) base class of that type (or a reference to it).

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 Conversion operators are inherited.

4 Conversion functions can be virtual.

5 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()
```

6 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

1 A member function of class `cl` named `~cl` is called a destructor; it is used to destroy values of type `cl` immediately before the object containing them is destroyed. A destructor takes no parameters, and no return type can be specified for it (not even `void`). It is not possible to take the address of a destructor. A destructor can be invoked for a `const` or `volatile` object. A destructor may not be declared `const` or `volatile` (9.3.1). A destructor may not be `static`.

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

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

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

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

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

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

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

7 Destructors are invoked implicitly (1) when an `auto` (3.5) or temporary (12.2, 8.4.3) object goes out of scope, (2) for constructed static (3.5) objects at program termination (3.4), and (3) through use of a *delete-expression* (5.3.4) for objects allocated by a *new-expression* (5.3.3). Destructors can also be invoked explicitly. A *delete-expression* invokes the destructor for the referenced object and passes the address of its memory to a dealloation function (5.3.4, 12.5). For example,

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

void g(X*);

void f()        // common use:
{
    X* p = new X(111);  // allocate and initialize
    g(p);
    delete p;           // cleanup and deallocate
}
```

8 Explicit calls of destructors are rarely needed. One use of such calls is for objects placed at specific addresses using a *new-expression* with the placement option. Such use of explicit placement and destruction of objects can be necessary to cope with dedicated hardware resources and for writing memory management facilities. For example,

```
void* operator new(size_t, void* p) { return p; }

void f(X* p);

static char buf[sizeof(X)];

void g()        // rare, specialized use:
{
    X* p = new(buf) X(222);  // use buf[]
                             // and initialize
    f(p);
    p->X::~X();              // cleanup
}
```

9 Invocation of destructors is subject to the usual rules for member functions, e.g., an object of the appropriate type is required (except invoking `delete` on a null pointer has no effect). When a destructor is invoked for an object, the object no longer exists; if the destructor is explicitly invoked again for the same object the behavior is undefined. For example, if the destructor for an automatic object is explicitly invoked, and the block is subsequently left in a manner that would ordinarily invoke implicit destruction of the object, the behavior is undefined.

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

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

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

12.5 Free store

1 When an object is created with a *new-expression*, an *allocation function* (`operator new()` for non-array objects or `operator new[]()` for arrays) is (implicitly) called to get the required storage. Allocation functions may be static class member functions or global functions. They may be overloaded, but the return type must always be `void*` and the first parameter type must always be `size_t`, an implementation-defined integral type defined in the standard header `<stddef.h>`. 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). 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.

2 The default `::operator new(size_t)` and `::operator new[](size_t)` are always declared and definitions are provided in the library (17.1.1). If a program contains a definition of `::operator new(size_t)` or `::operator new[](size_t)`, that definition is used in preference to the library version.

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. The function may return the address of a block of available storage (suitably aligned) of the requested size or, if it is unable to allocate such a block, it may throw an exception (15) of class `xalloc` (17.1.3.3.2) or a class derived from `xalloc`. For a request for a block of zero size, the pointer returned should be non-null and distinct from the address of any currently allocated object or zero-sized block. If the allocation function returns the null pointer the result is implementation defined. Any other result is undefined.

Can a user-supplied allocation function call the currently installed `new_handler`? How?

4 Any `X::operator new()` or `X::operator new[]()` for a class `X` is a static member (even if not explicitly declared `static`). 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
}
```

5 When an object is deleted with a *delete-expression*, 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. Like allocation functions, deallocation functions may be static class member functions or global functions.

6 The return type of each deallocation function must be `void` and its first parameter must be `void*`. For class member deallocation functions (only) a second parameter of type `size_t` may be added but deallocation functions may not be overloaded. 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 5.3.4) using the usual rules.

7 Default versions of `::operator delete(void*)` and `::operator delete[](void*)`, are provided in the library (17.1.1). If a program contains a definition of `::operator delete(void*)` or `::operator delete[](void*)`, that definition is used in preference to the library version. 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[23] as its second argument.

8 An `X::operator delete()` or `X::operator delete[]()` for a class `X` is a static member (even if not explicitly declared `static`). For example,

[23] If the static class in the *delete-expression* is different from the dynamic class and the destructor is not virtual the size might be incorrect, but that case is already undefined.

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

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

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

12.6 Initialization

1 An object of a class with no constructors, no private or protected members, no virtual functions, and no base classes can be initialized using an initializer list; see 8.4.1. An object of a class with a constructor must either be initialized or have a default constructor (12.1). The default constructor is used for objects that are not explicitly initialized.

12.6.1 Explicit initialization

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

```
class complex {
    // ...
public:
    complex();
    complex(double);
    complex(double,double);
    // ...
};

complex sqrt(complex,complex);

complex a(1);              // initialize by a call of
                           // complex(double)
complex b = a;             // initialize by a copy of 'a'
complex c = complex(1,2);  // construct complex(1,2)
                           // using complex(double,double)
                           // copy it into 'c'
complex d = sqrt(b,c);     // call sqrt(complex,complex)
                           // and copy the result into 'd'
complex e;                 // initialize by a call of
                           // complex()
complex f = 3;             // construct complex(3) using
                           // complex(double)
                           // copy it into 'f'
```

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

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

```
T x = a;
```

The initialization that occurs in `new` expressions (5.3.3) and in base and member initializers (12.6.2) is equivalent to the form

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

3 Arrays of objects of a class with constructors use constructors in initialization (12.1) just like individual objects. If there are fewer initializers in the list than elements in the array, the default constructor (12.1) is used. If there is no default constructor the *initializer-clause* must be complete. For example,

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

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

4 An object of class `M` can be a member of a class `X` only if (1) `M` does not have a constructor, or (2) `M` has a default constructor, or (3) `X` has a constructor and if every constructor of class `X` specifies a *ctor-initializer* (12.6.2) for that member. In case 2 the default constructor is called when the aggregate is created. If a member of an aggregate has a destructor, then that destructor is called when the aggregate is destroyed.

5 Constructors for nonlocal static objects are called in the order they occur in a file; destructors are called in reverse order. See also 3.4, 6.7, 9.4.

12.6.2 Initializing bases and members

1 Initializers for immediate base classes and for members not inherited from a base class may be specified in the definition of a constructor. 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:
  qualified-class-specifier (*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 (8.4.1), initialization by a brace-enclosed list) is the only way to initialize nonstatic `const` and reference members. For example,

```
struct B1 { B1(int); /* ... */ };
struct B2 { B2(int); /* ... */ };

struct D : B1, B2 {
    D(int);
    B1 b;
    const c;
};

D::D(int a) : B2(a+1), B1(a+2), c(a+3), b(a+4)
{ /* ... */ }

D d(10);
```

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

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

3 The class of a *complete object* (1.3) is said to be the *most derived* class for the sub-objects representing base classes of the object. All sub-objects for virtual base classes are initialized by the constructor of the most derived class. If a constructor of the most derived class does not specify a *mem-initializer* for a virtual base class then that virtual base class must have a default constructor or no constructors. Any *mem-initializer*s for virtual classes specified in a constructor for a class that is not the class of the complete object are ignored. For example,

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

class A : public virtual V {
public:
    A();
    A(int);
    // ...
};
```

```
class B : public virtual V {
public:
    B();
    B(int);
    // ...
};

class C : public A, public B, private virtual V {
public:
    C();
    C(int);
    // ...
};

A::A(int i) : V(i) { /* ... */ }
B::B(int i) { /* ... */ }
C::C(int i) { /* ... */ }

V v(1); // use V(int)
A a(2); // use V(int)
B b(3); // use V()
C c(4); // use V()
```

4 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`.

12.7 Constructors and destructors

1 Member functions may be called in constructors and destructors. This implies that virtual functions may be called (directly or indirectly). The function called will be the one defined in the constructor's (or destructor's) own class or its bases, but *not* any function overriding it in a derived class. This ensures that unconstructed parts of objects will not be accessed during construction or destruction. For example,

```
class X {
public:
    virtual void f();
    X() { f(); }   // calls X::f()
    ~X() { f(); }  // calls X::f()
};

class Y : public X {
    int& r;
public:
    void f()
    {
        r++;  // disaster if 'r' is uninitialized
    }
    Y(int& rr) :r(rr) {} // calls X::X() which calls X::f()
};
```

2 The effect of calling a pure virtual function directly or indirectly for the object being constructed from a constructor, except using explicit qualification, is undefined (10.3).

12.8 Copying class objects

1 A class object can be copied in two ways, by assignment (5.17) and by initialization (12.1, 8.4) including function argument passing (5.2.2) and function value return (6.6.3). Conceptually, for a class `X` these two operations are implemented by an assignment operator and a copy constructor (12.1). If not declared by the programmer, they will if possible be automatically defined ("synthesized") as memberwise assignment and memberwise initialization of the base classes and non-static data members of `X`, respectively. An explicit declaration of either of them will suppress the synthesized definition.

2 If all bases and members of a class `X` have copy constructors accepting `const` parameters, the synthesized copy constructor for `X` will have a single parameter of type `const X&`, as follows:

```
X::X(const X&)
```

Otherwise it will have a single parameter of type `X&`:

```
X::X(X&)
```

and programs that attempt initialization by copying of `const X` objects will be ill formed.

3 Similarly, if all bases and members of a class `X` have assignment operators accepting `const` parameters, the synthesized assignment operator for `X` will have a single parameter of type `const X&`, as follows:

```
X& X::operator=(const X&)
```

Otherwise it will have a single parameter of type `X&`:

```
X& X::operator=(X&)
```

and programs that attempt assignment by copying of `const X` objects will be ill formed. The synthesized assignment operator will return a reference to the object for which is invoked.

4 Objects representing virtual base classes will be initialized only once by a generated copy constructor. Objects representing virtual base classes will be assigned only once by a generated assignment operator.

5 Memberwise assignment and memberwise initialization implies that if a class `X` has a member or base of a class `M`, `M`'s assignment operator and `M`'s copy constructor are used to implement assignment and initialization of the member or base, respectively, in the synthesized operations. The default assignment operation cannot be generated for a class if the class has:

- a non-static data member that is a `const` or a reference,
- a non-static data member or base class whose assignment operator is inaccessible to the class, or
- a non-static data member or base class with no assignment operator for which a default assignment operation cannot be generated.

Similarly, the default copy constructor cannot be generated for a class if a non-static data member or a base of the class has an inaccessible copy constructor, or has no copy constructor and the default copy constructor cannot be generated for it.

6 The default assignment and copy constructor will be declared, but they will not be defined (that is, a function body generated) unless needed. That is, `X::operator=()` will be generated only if no assignment operation is explicitly declared and an object of class `X` is assigned an object of class `X` or an object of a class derived from `X` or if the address of `X::operator=` is taken. Initialization is handled similarly.

7 If implicitly declared, the assignment and the copy constructor will be public members and the assignment operator for a class `X` will be defined to return a reference of type `X&` referring to the object assigned to.

8 If a class `X` has any `X::operator=()` that has a parameter of class `X`, the default assignment will not be generated. If a class has any copy constructor defined, the default copy constructor will not be generated. For example,

```
class X {
    // ...
public:
    X(int);
    X(const X&, int = 1);
};

X a(1);          // calls X(int);
X b(a, 0);       // calls X(const X&, int);
X c = b;         // calls X(const X&, int);
```

9 Assignment of class objects X is defined in terms of `X::operator=(const X&)`. This implies (12.3) that objects of a derived class can be assigned to objects of a public base class. For example,

```
class X {
public:
    int b;
};

class Y : public X {
public:
    int c;
};

void f()
{
    X x1;
    Y y1;

    x1 = y1;    // ok
    y1 = x1;    // error
}
```

Here `y1.b` is assigned to `x1.b` and `y1.c` is not copied.

10 Copying one object into another using the default copy constructor or the default assignment operator does not change the structure of either object. For example,

```
struct s {
    virtual f();
    // ...
};

struct ss : public s {
    f();
    // ...
};

void f()
{
    s a;
    ss b;
    a = b;       // really a.s::operator=(b)
    b = a;       // error
    a.f();       // calls s::f
    b.f();       // calls ss::f
    (s&)b = a;   // assign to b's s part
                 // really ((s&)b).s::operator=(a)
    b.f();       // still calls ss::f
}
```

The call `a.f()` will invoke `s::f()` (as is suitable for an object of class s (10.2)) and the call `b.f()` will call `ss::f()` (as is suitable for an object of class ss).

13

Overloading

1 This chapter gives the syntax and semantics of operator and function overloading. Overloading allows multiple functions with the same name to be defined provided their parameter lists differ sufficiently for calls to be resolved. By overloading operators, the programmer can redefine the meaning of most C++ operators (including function call `()`, subscripting `[]`, assignment `=`, address-of `&`, and class member access `->`) when at least one operand is a class object.

13 Overloading

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

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

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

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

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

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

Similarly, since for any type `T`, a `T`, a `const T`, and a `volatile T` accept the same set of initializer values, functions with parameter types differing only in this respect may not have the same name. It is, however, possible to distinguish between `const T&`, `volatile T&`, and plain `T&` so functions that differ only in this respect may be defined. Similarly, it is possible to distinguish between `const T*`, `volatile T*`, and plain `T*` so functions that differ only in this respect may be defined.

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

3 Member functions that differ only in that one is a `static` member and the other isn't may not have the same name (9.4).

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

```
typedef int Int;

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

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

```
enum E { a };

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

5 Parameter types that differ only in a pointer `*` versus an array `[]` are identical, that is, the array declaration is adjusted to become a pointer declaration (8.2.5). Note that only the second and subsequent array dimensions are significant in parameter types (8.2.4).

```
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]);
```

13.1 Declaration matching

1 Two function declarations of the same name refer to the same function if they are in the same scope and have identical parameter types (13). A function member of a derived class is *not* in the same scope as a function member of the same name in a base class. For example,

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

class D : public B {
public:
    int f(char*);
};
```

Here `D::f(char*)` hides `B::f(int)` rather than overloading it.

```
void h(D* pd)
{
    pd->f(1);       // error:
                    // D::f(char*) hides B::f(int)
    pd->B::f(1);    // ok
    pd->f("Ben");   // ok, calls D::f
}
```

A locally declared function is not in the same scope as a function in file 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
}
```

2 Different versions of an overloaded member function may be given different access rules. For example,

```
class buffer {
private:
    char* p;
    int size;

protected:
    buffer(int s, char* store) { size = s; p = store; }
    // ...

public:
    buffer(int s) { p = new char[size = s]; }
    // ...
};
```

13.2 Argument matching

1 A call of a given function name chooses, from among all functions by that name that are in scope and for which a set of conversions exists so that the function could possibly be called, the function whose parameters best match the given arguments. The best-matching function is the intersection of sets of functions that best match on each argument. Unless this intersection has exactly one member, the call is ill formed. The function thus selected must be a better match to the call than any other candidate function. Otherwise, the call is ill formed.

2 One function is a better match than another if for each argument in the call, the corresponding parameter of the first function is at least as good a match as the corresponding parameter of the second function, and for some argument the corresponding parameter of the first function is a better match.

3 For purposes of argument matching, a function with *n* default parameters (8.2.6) is considered to be *n+1* functions with different numbers of parameters.

4 For purposes of argument matching, a nonstatic member function is considered to have an extra parameter specifying the object for which it is called. This extra parameter requires a match either by the object or pointer specified in the explicit member function call notation (5.2.4) or by the first operand of an overloaded operator (13.4). No temporaries will be introduced for this extra parameter and no user-defined conversions will be applied to achieve a type match.

5 Where a member funtion of a class `X` is explicitly called for a pointer using the `->` operator, this extra parameter is assumed to have type `X*`, cv-qualified by the cv-qualifiers of the member function, if any. For example, the extra parameter is assumed to have type `const X*` for a `const` member function. Where the member function is explicitly called for an object or reference using the `.` operator, or the function is invoked for the first operand of an overloaded operator (13.4), this extra parameter is assumed to have type `X&` cv-qualified by the cv-qualifiers of the member function, if any. The first operand of `->*` and `.*` is treated in the same way as the first operand of `->` and `.`, respectively.

6 An ellipsis in a parameter list (8.2.5) is a match for an argument of any type.

7 For a given argument, no sequence of conversions will be considered that contains more than one user-defined conversion or that can be shortened by deleting one or more conversions into another sequence that leads to the type of the corresponding parameter of any function in consideration. Such a sequence is called a *best-matching* sequence.

8 For example, `int`→`float`→`double` is a sequence of conversions from `int` to `double`, but it is not a best-matching sequence because it contains the shorter sequence `int`→`double`.

9 Except as mentioned below, the following *trivial conversions* involving a type `T` do not affect which of two conversion sequences is better:

from:	**to:**
`T`	`T&`
`T&`	`T`
`T[]`	`T*`
`T(args)`	`(*T)(args)`
`T`	`const T`
`T`	`volatile T`
`T`	`const volatile T`
`T*`	`const T*`
`T*`	`volatile T*`
`T*`	`const volatile T*`

Sequences of trivial conversions that differ only in order are indistinguishable. Note that functions with parameters of type `T`, `const T`, `volatile T`, and `const volatile T` accept exactly the same set of values. Where necessary, `const` and `volatile` are used as tie-breakers as described in rule [1] below.

10 A temporary variable is needed for a parameter of type `T&` if the argument is not an lvalue, has a type different from `T`, or is a `volatile` and `T` isn't. This does not affect argument matching. It may, however, affect the correctness of the resulting match since a temporary may not be used to initialize a non-`const` reference (8.4.3).

11 Sequences of conversions are considered according to these rules:

12 [1] Exact match: Sequences of zero or more trivial conversions are better than all other sequences. Of these, those that do not convert `T*` to `const T*`, `T*` to `volatile T*`, `T&` to `const T&`, or `T&` to `volatile T&` are better than those that do.

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

[3] Match with standard conversions: Of sequences not mentioned in [2], those with only standard (4) and trivial conversions are better than all others. Of these, if `B` is derived directly or indirectly from `A`, converting a `B*` to `A*` is better than converting to `void*` or `const void*`; further, if `C` is publicly derived directly or indirectly from `B`, converting a `C*` to `B*` is better than converting to `A*` and converting a `C&` to `B&` is better than converting to `A&`. The class hierarchy acts similarly as a selection mechanism for pointer to member conversions (4.8).

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

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

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

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

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

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

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

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

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

Except when one conversion sequence is a subsequence of another, if user-defined coercions are needed for an argument, no account is taken of any standard coercions that might also be involved. For example,

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

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

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

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

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

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

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

struct Y {
    Y(X);
};
```

```
Y operator+(Y,Y);

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

13.3 Address of overloaded function

1 A use of a function name without arguments selects, among all functions of that name that are in scope, the (only) function that exactly matches the target. The target may be

an object being initialized (8.4)

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)

a function return type (8.2.5)

an explicit type conversion (5.2.3, 5.4)

2 Note that if `f()` and `g()` are both overloaded functions, the cross product of possibilities must be considered to resolve `f(&g)`, or the equivalent expression `f(g)`.

3 For example,

```
int f(double);
int f(int);
(int (*)(int))&f      // cast expression as selector
int (*pfd)(double) = &f;
int (*pfi)(int) = &f;
int (*pfe)(...) = &f; // error: type mismatch
```

The last initialization is ill-formed because no `f()` with type `int(...)` has been defined, and not because of any ambiguity.

4 Note also that there are no standard conversions (4) of one pointer to function type into another (4.6). In particular, even if `B` is a public base of `D` we have

```
D* f();
B* (*p1)() = &f;       // error

void g(D*);
void (*p2)(B*) = &g;   // error
```

13.4 Overloaded operators

1 Most operators can be overloaded.

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 implement operators (13.4.1, 13.4.2). 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 operators `new`, `new[]`, `delete` and `delete[]` are described in 12.5 and and the rules described below in this section do not apply to them.

6 An operator function must either be a non-static member function or have at least one parameter of 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 predefined meaning of the operators =, (unary) `&`, and `,` (comma) applied to class objects may be changed. Except for `operator=()`, operator functions are inherited; see 12.8 for the rules for `operator=()`.

7 Identities among operators applied to basic types (for example, `++a` ≡ `a+=1`) need not hold for operators applied to class types. Some operators, for example, `+=`, require an operand to be an lvalue when applied to basic types; this is not required when the operators are declared for class types.

8 An overloaded operator cannot have default parameters (8.2.6).

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

1 A prefix unary operator may be declared by a nonstatic member function (9.3) with no parameters or a nonmember 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, argument matching (13.2) determines which, if any, interpretation is used. See 13.4.7 for an explanation of postfix unary operators, that is, `++` and `--`.

13.4.2 Binary operators

1 A binary operator may be declared either by a nonstatic member function (9.3) with one parameter or by a nonmember 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, argument matching (13.2) determines which, if any, interpretation is used.

13.4.3 Assignment

1 The assignment function `operator=()` must be a nonstatic member function; it is not inherited (12.8). Instead, unless the user defines `operator=` for a class `X`, `operator=` is defined, by default, as memberwise assignment of the members of class `X`.

```
X& X::operator=(const X& from)
{
    // copy members of X
}
```

13.4.4 Function call

1 Function call

postfix-expression (*expression-list*$_{opt}$)

is considered a binary operator with the *postfix-expression* as the first operand and the possibly empty *expression-list* as the second. The name of the defining function is `operator()`. Thus, a call `x(arg1,arg2,arg3)` is interpreted as `x.operator()(arg1,arg2,arg3)` for a class object `x`. `operator()` must be a nonstatic member function.

13.4.5 Subscripting

1 Subscripting

postfix-expression [*expression*]

is considered a binary operator. A subscripting expression `x[y]` is interpreted as `x.operator[](y)` for a class object `x`. `operator[]` must be a nonstatic member function.

13.4.6 Class member access

1 Class member access using `->`

postfix-expression `->` *primary-expression*

is considered a unary operator. An expression `x->m` is interpreted as `(x.operator->())->m` for a class object `x`. It follows that `operator->()` must return either a pointer to a class that has a member `m` or an object of or a reference to a class for which `operator->()` is defined. `operator->` must be a nonstatic member function.

13.4.7 Increment and decrement

1 The prefix and postfix increment operators can be defined by a function called `operator++`. If this function is a member function with no parameters, or a non-member function with one class parameter, it defines the prefix increment operator ++ for objects of that class. If the function is a member function with one parameter (which must be of type `int`) or a non-member function with two parameters (the second must be of type `int`), it defines the postfix increment operator ++ for objects of that class. When the postfix increment is called, the `int` argument will have value zero. For example,

```
class X {
public:
    const X&   operator++();      // prefix ++a
    const X&   operator++(int);   // postfix a++
};

class Y {
public:
};
const Y&   operator++(Y&);        // prefix ++b
const Y&   operator++(Y&, int);   // postfix b++
```

```
void f(X a, Y b)
{
    ++a;         // a.operator++();
    a++;         // a.operator++(0);
    ++b;         // operator++(b);
    b++;         // operator++(b, 0);

    a.operator++();     // explicit call: like ++a;
    a.operator++(0);    // explicit call: like a++;
    operator++(b);      // explicit call: like ++b;
    operator++(b, 0);   // explicit call: like b++;
}
```

2 The prefix and postfix decrement operators `--` are handled similarly.

14

Templates

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 `Shapes`. 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.

14 Templates

1 The template design was first presented in Bjarne Stroustrup: *Parameterized Types for C++*, Proc. USENIX C++ Conference, Denver, October 1988.

14.1 Templates

1 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*

template-parameter:
  type-parameter
  parameter-declaration

type-parameter:
  `class` *identifier*

The *declaration* in a *template-declaration* must declare or define a function or a class or define a static data member of a template class.

2 A *type-parameter* defines its *identifier* to be a *type-id* in the scope of the template declaration. 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*.

3 Template names obey the usual scope and access control rules. A *template-declaration* is a *declaration*. A *template-declaration* may appear only as a global declaration.

14.2 Class Templates

1 A class template specifies how individual classes can be constructed much as a class declaration specifies how individual objects can be constructed. A vector class template might be declared like this:

```
template<class T> class vector {
    T* v;
    int sz;
public:
    vector(int);
    T& operator[](int);
    T& elem(int i) { return v[i]; }
    // ...
};
```

The prefix `template <class T>` specifies that a template is being declared and that a *type-id* `T` will be used in the declaration. In other words, `vector` is a parameterized type with `T` as its parameter.

2 A class can be specified by a *template-class-id*:

template-class-id:
    template-name `<` *template-argument-list* `>`

template-name:
    identifier

template-argument-list:
    template-argument
    template-argument-list `,` *template-argument*

template-argument:
    expression
    type-id

3 A *template-class-id* is a *class-name* (9).

4 A class generated from a class template is called a template class, as is a class specifically defined with a *template-class-id* as its name; see 14.5.

5 A *template-class-id* where the *template-name* is not defined names an undefined class.

6 A class template name must be unique in a program and may not be declared to refer to any other template, class, function, object, value, or type in the same scope.

7 The types of the *template-argument*s specified in a *template-class-id* must match the types specified for the template in its *template-parameter-list*.

8 Other *template-argument*s must be *constant-expression*s, addresses of objects or functions with external linkage,[24] or of static class members. An exact match (13.2) is required for nontype arguments.

9 For example, `vector`s 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);
```

10 Here, `vector<int>` and `vector<complex>` are template classes, and their definitions will by default be generated from the `vector` template.

[24] 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.

11 Since a *template-class-id* is a *class-name*, it can be used wherever a *class-name* can be used. For example,

```
class vector<Shape*>;

vector<Window>* current_window;

class svector : public vector<Shape*> { /* ... */ };
```

12 Definition of class template member functions is described in 14.6.

14.3 Type Equivalence

1 Two *template-class-id*s refer to the same class if their *template* names are identical and their arguments have identical values. For example,

```
template<class E, int size> class buffer;

buffer<char,2*512> x;
buffer<char,1024> y;
```

declares `x` and `y` to be of the same type, and

```
template<class T, void(*err_fct)()>
    class list { /* ... */ };

list<int,&error_handler1> x1;
list<int,&error_handler2> x2;
list<int,&error_handler2> x3;
list<char,&error_handler2> x4;
```

declares `x2` and `x3` to be of the same type. Their type differs from the types of `x1` and `x4`.

14.4 Function Templates

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, as is a function defined with a type that matches a function template; see 14.5.

2 Template arguments are not explicitly specified when calling a function template; instead, overloading resolution is used. For example,

```
vector<complex> cv(100);
vector<int> ci(200);

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

3 A template function may be overloaded either by (other) functions of its name or by (other) template functions of that same name. Overloading resolution for template functions and other functions of the same name is done in 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] Try ordinary overloading resolution (13.2) for the functions; if a function is found, call it.

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

5 The same process is used for type matching for pointers to functions (13.3).

6 Here is an example:

```
template<class T> T max(T a, T b) { return a>b?a:b; };

void f(int a, int b, char c, char d)
{
    int m1 = max(a,b);  // max(int a, int b)
    char m2 = max(c,d); // max(char a, char b)
    int m3 = max(a,c);  // error: cannot generate
                        // max(int,char)
}
```

7 For example, adding

```
int max(int,int);
```

to the example above would resolve the third call, by providing a function that could be called for max(a,c) after using the standard conversion of char to int for c.

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

9 Every *template-parameter* specified in the *template-parameter-list* must be used in the parameter list of a function template. For example,

```
template<class T> T* create();  // error

template<class T>
    void f() {  // error
        T a;
        // ...
    }
```

All *template-parameter*s for a function template must be *type-parameter*s.

14.5 Declarations and Definitions

1 There must be exactly one definition for each template of a given name in a program. There can be many declarations. The definition is used to generate specific template classes and template functions to match the uses of the template.

2 Using a *template-class-id* constitutes a declaration of a template class.

3 Calling a function template or taking its address constitutes a declaration of a template function. There is no special syntax for calling or taking the address of a template function; the name of a function template is used exactly as is a function name. Declaring a function with the same name as a function template with a matching type constitutes a declaration of a specific template function.

4 If the definition of a specific template function or specific template class is needed to perform some operation and if no explicit definition of that specific template function or class is found in the program, a definition is generated.

5 The definition of a (nontemplate) function with a type that exactly matches the type of a function template declaration is a definition of that specific function template. For example,

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

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

Here, the function definition will be used as the sort function for arguments of type vector<char*>. For other vector types the appropriate function definition is generated from the template.

6 A class can be defined as the definition of a template class. For example,

```
template<class T> class stream { /* ... */ };

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

Here, the class declaration will be used as the definition of streams of characters (`stream<char>`). Other streams will be handled by template classes generated from the class template.

7 No operation that requires a defined class can be performed on a template class until the class template has been seen. After that, a specific template class is considered defined immediately before the first global declaration that names it.

14.6 Member Function Templates

1 A member function of a template class is implicitly a template function with the template parameters of its class as its template parameters. 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[]()
```

14.7 Friends

1 A friend function of a template may or may not be a template function. For example,

```
template<class T> class task {
    // ...
    friend void next_time();
    friend task<T>* preempt(task<T>*);
    friend task* prmt(task*);           // error
    // ...
};
```

Here, `next_time()` becomes the friend of all `task` classes, and each `task` has an appropriately typed function called `preempt()` as a friend. The `preempt` functions might be defined as a template.

```
template<class T>
    task<T>* preempt(task<T>* t) { /* ... */ }
```

2 The declaration of `prmt()` is ill-formed because there is no type `task`, only specific template types, `task<int>`, `task<record>`, and so on.

14.8 Static Members and Variables

1 Each template class or function generated from a template has its own copies of any static variables or members. For example,

```
template<class T> class X {
    static T s;
    // ...
};

X<int> aa;
X<char*> bb;
```

Here `X<int>` has a static member `s` of type `int` and `X<char*>` has a static member `s` of type `char*`.

2 Static class member templates are defined similarly to member function templates. For example,

```
template<class T> T X<T>::s = 0;

int X<int>::s = 3;
```

3 Similarly,

```
template<class T> f(T* p)
{
    static T s;
    // ...
};

void g(int a, char* b)
{
    f(&a);
    f(&b);
}
```

Here `f(int*)` has a static member `s` of type `int` and `f(char**)` has a static member `s` of type `char*`.

15

Exception Handling

1 Exception handling provides a way of transferring control and information to an unspecified caller that has expressed willingness to handle exceptions of a given type. Exceptions of arbitrary types can be *thrown* and *caught* and the set of exceptions a function may throw can be specified. The termination model of exception handling is provided. Exception handling can be used to support notions of error handling and fault-tolerant computing.

15 Exception handling

1 The exception handling design is a variant of the scheme presented in Andrew Koenig and Bjarne Stroustrup: *Exception Handling for C++ (revised)*, Proc. USENIX C++ Conference, San Francisco, April 1990.

15.1 Exception Handling

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*$_{\text{opt}}$

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` statement may be used to transfer control out of a handler, but not into one.

15.2 Throwing an Exception

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 15.4.

3 A *throw-expression* initializes a temporary object of the static type of the operand of `throw` and uses that temporary to initialize the appropriately-typed variable named in the handler. Except for the restrictions on type matching mentioned in 15.4 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.

4 If the use of the temporary object can be eliminated without changing the meaning of the program except for the execution of constructors and destructors associated with the use of the temporary object (12.2), then the exception in the handler may be initialized directly with the argument of the throw expression.

5 A *throw-expression* with no operand rethrows the exception being handled. A *throw-expression* with no operand may appear only in a handler or in a function directly or indirectly called from a handler. 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
}
```

15.3 Constructors and Destructors

1 As control passes from a throw-point to a handler, destructors are invoked for all automatic objects constructed since the *try-block* was entered.

2 An object that is partially constructed will have destructors executed only for its fully constructed sub-objects. Also, should a constructor for an element of an automatic array throw an exception, only the constructed elements of that array will be destroyed.

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*."

15.4 Handling an Exception

1 A *handler* with type `T`, `const T`, `T&`, or `const T&` is a match for a *throw-expression* with an object of type `E` if

[1] `T` and `E` are the same type, or
[2] `T` is an accessible (4.6) base class of `E` at the throw point, or
[3] `T` is a pointer type and `E` is a pointer type that can be converted to `T` by a standard pointer conversion (4.6) at the throw point.

2 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. A program is ill-formed if it places a handler for a base class ahead of a handler for its derived class (or a handler for a pointer or reference to base ahead of a handler for a pointer or reference to derived) since that would ensure that the handler for the derived class would never be invoked. The processor shall diagnose this error if the classes are defined at the beginning of the try block.

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 must 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*. If no matching handler is found in a program, the function `terminate()` (15.6.1) is called.

6 An exception is considered handled upon entry to a handler. The stack will have been unwound at that point.

15.5 Exception Specifications

1 Raising or catching an exception affects the way a function relates to other functions. It is possible to list the set of exceptions that a function may directly or indirectly throw as part of a function declaration. An *exception-specification* can be used as a suffix of a function declarator.

exception-specification:
  `throw (` *type-id-list*$_{opt}$ `)`

type-id-list:
  type-id
  type-id-list `,` *type-id*

For example,

```
void f() throw (X,Y)
{
    // ...
}
```

2 If any declaration or the definition of a function has an *exception-specification*, all declarations and the definition of that function must have an *exception-specification* containing the same set of *type-id*'s.

3 An attempt by a function to throw an exception not in its exception list will cause a call of the function `unexpected()`; see 15.6.2.

4 An implementation may not reject an expression simply because it *might* throw an exception not specified in an *exception-specification* of the function containing the expression; the handling of violations of an *exception-specification* is done at run-time.

5 A function with no *exception-specification* may throw any exception.

6 A function with an empty *exception-specification*, `throw()`, may not throw any exception (unless `unexpected()` itself throws an exception).

7 If a class `X` is in the *type-id-list* of the *exception-specification* of a function, the function may throw an exception object of any class publicly derived from `X`. Similarly, if a pointer to class `Y*` is in the *type-id-list* of the *exception-specification* of a function, the function may throw a pointer to object of any class publicly derived from `Y`.

8 An *exception-specification* is not considered part of a function's type.

15.6 Special Functions

1 The exception handling mechanism relies on two functions, `terminate()` and `unexpected()`, for coping with errors related to the exception handling mechanism itself.

15.6.1 The `terminate()` Function

1 Occasionally, exception handling must be abandoned for less subtle error handling techniques. For example,

- when the exception handling mechanism cannot find a handler for a thrown exception,
- when the exception handling mechanism finds the stack corrupted, or
- when a destructor called during stack unwinding caused by an exception tries to exit using an exception.

In such cases,

```
void terminate();
```

is called; `terminate()` calls the function given on the most recent call of `set_terminate()`:

```
typedef void(*PFV)();
PFV set_terminate(PFV);
```

The previous function given to `set_terminate()` will be the return value; this enables users to implement a stack strategy for using `terminate()`. The default function called by `terminate()` is `abort()`.

2 Selecting a terminate function that does not in fact terminate but tries to return to its caller either with `return` or by throwing an exception is an error.

15.6.2 The `unexpected()` Function

1 If a function with an *exception-specification* throws an exception that is not listed in the *exception-specification*, the function

```
void unexpected();
```

is called; `unexpected()` calls the function given on the most recent call of `set_unexpected()`:

```
typedef void(*PFV)();
PFV set_unexpected(PFV);
```

The previous function given to `set_unexpected()` will be the return value; this enables users to implement a stack strategy for using `unexpected()`. The default function called by `unexpected()` is `terminate()`. Since the default function called by `terminate()` is `abort()`, this leads to immediate and precise detection of the error.

2 The `unexpected()` function may not return, but it may throw an exception. Handlers for this exception will be looked for starting at the call of the function whose *exception-specification* was violated. Thus an *exception-specification* does not guarantee that only the listed classes will be thrown. For example,

```
void  pass_through() { throw; }
void  f(PFV pf) throw()    // f claims to throw no exceptions
{
        (*pf)();           // but the argument function might
}
void  g(PFV pf)
{
        set_unexpected(&pass_through);
        f(pf);
}
```

After the call in `g()` to `set_unexpected()`, `f()` behaves as if it had no *exception-specification* at all. |

15.7 Exceptions and Access

1 The parameter of a catch clause obeys the same access rules as a parameter of the function in which the catch clause occurs.

2 An object may be thrown if it can be copied and destroyed in the context of the function in which the throw occurs.

16

Preprocessing Directives

1 This chapter describes preprocessing in C++. C++ preprocessing, which is based on ANSI C preprocessing, provides macro substitution, conditional compilation, and source file inclusion. In addition, directives are provided to control line numbering in diagnostics and for symbolic debugging, to generate a diagnostic message with a given token sequence, and to perform implementation-dependent actions (the `#pragma` directive). Certain predefined names are available. These facilities are conceptually handled by a preprocessor, which may or may not actually be implemented as a separate process.

16 Preprocessing directives

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.[25]

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

[25] 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).

elif-group:
`# elif` *constant-expression new-line group*$_{opt}$

else-group:
`# else` *new-line group*$_{opt}$

endif-line:
`# endif` *new-line*

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

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;[26] 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.

[26] 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.

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.

4 Prior to evaluation, macro invocations in the list of preprocessing tokens that will become the controlling constant expression are replaced (except for those macro names modified by the `defined` unary operator), just as in normal text. If the token `defined` is generated as a result of this replacement process or use of the `defined` unary operator does not match one of the two specified forms prior to macro replacement, the behavior is undefined. After all replacements due to macro expansion and the `defined` unary operator have been performed, all remaining identifiers are replaced with the pp-number `0`, and then each preprocessing token is converted into a token. The resulting tokens comprise the controlling constant expression which is evaluated according to the rules of 5.19 using arithmetic that has at least the ranges specified in <<<<<<???>>>>>>, except that `int` and `unsigned int` act as if they have the same representation as, respectively, `long` and `unsigned long`. This includes interpreting character constants, which may involve converting escape sequences into execution character set members. Whether the numeric value for these character constants matches the value obtained when an identical character constant occurs in an expression (other than within a `#if` or `#elif` directive) is implementation-defined.[27] 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.[28]

16.2 Source file inclusion

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

---

[27] 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)
```

[28] 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.

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.

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.[29] 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 letters (as defined in <<<<character set section>>>>) followed by a period (`.`) and a single letter. The implementation may ignore the distinctions of alphabetical case and restrict the mapping to six significant characters before the period.

6 A `#include` preprocessing directive may appear in a source file that has been read because of a `#include` directive in another file, up to an implementation-defined nesting limit (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

1 Two replacement lists are identical if and only if the preprocessing tokens in both have the same number, ordering, spelling, and white-space separation, where all white-space separations are considered identical.

2 An identifier currently defined as a macro without use of lparen (an *object-like* macro) may be redefined by another `#define` preprocessing directive provided that the second definition is an object-like macro definition and the two replacement lists are identical.

3 An identifier currently defined as a macro using lparen (a *function-like* macro) may be redefined by another `#define` preprocessing directive provided that the second definition is a function-like macro definition that has the same number and spelling of parameters, and the two replacement lists are identical.

4 The number of arguments in an invocation of a function-like macro shall agree with the number of parameters in the macro definition, and there shall exist a `)` preprocessing token that terminates the invocation.

5 A parameter identifier in a function-like macro shall be uniquely declared within its scope.

6 The identifier immediately following the `define` is called the *macro name*. There is one name space for macro names. Any white-space characters preceding or following the replacement list of preprocessing tokens are not considered part of the replacement list for either form of macro.

[29] Note that adjacent string literals are not concatenated into a single string literal (see the translation phases in 2.1); thus, an expansion that results in two string literals is an invalid directive.

7 If a `#` preprocessing token, followed by an identifier, occurs lexically at the point at which a preprocessing directive could begin, the identifier is not subject to macro replacement.

8 A preprocessing directive of the form

`# define` *identifier replacement-list new-line*

defines an object-like macro that causes each subsequent instance of the macro name[30] 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

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

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 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.

[30] 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.

16.3.3 The ## operator

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

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

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

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

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

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

1 A preprocessing directive of the form

`#` *new-line*

has no effect.

16.8 Predefined macro names

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.

2 The values of the predefined macros (except for `__LINE__` and `__FILE__`) remain constant throughout the translation unit.

3 None of these macro names, nor the identifier `defined`, shall be the subject of a `#define` or a `#undef` preprocessing directive. All predefined macro names shall begin with a leading underscore followed by an uppercase letter or a second underscore.

17

Library

1 The C++ Standard Library consists of classes designed for C++ as well as functions and macros inherited from C.

17 Library

1 The signatures defined in the standard library are reserved to the implementation. The behavior of a C++ program that defines functions with signatures matching any of the reserved signatures is undefined.

> Reentrancy: The intent is to allow the library to be reentrant, despite the implied existence of single, global state information (such as cin, cout, cerr, new-handler, unexpected-function, and terminate-function). Multi-threaded implementations will need to provide the appropriate concurrency interlocks if they support a single, global state. Alternatively, they may provide separate copies for each thread.

17.1 Language support

1 The classes and functions in this section are required to support certain aspects of the C++ language.

17.1.1 Free store `<new>`

1 These functions support the free store management described in 5.3 and 12. The implementation calls `::operator new()` or `::operator new[]()` to allocate storage for objects created by *new-expressions* (5.3.3), and calls `::operator delete()` or `::operator delete[]()` to deallocate the storage for objects in *delete-expression*s (5.3.4).

2 The signatures `::operator new(size_t)`, `::operator new[](size_t)`, `::operator delete(void*)`, and `::operator delete[](void*)` are not reserved. A C++ program may provide at most one definition of each of these functions. Any such functions will replace the default versions. This replacement is global and takes effect upon program startup (3.4). Any parts of the implementation (including other portions of the standard library) that directly or indirectly invoke these functions for free store management will use the supplied replacements. The versions of `::operator new(size_t)`, `::operator new[](size_t)`, `::operator delete(void*)`, and `::operator delete[](void*)` described here are the default versions supplied by an implementation. Any C++ program that replaces any of them with functions having different result semantics causes that program to have undefined behavior.

3 The relationship between these memory management functions and the functions `malloc()`, `calloc()`, `realloc()`, and `free()` (17.4.10.3) is unspecified.

17.1.1.1 `operator new()` and `operator new[]()`

```
void* operator new(size_t) throw(xalloc);
void* operator new[](size_t) throw(xalloc);
```

1 When a non-array object or an array is created with a *new-expression* the implementation uses `::operator new()` or `::operator new[]()` (respectively) to obtain the store needed.

2 For array allocation, the implementation calculates the storage required to hold the array and calls `::operator new[]()` with the resulting size or a larger size.

Result semantics:

3 If successful, returns a pointer to allocated storage. Otherwise, throws an `xalloc` exception (17.1.3.3). Any other action is undefined.

> Since the exception will be propagated through a new-expression, it changes the semantics of such expressions. Error handling now relies on a catch clause, not a null pointer result. Returning a null pointer is undefined, but allowed to ease transition from earlier language implementations.

4 The order and contiguity of storage allocated by successive calls to `::operator new()` or `::operator new[]()` is unspecified. The initial value of this storage is unspecified. The pointer returned is suitably aligned so that it may be assigned to a pointer of any type and then used to initialize and access such an object or an array of such objects in the storage allocated (until the storage is explicitly deallocated by a call to the corresponding deallocation function (_expr.free_, 17.1.1.2). Each such allocation shall yield a pointer to storage disjoint from any other allocated storage. The pointer returned points to the start (lowest byte address) of the allocated storage. If the size of the space is requested is zero, the value returned shall be a pointer differerent from the address of any other currently allocated storage. Repeated such calls return distinct non-null pointers (5.3.3). The result of dereferencing a pointer returned from a request for zero size is undefined.

> The wording for the above 8 sentences was adapted from §17.4.10.3. 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.

Description of default implementation:

5

1. Attempts to allocate storage to hold at least the amount of storage requested.

   > Note that the actual size may be larger than the requested size, due to alignment or other requirements. Some implementations convert a request for zero bytes into a request for 1 byte.

   If successful, returns the address of storage allocated.

2. If unsuccessful, checks the current *new-handler* (17.1.1.4): If there is no *new-handler* installed, the result is implementation defined. Otherwise, calls the new-handler.

The installed new-handler may throw an xalloc exception.

3. If the call to the *new-handler* returns, repeats the attempt to allocate memory (go to step 1 above).

6 The default `operator new[]()` calls `operator new()`. Thus if `operator new()` is replaced without replacing `operator new[]()`, the replacement function will be used for both non-array and array allocations.

17.1.1.2 `operator delete()` and `operator delete[]()`

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

1 The *delete-expression* (5.3.4) destroys an object created by the *new-expression*, and (implicitly) calls the deallocation function, `operator delete()` for non-arrays, or `operator delete[]()` for arrays (12.5). A null pointer is a valid argument but has no effect. Otherwise, the effect of the deallocation function is to reclaim the storage pointed to by its argument. The argument then becomes invalid.

An invalid pointer contains an unusable value — it cannot even be used in an expression. This still needs work.

Result semantics:

2 If the argument is a non-null pointer

- The value of a pointer that refers to deallocated space is indeterminate.
- The effect of dereferencing a pointer already deleted is undefined.
- The effect of applying the deallocation function to a pointer already deleted is undefined.

An implementation could (should) throw an exception if it can detect these conditions.

Description of default implementation:

3 Deallocates the storage referenced by the pointer. The storage may be available for further allocation. The argument to the default `::operator delete()` must be a pointer returned by the default `::operator new()` and the argument to the default `::operator delete[]()` must be a pointer returned by the default `::operator new[]()` (17.1.1.1). Applying the default `::operator delete()` or `::operator delete[]()` to a null pointer has no effect.

4 The default `::operator delete[]()` calls `::operator delete()` so that if just `::operator delete()` is replaced, the replacement will be used for both array and non-array deletion.

17.1.1.3 *placement* `operator new()`

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

1 This function is reserved.

> This second form of `::operator new()` is one of an unbounded set of overloaded functions for use with placement expressions.

The placement version of operator new() returns its second argument as its result:

```
void* operator new(size_t, void* p) { return p; }
```

17.1.1.4 *new-handler* function

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

1 When `::operator new()` (17.1.1.1) cannot allocate storage to satisfy a request, it calls the currently installed *new-handler* function. A C++ program may install *new-handler* functions via calls to `set_new_handler()` (17.1.1.5).

Result semantics:

2 A new-handler function shall either

1. return after deallocating some currently-allocated storage, or
2 throw an `xalloc` exception or an exception derived from `xalloc` (17.1.3.3), or
3. call `abort()` (17.4.10.4.1) or `exit()` (17.4.10.4.3).

3 Any C++ program that installs a *new-handler* having different result semantics causes that program to have undefined behavior.

> In particular, a *new-handler* that returns to the default `::operator new()` (17.1.1.1) without freeing any storage will cause an infinite loop.

Description of default implementation:

4 The default *new-handler* function throws an `xalloc` exception (17.1.3.3):

```
void new_handler() { throw xalloc; }
```

> Earlier implementations provided no default new-handler, causing new expressions to return null when the memory request could not be met. C++ programs that used the result of new expressions without checking the result were erroneous, while those that checked were correct. Providing a default new-handler that throws an exception "fixes" the erroneous programs (by detecting all memory exhaustion conditions), but breaks the previously correct ones (by requiring them to do the checking with a catch-clause). The old behavior can be restored by calling `set_new_handler(0)`.

17.1.1.5 set_new_handler()

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

1 Installs the function given as argument as the current *new-handler* (17.1.1.4). Returns the previous function given to set_new_handler().

> This enables callers to implement a stack strategy for using *new-handler*s. Note that the *new-handler* function is anonymous — it cannot be called directly. To obtain the current new-handler, call set_new_handler() with a known argument (for example, zero), save the result, and call set_new_handler() again with the result to re-set the new-handler back to what it was.

17.1.2 Type identification <type_info>

1 Class Type_info is declared in <Type_info.h> like this:

```
class Type_info {
    // implementation dependent representation
private:
    Type_info(const Type_info&);               // objects cannot
    Type_info& operator=(const Type_info&);    // be copied by users
public:
    virtual ~Type_info();                      // is polymorphic

    int operator==(const Type_info&) const;    // can be compared
    int operator!=(const Type_info&) const;
    int before(const Type_info&);              // define order among
                                               // Type_info objects

    const char* name() const;                  // get the type name
};
```

,P The ordering defined by before is complete and valid only for the duration of the execution of program. There is no guaranteed relation between the ordering defined by before and inheritance relationships.

17.1.3 Exceptions

1 These functions support the Exception Handling described in Chapter 15.

17.1.3.1 Abnormal termination

1 These functions allow C++ programs to control how the implementation responds to faults in the exception handling mechanism.

17.1.3.1.1 terminate()

```
void terminate();
```

1 This function is called when exception handling must be abandoned (15.6). For example,

- when the exception handling mechanism cannot find a handler for a thrown exception,
- when the exception handling mechanism finds the stack corrupted, or
- when a destructor called during stack unwinding caused by an exception tries to exit using an exception.

2 `terminate()` calls the current *terminate-function* (17.1.3.1.2).

17.1.3.1.2 *terminate-function*

```
typedef void (*terminate_function)();
```

1 A C++ program may install *terminate-function*s via calls to `set_terminate()` (17.1.3.1.3).

Result semantics:

2 This function shall not return. It may call (17.4.10.4.1) or (17.4.10.4.3). Any other action is undefined.

> It may re-start the application process, invoke some other last-chance disaster-recovery mechanism, or take some other action that cannot be specified in the standard.

Description of default implementation:

3 The default terminate-function is `abort()` (17.4.10.4.1).

17.1.3.1.3 `set_terminate()`

```
terminate_function set_terminate( terminate_function );
```

1 Installs the function given as an argument as the current *terminate-function* (17.1.3.1.2). Returns the previous function given to `set_terminate()`.

> This enables callers to implement a stack strategy for using *terminate-functions*.

17.1.3.2 Violating *exception-specification*s

1 These functions allow C++ programs to control how the implementation responds to inconsistencies between declared *exception-specification*s and actual exceptions detected at runtime.

17.1.3.2.1 `unexpected()`

```
void unexpected()
```

The implementation calls `unexpected()` if a function with an *exception-specification* throws an exception that is not listed in the *exception-specification*.

1 `unexpected()` calls the current *unexpected-function* (17.1.3.2.2).

17.1.3.2.2 *unexpected-function*

```
typedef void (*unexpected_function)();
```

A C++ program may install *unexpected-function*s via calls to `set_unexpected()` (17.1.3.2.3).

Result semantics:

1 This function shall not return. It may call `terminate()` (17.1.3.1.1), `abort()` (17.4.10.4.1), or `exit(` (_lib.exot_). Any other action is undefined.

Since these kinds of errors usually indicate design problems that need to be fixed, the intent is to make them easy to detect.

Description of default implementation:

2 The default *unexpected-function* is `terminate()` (17.1.3.1.1).

17.1.3.2.3 `set_unexpected()`

```
unexpected_function set_unexpected( unexpected_function );
```

1 Installs the function given as an argument as the current *unexpected-function* (17.1.3.2.2). Returns the previous function given to `set_unexpected()`.

This enables callers to implement a stack strategy for using *unexpected-function*s.

17.1.3.3 Predefined exceptions

1 These classes define the exceptions reported by various functions in the standard library.

17.1.3.3.1 `xmsg` **exception**

```
class xmsg {
public:
    xmsg(const string& msg);

    string why() const;
    void raise() throw(xmsg);
private:
    // implementation-defined
};
```

The intent of the xmsg exception class was to allow programs to catch all exceptions in the library:

1 For example,

```
#include <stdlib.h>
#include <iostream>

int main(int argc, char** argv)
{
    try {
        real_main(argc,argv);
        return EXIT_SUCCESS;
    } catch(xmsg& m) {
        cerr << "exiting because of exception: " << m.why() << endl;
        return EXIT_FAILURE;
    }
}
```

`x.why()` is the string used to construct an `xmsg x`. That is: `xmsg(s).why()~==~s`.

The absence of a default constructor means that every xmsg must contain a meaningful (:-)) message.

`xmsg::raise()` is defined by:

```
void xmsg::raise() throw(xmsg) { throw *this; }
```

> xmsg::raise() adds no functionality but is included as a convenient hook for debugging. Shouldn't it be virtual?

17.1.3.3.2 `xalloc` exception

```
class xalloc : public xmsg {
public:
    xalloc(const string& msg, size_t requested_size);

    size_t requested() const;
    void raise() throw(xalloc);
private:
    // implementation-defined
};
```

1 An `xalloc` exception can be thrown by the *new-handler* (17.1.1.4) when it cannot find storage to allocate.

> The standard does not define the form of an xalloc error message. The following might be plausible (subject to locale settings): `msg + ": Insufficient space to allocate " + int_to_string(size) + " bytes"`. However, since the xalloc exception is going to be thrown when the system runs out of space, the space for constructing and throwing the exception must exist. This implies the error message cannot rely on a string catenation operation that attempts to allocate storage. A plausible implementation would be an allocator that holds back enough storage, such as a static instance of an xalloc object.

17.1.3.3.3 `bad_cast` exception

```
class Bad_cast : public xmsg {
public:
    Bad_cast(const string& msg, /* ??? */);

    // ???
private:
    // implementation-defined
};
```

1 A `Bad_cast` exception can be thrown by a `dynamic_cast` expression (5.2.6).

17.2 The string class

17.3 Input/output

17.4 C library

17.4.1 Introduction

17.4.1.1 Definitions of terms

1 A *null-terminated character sequence* (NTCS)[31] is a contiguous sequence of characters terminated by and

[31] In the ISO C Standard, a NTCS is called a "string." This Standard uses "NTCS" to avoid confusion with the String class (17.2).

including the first null character. A “pointer to” a NTCS is a pointer to its initial (lowest addressed) character. The “length” of a NTCS is the number of characters preceding the null character and its “value” is the sequence of the values of the contained characters, in order.

2 A *letter* is a printing character in the execution character set corresponding to any of the 52 required lowercase and uppercase letters (17.4.3) in the source character set, listed in ???.

3 The *decimal-point character* is the character used by functions that convert floating-point numbers to or from character sequences to denote the beginning of the fractional part of such character sequences.[32] It is represented in the text and examples by a period, but may be changed by the `setlocale` function (17.4.4).

17.4.1.2 Standard headers

1 Each library function is declared in a *header*,[33] whose contents are made available by the `#include` preprocessing directive. The header declares a set of related functions, plus any necessary types and additional macros needed to facilitate their use.

2 The standard headers are

```
<assert.h>          <locale.h>          <stddef.h>
<ctype.h>           <math.h>            <stdio.h>
<errno.h>           <setjmp.h>          <stdlib.h>
<float.h>           <signal.h>          <string.h>
<limits.h>          <stdarg.h>          <time.h>
```

3 If a file with the same name as one of the above < and > delimited sequences, not provided as part of the implementation, is placed in any of the standard places for a source file to be included, the behavior is undefined.

4 Headers may be included in any order; each may be included more than once in a given scope, with no effect different from being included only once, except that the effect of including `<assert.h>` depends on the definition of `NDEBUG`. A header shall be included[34] outside of any external declaration or definition, and it shall first be included before the first reference to any of the functions or objects it declares, or to any of the types or macros it defines. However, if the identifier is declared or defined in more than one header, the second and subsequent associated headers may be included after the initial reference to the identifier. The program shall not have any macros with names lexically identical to keywords currently defined prior to the inclusion.

17.4.1.2.1 Reserved identifiers

1 Each header declares or defines all identifiers listed in its associated section, and optionally declares or defines identifiers listed in its associated future library directions section and identifiers which are always reserved either for any use or for use as file scope identifiers.

- All identifiers that begin with an underscore and either an uppercase letter or another underscore are always reserved for any use.
- All identifiers having two consecutive underscores are always reserved for use as identifiers with both `extern "C++"` and `extern "C"` linkages.
- All identifiers that begin with an underscore are always reserved for use as identifiers with file scope in both the ordinary identifier and tag name spaces.
- Each macro name listed in any of the following sections (including the future library directions) is reserved for any use if any of its associated headers is included.

---

[32] The functions that make use of the decimal-point character are `localeconv`, `fprintf`, `fscanf`, `printf`, `scanf`, `sprintf`, `sscanf`, `vfprintf`, `vprintf`, `vsprintf`, `atof`, and `strtod`.

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

[34] ISO C Standard says “If used, a header shall be...” This standard requires a header to be included if any of its contents are used.

- All identifiers with external linkage in any of the following sections (including the future library directions) are always reserved for use as identifiers with `extern "C"` linkage.[35]

- Each identifier with file scope listed in any of the following sections (including the future library directions) is reserved for use as an identifier with file scope.

- Each function signature listed in any of the following sections (including the future library directions) is reserved for use with both `extern "C++"` and `extern "C"` linkages as a function signature with file scope in the same name space if any of its associated headers is included.[36]

2 No other identifiers are reserved. If the program declares or defines an identifier with the same name as an identifier reserved in that context (other than as allowed by 17.4.1.6), the behavior is undefined.[37]

17.4.1.3 Errors `<errno.h>`

1 The header `<errno.h>` defines several macros, all relating to the reporting of error conditions.

2 *[This section is the same as \(sc7.1.3 of the ISO C Standard.]*

17.4.1.4 Limits `<float.h>` and `<limits.h>`

1 The headers `<float.h>` and `<limits.h>` define several macros that expand to various limits and parameters.

2 *[This section is the same as \(sc2.2.4.2 (??? this is the ANSI paragraph) of the ISO C Standard.]*

17.4.1.5 Common definitions `<stddef.h>`

1 The following types and macros are defined in the standard header `<stddef.h>`. Some are also defined in other headers, as noted in their respective sections.

2 The types are

```
ptrdiff_t
```

which is the signed integral type of the result of subtracting two pointers;

```
size_t
```

which is the unsigned integral type of the result of the `sizeof` operator; and

```
wchar_t
```

which is an integral type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locales (17.4.4); the null character shall have the code value zero and each member of the basic character set defined in <2.2.1> shall have a code value equal to its value when used as the lone character in an integer character constant.

Is this obsolete now that `wchar_t` is a type on its own and a reserved word?

3 The macros are

```
NULL
```

which expands to an implementation-defined C++ null pointer constant; and

`offsetof(`*type*`,` *member-designator*`)`

which expands to an integral constant expression that has type `size_t`, the value of which is the offset in bytes, to the structure member (designated by from the beginning of its C-style structure (designated by

[35] The list of reserved identifiers with external linkage includes `errno`, `setjmp`, and `va_end`.

[36] Class member functions may duplicate signatures of listed functions because they have class scope.

[37] Since macro names are replaced whenever found, independent of scope and name space, macro names matching any of the reserved identifier names must not be defined if an associated header, if any, is included.

Is a C-style structure the same as a POD-struct?

The *member-designator* shall be such that given

static *type* t;

then the expression &(t.*member-designator*) evaluates to an address constant. (If the specified member is a bit-field, the behavior is undefined.)

4 By *C-style structure* is meant a structure defined with either struct or union and whose definition is legal in Standard C. That is, it has no base classes, no member functions, and no access modifiers. The result of applying offsetof to a structure with C++ features is undefined.

This is an issue of form, not behavior.

17.4.1.6 Use of library functions

1 Each of the following statements applies unless explicitly stated otherwise in the detailed descriptions that follow. If an argument to a function has an invalid value (such as a value outside the domain of the function, or a pointer outside the address space of the program, or a null pointer), the behavior is undefined. If a function parameter is described as being an array, the pointer actually passed to the function shall have a value such that all address computations and accesses to objects (that would be valid if the pointer did point to the first element of such an array) are in fact valid. Any function declared in a header must be declared so as to allow it to be overloaded by use of another signature.[38] Those function-like macros described in the following sections may be invoked in an expression anywhere a function with a compatible return type could be called. All object-like macros listed as expanding to integral constant expressions shall additionally be suitable for use in #if preprocessing directives.

2 No library function shall be declared explicitly in a user program, but instead its associated header shall be included if it is to be used.[39] Furthermore, proper prototypes shall be supplied in the appropriate header for each library function.

17.4.2 Diagnostics <assert.h>

1 The header <assert.h> defines the assert macro and refers to another macro,

NDEBUG

which is *not* defined by <assert.h>. The assert macro need not be implemented as a function, and need not be overloadable if it is implemented as a function.

2 *[The remainder of this section is the same as section 7.2 of the ISO C Standard.]*

17.4.3 Character handling <ctype.h>

1 *[This section is the same as section 7.3 of the ISO C Standard.]*

17.4.4 Localization <locale.h>

1 *[This section is the same as section 7.4 of the ISO C Standard.]*

17.4.5 Mathematics <math.h>

1 *[This section is the same as section 7.5 of the ISO C Standard.]*

17.4.6 Nonlocal jumps <setjmp.h>

[38] This means that the library functions must not be implemented as macros, although they may be implemented as inline functions.
[39] This allows the implementation to supply a function with either C or C++ linkage.

1 The header <setjmp.h> defines the macro setjmp, and declares one function and one type, for bypassing the normal function call and return discipline.[40]

2 The type declared is

```
jmp_buf
```

which is an array type suitable for holding the information needed to restore a calling environment.

3 It is unspecified whether setjmp is a macro or an identifier declared with external linkage. If a macro definition is suppressed in order to access an actual function, or a program defines an external identifier with the name setjmp, the behavior is undefined.

17.4.6.1 Save calling environment

17.4.6.1.1 The setjmp macro

```
#include <setjmp.h>
int setjmp(jmp_buf env);
```

1 The setjmp macro saves its calling environment in its jmp_buf argument for later use by the longjmp function.

2 If the return is from a direct invocation, the setjmp macro returns the value zero. If the return is from a call to the longjmp function, the setjmp macro returns a nonzero value.

3 An invocation of the setjmp macro shall appear only in one of the following contexts:

- the entire controlling expression of a selection or iteration statement;
- one operand of a relational or equality operator with the other operand an integral constant expression, with the resulting expression being the entire controlling expression of a selection or iteration statement;
- the operand of a unary ! operator with the resulting expression being the entire controlling expression of a selection or iteration statement; or
- the entire expression of an expression statement (possibly cast to void).

17.4.6.2 Restore calling environment

17.4.6.2.1 The longjmp function

```
#include <setjmp.h>
void longjmp(jmp_buf env, int val);
```

1 The longjmp function restores the environment, saved by the most recent invocation of the setjmp macro in the same invocation of the program, with the corresponding jmp_buf argument. If there has been no such invocation, or if the function containing the invocation of the setjmp macro has terminated execution[41] in the interim, the behavior is undefined.

2 All accessible objects have values as of the time longjmp was called, except that the values of objects of automatic storage duration that are local to the function containing the invocation of the corresponding setjmp macro that do not have volatile-qualified type and have been changed between the setjmp invocation and longjmp call are indeterminate.

3 As it bypasses the usual function call and return mechanisms, the longjmp function shall execute correctly in contexts of interrupts, signals and any of their associated functions. However, if the longjmp function is invoked from a nested signal handler (that is, from a function invoked as a result of a signal

[40] These functions are useful for dealing with unusual conditions encountered in a low-level function of a program.

[41] For example, by executing a return statement or because another longjmp call has caused a transfer to a setjmp invocation in a function earlier in the set of nested calls.

raised during the handling of another signal), the behavior is undefined.

4 If any automatic objects would have been destroyed due to an exception transferring control to the same function as the longjmp, the results of the longjmp are undefined.

5 After longjmp is completed, program execution continues as if the corresponding invocation of the setjmp macro had just returned the value specified by val. The longjmp function cannot cause the setjmp macro to return the value zero; if val is zero, the setjmp macro returns the value 1.

17.4.7 Signal handling <signal.h>

1 *[This section is the same as section 7.7 of the ISO C Standard.]*

17.4.8 Variable arguments <stdarg.h>

1 *[This section is the same as section 7.8 of the ISO C Standard.]*

17.4.9 Input/output <stdio.h>

1 *[This section is the same as section 7.9 of the ISO C Standard.]*

17.4.10 General utilities <stdlib.h>

1 The header <stdlib.h> declares three types and several functions of general utility, and defines several macros.[42]

2 The types declared are size_t (described in 17.4.1.5),

    div_t

which is a structure type that is the type of the value returned by the div function, and

    ldiv_t

which is a structure type that is the type of the value returned by the ldiv function.

3 The macros defined are NULL (described in 17.4.1.5);

    EXIT_FAILURE

and

    EXIT_SUCCESS

which expand to integral expressions that may be used as the argument to the exit function to return unsuccessful or successful termination status, respectively, to the host environment;

    RAND_MAX

which expands to an integral constant expression, the value of which is the maximum value returned by the rand function; and

    MB_CUR_MAX

which expands to a positive integer expression whose value is the maximum number of bytes in a multibyte character for the extended character set specified by the current locale (category LC_CTYPE), and whose value is never greater than MB_LEN_MAX.

17.4.10.1 NTCS conversion functions

1 *[This section is the same as section 7.10.1 of the ISO C Standard.]*

17.4.10.2 Pseudo-random sequence generation functions

1 *[This section is the same as section 7.10.2 of the ISO C Standard.]*

[42] See future library directions (17.4.13).

17.4.10.3 Memory management functions

1 The order and contiguity of storage allocated by successive calls to the `calloc`, `malloc`, and `realloc` functions is unspecified. The pointer returned if the allocation succeeds is suitably aligned so that it may be assigned to a pointer to any type of object and then used to access such an object or an array of such objects in the space allocated (until the space is explicitly freed or reallocated). Each such allocation shall yield a pointer to an object disjoint from any other object. The pointer returned points to the start (lowest byte address) of the allocated space. If the space cannot be allocated, a null pointer is returned. If the size of the space requested is zero, the behavior is implementation-defined; the value returned shall be either a null pointer or a unique pointer. The value of a pointer that refers to freed space is indeterminate.

2 The relationship between these memory management functions and the C++ `operator new` and `operator delete` is unspecified.[43]

3 *[The remainder of this section is the same as section 7.10.3 of the ISO C Standard.]*

17.4.10.4 Communication with the environment

17.4.10.4.1 The `abort` Function

1 *[This section is the same*[44] *as section 7.10.4.1 of the ISO C Standard.]*

17.4.10.4.2 The `atexit` function

1 *[This section is the same as section 7.10.4.2 of the ISO C Standard.]*

17.4.10.4.3 The `exit` function

```
#include <stdlib.h>
void exit(int status);
```

1 The `exit` function causes normal program termination to occur. If more than one call to the `exit` function is executed by a program, the behavior is undefined.

2 First, all functions registered by the `atexit` function are called, in the reverse order of their registration.[45]

3 Next, all static objects are destroyed in the reverse order of their construction. (Non-static local objects are not destroyed as a result of calling `exit`.)[46]

4 Next, all open C stdio streams (in the sense of ISO Standard section 7.9.2) with unwritten buffered data are flushed, all open C stdio streams are closed, and all files created by the `tmpfile` function are removed.[47]

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

6 The `exit` function cannot return to its caller.

17.4.10.4.4 The `getenv` function

1 *[This section is the same as section 7.10.4.4 of the ISO C Standard.]*

17.4.10.4.5 The `system` function

1 *[This section is the same as section 7.10.4.5 of the ISO C Standard.]*

---

[43] For example, either of `operator new` and `malloc` might be written in terms of the other. On the other hand, they need not have anything in common or even maintain storage in the same address space.

[44] No destructors are invoked as a result of a call to `abort`. In particular, constructed local and static objects remain undestroyed.

[45] Each function is called as many times as it was registered.

[46] To achieve the effect of destroying automatic objects (other than those declared at the outermost level of `main`), throw an exception which is caught in `main`. The stack will unwind, destroying automatic objects. At the place where the exception is caught, call `exit`.

[47] The standard C++ iostreams will have already been flushed and closed by the previous step.

17.4.10.5 Searching and sorting utilities

1 *[This section is the same as section 7.10.5 of the ISO C Standard.]*

17.4.10.6 Integer arithmetic functions

1 *[This section is the same as section 7.10.6 of the ISO C Standard.]*

17.4.10.7 Multibyte character functions

1 *[This section is the same as section 7.10.7 of the ISO C Standard.]*

17.4.10.8 Multibyte NTCS functions

1 *[This section is the same as section 7.10.8 of the ISO C Standard.]*

17.4.11 NTCS handling <string.h>

17.4.11.1 NTCS function conventions

1 The header <string.h> declares one type and several functions, and defines one macro useful for manipulating arrays of character type and other objects treated as arrays of character type.[48] The type is size_t and the macro is NULL (both described in 17.4.1.5). Various methods are used for determining the lengths of the arrays, but in all cases a char * or void * argument points to the initial (lowest addressed) character of the array. If an array is accessed beyond the end of an object, the behavior is undefined.

17.4.11.2 Copying functions

1 *[This section is the same as section 7.11.2 of the ISO C Standard, containing the* memcpy*,* memmove*,* strcpy*, and* strncpy *functions.]*

17.4.11.3 Concatenation functions

1 *[This section is the same as section 7.11.3 of the ISO C Standard, containing the* strcat *and* strncat *functions.]*

17.4.11.4 Comparison functions

1 *[This section is the same as section 7.11.4 of the ISO C Standard.]*

17.4.11.5 Search functions

17.4.11.5.1 The memchr functions

```
#include <string.h>
const void *memchr(const void *s, int c, size_t n);
      void *memchr(      void *s, int c, size_t n);
```

1 The memchr functions[49] locate the first occurrence of c (converted to an unsigned char) in the initial n characters (each interpreted as unsigned char) of the object pointed to by s.

2 The memchr functions return a pointer to the located character, or a null pointer if the character does not occur in the object.

17.4.11.5.2 The strchr functions

```
#include <string.h>
const char *strchr(const char *s, int c);
      char *strchr(      char *s, int c);
```

[48] See future library directions (17.4.13).

[49] The ISO C library contains a single function which returns a non-const pointer into its const first parameter.

1 The strchr functions[50] locate the first occurrence of c (converted to a char) in the NTCS pointed to by s. The terminating null character is considered to be part of the NTCS.

2 The strchr functions return a pointer to the located character, or a null pointer if the character does not occur in the NTCS.

17.4.11.5.3 The strcspn function

1 *[This section is the same as section 7.11.5.3 of the ISO C Standard.]*

17.4.11.5.4 The strpbrk functions

```
#include <string.h>
const char *strpbrk(const char *s1, const char *s2);
      char *strpbrk(      char *s1, const char *s2);
```

1 The strpbrk functions[51] locate the first occurrence in the NTCS pointed to by s1 of any character from the NTCS pointed to by s2.

2 The strpbrk functions return a pointer to the character, or a null pointer if no character from s2 occurs in s1.

17.4.11.5.5 The strrchr function

```
#include <string.h>
const char *strrchr(const char *s, int c);
      char *strrchr(      char *s, int c);
```

1 The strrchr functions[52] locate the last occurrence of c (converted to a char) in the NTCS pointed to by s. The terminating null character is considered to be part of the NTCS.

2 The strrchr functions return a pointer to the character, or a null pointer if c does not occur in the NTCS.

17.4.11.5.6 The strspn function

[This section is the same as section 7.11.5.6 of the ISO C Standard.]

17.4.11.5.7 The strstr functions

```
#include <string.h>
const char *strstr(const char *s1, const char *s2);
      char *strstr(      char *s1, const char *s2);
```

1 The strstr functions[53] locate the first occurrence in the NTCS pointed to by s1 of the sequence of characters (excluding the terminating null character) in the NTCS pointed to by s2

2 The strstr functions return a pointer to the located NTCS, or a null pointer if the NTCS is not found. If s2 points to a NTCS with zero length, the function returns s1.

17.4.11.5.8 The strtok function

1 *[This section is the same as section 7.11.5.8 of the ISO C Standard.]*

17.4.11.6 Miscellaneous functions

1 *[This section is the same as section 7.11.6 of the ISO C Standard.]*

17.4.12 Date and time <time.h>

[50] The ISO C library contains a single function which returns a non-const pointer into its const first parameter.
[51] The ISO C library contains a single function which returns a non-const pointer into its const first parameter.
[52] The ISO C library contains a single function which returns a non-const pointer into its const first parameter.
[53] The ISO C library contains a single function which returns a non-const pointer into its const first parameter.

1 *[This section is the same as section 7.12 of the ISO C Standard.]*

17.4.13 Future c library directions

1 *[This section is the same as section 7.13 of the ISO C Standard.]*

17.5 Future library directions

18

Grammar Summary

1 This chapter provides a summary of the C++ grammar.

18 Appendix A: Grammar summary

1 This appendix is not part of the C++ reference manual proper and does not define C++ language features.

2 This summary of C++ syntax is intended to be an aid to comprehension. It is not an exact statement of the language. In particular, the grammar described here accepts a superset of valid C++ constructs. Disambiguation rules (6.8, 7.1, 10.1.1) must be applied to distinguish expressions from declarations. Further, access control, ambiguity, and type rules must be used to weed out syntactically valid but meaningless constructs.

18.1 Keywords

1 New context-dependent keywords are introduced into a program by `typedef` (7.1.3), class (9), enumeration (7.2), and `template` (14) declarations.

class-name:
    identifier
    template-class-id

enum-name:
    identifier

typedef-name:
    identifier

Note that a *typedef-name* naming a class is also a *class-name* (9.1).

18.2 Expressions

1

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

direct-new-declarator:
    direct-new-declarator$_{opt}$ `[` *expression* `]`

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

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

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-name* `>` `(` *expression* `)`
    `typeid` `(` *expression* `)`
    `typeid` `(` *type-name* `)`

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

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

id-expression:
  identifier
  operator-function-id
  conversion-function-id
  `~` *class-name*
  qualified-id

qualified-id:
  nested-class-specifier `::` *id-expression*

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

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

decimal-literal:
  nonzero-digit
  decimal-literal digit

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

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

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

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

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

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

unsigned-suffix: one of
  `u U`

long-suffix: one of
  `l L`

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

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

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

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

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

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

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

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

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

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

sign: one of
  `+ -`

digit-sequence:
  digit
  digit-sequence digit

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

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

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

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

2 *declaration:*
decl-specifier-seq$_{opt}$ *init-declarator-list*$_{opt}$ `;`
asm-definition
function-definition
template-declaration
linkage-specification

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

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

storage-class-specifier:
`auto`
`register`
`static`
`extern`

function-specifier:
`inline`
`virtual`

type-specifier:
simple-type-specifier
class-specifier
enum-specifier
elaborated-type-specifier
`::` *typedef-name*
`const`
`volatile`

simple-type-specifier:
  qualified-class-specifier
  qualified-type-specifier
  `char`
  `wchar_t`
  `short`
  `int`
  `long`
  `signed`
  `unsigned`
  `float`
  `double`
  `void`

elaborated-type-specifier:
  class-key identifier
  class-key qualified-class-specifier `::` *identifier*
  `enum` *identifier*
  `enum` *qualified-class-specifier* `::` *identifier*

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

qualified-type-specifier:
  typedef-name
  class-name `::` *qualified-type-specifier*

qualified-class-specifier:
  nested-class-specifier
  `::` *nested-class-specifier*

nested-class-specifier:
  class-name
  class-name `::` *nested-class-specifier*

enum-specifier:
  `enum` *identifier*$_{opt}$ `{` *enumerator-list*$_{opt}$ `}`

enumerator-list:
  enumerator
  enumerator-list `,` *enumerator*

enumerator:
  identifier
  identifier `=` *constant-expression*

constant-expression:
  conditional-expression

linkage-specification:
  `extern` *string-literal* `{` *declaration-seq*$_{opt}$ `}`
  `extern` *string-literal declaration*

declaration-seq:
  declaration
  declaration-seq declaration

asm-definition:
  `asm ( `*string-literal*` ) ;`

18.3 Declarators

1 *init-declarator-list:*
  init-declarator
  init-declarator-list `,` *init-declarator*

init-declarator:
  declarator *initializer*$_{opt}$

declarator:
  direct-declarator
  ptr-operator declarator

direct-declarator:
  declarator-id
  direct-declarator `(` *parameter-declaration-clause* `)` *cv-qualifier-seq*$_{opt}$ *exception-specification*$_{opt}$
  direct-declarator `[` *constant-expression*$_{opt}$ `]`
  `(` *declarator* `)`

ptr-operator:
  `*` *cv-qualifier-seq*$_{opt}$
  `&` *cv-qualifier-seq*$_{opt}$
  qualified-class-specifier `:: *` *cv-qualifier-seq*$_{opt}$

cv-qualifier-seq:
  cv-qualifier *cv-qualifier-seq*$_{opt}$

cv-qualifier:
  `const`
  `volatile`

declarator-id:
  id-expression
  qualified-type-specifier

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

type-specifier-seq:
  type-specifier type-specifier-seq$_{opt}$

abstract-declarator:
  ptr-operator abstract-declarator$_{opt}$
  direct-abstract-declarator

direct-abstract-declarator:
  direct-abstract-declarator$_{opt}$ `(` *parameter-declaration-clause* `)` *cv-qualifier-seq*$_{opt}$ *exception-specification*$_{opt}$
  direct-abstract-declarator$_{opt}$ `[` *constant-expression*$_{opt}$ `]`
  `(` *abstract-declarator* `)`

parameter-declaration-clause:
  parameter-declaration-list$_{opt}$ `...`$_{opt}$
  parameter-declaration-list `, ...`

parameter-declaration-list:
parameter-declaration
parameter-declaration-list **,** *parameter-declaration*

parameter-declaration:
decl-specifier-seq declarator
decl-specifier-seq declarator = *expression*
decl-specifier-seq abstract-declarator$_{opt}$
decl-specifier-seq abstract-declarator$_{opt}$ = *expression*

function-definition:
decl-specifier-seq$_{opt}$ *declarator ctor-initializer*$_{opt}$ *function-body*

function-body:
compound-statement

initializer:
= *initializer-clause*
(*expression-list*)

initializer-clause:
assignment-expression
{ *initializer-list* **,**$_{opt}$ }

initializer-list:
initializer-clause
initializer-list **,** *initializer-clause*

18.4 Class declarations

1 *class-specifier:*
class-head { *member-specification*$_{opt}$ }

class-head:
class-key identifier$_{opt}$ *base-clause*$_{opt}$
class-key nested-class-specifier base-clause$_{opt}$

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 **;**

member-declarator-list:
member-declarator
member-declarator-list **,** *member-declarator*

member-declarator:
declarator pure-specifier$_{opt}$
identifier$_{opt}$ **:** *constant-expression*

pure-specifier:
= 0

base-clause:
  `:` *base-specifier-list*

base-specifier-list:
  base-specifier
  base-specifier-list `,` *base-specifier*

base-specifier:
  qualified-class-specifier
  `virtual` *access-specifier*$_{opt}$ *qualified-class-specifier*
  access-specifier `virtual`$_{opt}$ *qualified-class-specifier*

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

conversion-function-id:
  `operator` *conversion-type-id*

conversion-type-id:
  type-specifier-seq ptr-operator$_{opt}$

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

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

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

operator-function-id:
  `operator` *operator*

operator: one of

```
new  delete    new[]     delete[]
+    -    *    /    %    ^    &    |    ~
!    =    <    >    +=   -=   *=   /=   %=
^=   &=   |=   <<   >>   >>=  <<=  ==   !=
<=   >=   &&   ||   ++   --   ,    ->*  ->
()   []
```

18.5 Statements

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

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

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

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

statement-seq:
statement
statement-seq statement

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

condition:
expression
type-specifier declarator = *expression*

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

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

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

declaration-statement:
declaration

18.6 Preprocessor

1 #define *identifier token-string*
#define *identifier*(*identifier* , ... , *identifier*) *token-string*

#include "*filename*"
#include <*filename*>

#line *constant* "*filename*"$_{opt}$
#undef *identifier*

conditional:
if-part elif-parts$_{opt}$ *else-part*$_{opt}$ *endif-line*

if-part:
if-line text

if-line:
`# if` *constant-expression*
`# ifdef` *identifier*
`# ifndef` *identifier*

elif-parts:
elif-line text
elif-parts elif-line text

elif-line:
`# elif` *constant-expression*

else-part:
else-line text

else-line:
`# else`

endif-line:
`# endif`

18.7 Templates

1 *template-declaration:*
`template` < *template-parameter-list* > *declaration*

template-parameter-list:
template-parameter
template-parameter-list `,` *template-parameter*

template-parameter:
type-parameter
parameter-declaration

type-parameter:
`class` *identifier*

template-class-id:
template-name < *template-argument-list* >

template-name:
identifier

template-argument-list:
template-argument
template-argument-list `,` *template-argument*

template-argument:
expression
type-id

18.8 Exception handling

1 *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*$_{\mathtt{opt}}$

exception-specification:
  `throw ( ` *type-id-list*$_{opt}$ ` )`

type-id-list:
  type-id
  type-id-list `,` *type-id*

19

Compatibility

1 This chapter summarizes the evolution of C++ since the first edition of *The C++ Programming Language* and explains in detail the differences between C++ and C. Because the C language as described by the ANSI C Standard differs from the dialects of Classic C used up till now, we discuss the differences between C++ and ANSI C as well as the differences between C++ and Classic C.

19 Appendix B: Compatibility

1 This appendix is not part of the C++ reference manual proper and does not define C++ language features.

2 C++ is based on C (K&R78) and adopts most of the changes specified by the ANSI C standard. Converting programs among C++, K&R C, and ANSI C may be subject to vicissitudes of expression evaluation. All differences between C++ and ANSI C can be diagnosed by a compiler. With the following three exceptions, programs that are both C++ and ANSI C have the same meaning in both languages:

3 In C, `sizeof('a')` equals `sizeof(int)`; in C++, it equals `sizeof(char)`.

4 In C, given

```
enum e { A };
```

`sizeof(A)` equals `sizeof(int)`; in C++, it equals `sizeof(e)`, which need not equal `sizeof(int)`.

5 A class name declared in an inner scope can hide the name of an object, function, enumerator, or type in an outer scope. For example,

```
int x[99];
void f()
{
    struct x { int a; };
    sizeof(x);  /* size of the array in C */
                /* size of the struct in C++ */
}
```

19.1 Extensions

1 This section summarizes the major extensions to C provided by C++.

19.1.1 C++ features available in 1985

1 This subsection summarizes the extensions to C provided by C++ in the 1985 version of this manual:

2 The types of function parameters can be specified (8.2.5) and will be checked (5.2.2). Type conversions will be performed (5.2.2). This is also in ANSI C.

3 Single-precision floating point arithmetic may be used for `float` expressions; 3.6.1 and 4.3. This is also in ANSI C.

4 Function names can be overloaded; 13.

5 Operators can be overloaded; 13.4.

6 Functions can be inline substituted; 7.1.2.

7 Data objects can be `const`; 7.1.6. This is also in ANSI C.

8 Objects of reference type can be declared; 8.2.2 and 8.4.3.

9 A free store is provided by the `new` and `delete` operators; 5.3.3, 5.3.4.

10 Classes can provide data hiding (11), guaranteed initialization (12.1), user-defined conversions (12.3), and dynamic typing through use of virtual functions (10.2).

11 The name of a class or enumeration is a type name; 9.

12 A pointer to any non-`const` and non-`volatile` object type can be assigned to a `void*`; 4.6. This is also in ANSI C.

13 A pointer to function can be assigned to a `void*`; 4.6.

14 A declaration within a block is a statement; 6.7.

15 Anonymous unions can be declared; 9.5.

19.1.2 C++ features added since 1985

1 This subsection summarizes the major extensions of C++ since the 1985 version of this manual:

2 A class can have more than one direct base class (multiple inheritance); 10.1.

3 Class members can be `protected`; 11 .

4 Pointers to class members can be declared and used; 8.2.3, 5.5.

5 Operators `new` and `delete` can be overloaded and declared for a class; 5.3.3, 5.3.4, 12.5. This allows the "assignment to `this`" technique for class specific storage management to be removed to the anachronism section; 19.3.3.

6 Objects can be explicitly destroyed; 12.4.

7 Assignment and initialization are defined as memberwise assignment and initialization; 12.8.

8 The `overload` keyword was made redundant and moved to the anachronism section; 19.3.

9 General expressions are allowed as initializers for static objects; 8.4.

10 Data objects can be `volatile`; 7.1.6. Also in ANSI C.

11 Initializers are allowed for `static` class members; 9.4.

12 Member functions can be `static`; 9.4.

13 Member functions can be `const` and `volatile`; 9.3.1.

14 Linkage to non-C++ program fragments can be explicitly declared; 7.4.

15 Operators `->`, `->*`, and `,` can be overloaded; 13.4.

16 Classes can be abstract; 10.3.

17 Prefix and postfix application of `++` and `--` on a user-defined type can be distinguished.

18 Templates; 14.

19 Exception handling; 15.

19.2 C++ and ISO C

1 The subsections of this section list the differences between C++ and ISO C, by the chapters of this document.

19.2.1 Chapter 2: Lexical conventions

Section 2.2

1 *CHANGE*: C++ style comments (`//`) are added
A pair of slashes now introduce a one-line comment.
RATIONALE: This style of comments is a useful addition to the language.
EFFECT ON ORIGINAL FEATURE: Change to semantics of well-defined feature. A valid ISO C expression containing a division operator followed immediately by a C-style comment will now be treated as a

C++ style comment. For example:

```
{
    int a = 4;
    int b = 8 //* divide by a*/ a;
    +a;
}
```

DIFFICULTY OF CONVERTING: Syntactic transformation. Just add white space after the division operator.
HOW WIDELY USED: The token sequence `//*` probably occurs very seldom.

Section 2.4

2 *CHANGE*: New Keywords
The following keywords are added to C++:

```
asm       catch    class    delete  friend
inline    new      operator private protected
public    template try      this    virtual
throw

and       and_eq   bitand   bitor   or
or_eq     xor_eq   xor      not     not_eq
compl
```

RATIONALE: These keywords were added in order to implement the new semantics of C++.
EFFECT ON ORIGINAL FEATURE: Change to semantics of well-defined feature. Any ISO C programs that used any of these keywords as identifiers are not valid C++ programs.
DIFFICULTY OF CONVERTING: Syntactic transformation. Converting one specific program is easy. Converting a large collection of related programs takes more work.
HOW WIDELY USED: Not uncommon.

Section 2.5.2

3 *CHANGE*: Type of character literal is changed from `int` to `char`
RATIONALE: This is needed for improved overloaded function argument type matching. For example:

```
int function( int i );
int function( char c );

function( 'x' );
```

It is preferable that this call match the second version of function rather than the first.
EFFECT ON ORIGINAL FEATURE: Change to semantics of well-defined feature. ISO C programs which depend on

```
sizeof('x') == sizeof(int)
```

will not work the same as C++ programs.
DIFFICULTY OF CONVERTING: Simple.
HOW WIDELY USED: Programs which depend upon sizeof('x') are probably rare.

19.2.2 Chapter 3: Basic concepts

Section 3.1/1

1 *CHANGE*: C++ does not have “tentative definitions” as in C
E.g., at file scope,

```
int i;
int i;
```

is valid in C, invalid in C++. This makes it impossible to define mutually referential file-local static objects, if initializers are restricted to the syntactic forms of C. For example,

```
struct X { int i; struct X *next; };

static struct X a;
static struct X b = { 0, &a };
static struct X a = { 1, &b };
```

RATIONALE: This avoids having different initialization rules for built-in types and user-defined types.
EFFECT ON ORIGINAL FEATURE: Deletion of semantically well-defined feature.
DIFFICULTY OF CONVERTING: Semantic transformation. In C++, the initializer for one of a set of mutually-referential file-local static objects must invoke a function call to achieve the initialization.
HOW WIDELY USED: Seldom.

Section 3.2

2 *CHANGE*: A 5struct *is a scope in C++, not in C*
RATIONALE: Class scope is crucial to C++, and a struct is a class.
EFFECT ON ORIGINAL FEATURE: Change to semantics of well-defined feature.
DIFFICULTY OF CONVERTING: Semantic transformation.
HOW WIDELY USED: C programs use `struct` extremely frequently, but the change is only noticeable when struct , enumeration, or enumerator names are referred to outside the struct .

Section .3.3/2 [also 7.1.6/1]

3 *CHANGE*: A name of file scope that is explicitly declared `const`, and not explicitly declared `extern`, has internal linkage, while in C it would have external linkage
RATIONALE: Because `const` objects can be used as compile-time values in C++, this feature urges programmers to provide explicit initializer values for each `const`. This feature allows the user to put `const` objects in header files that are included in many compilation units.
EFFECT ON ORIGINAL FEATURE: Change to semantics of well-defined feature.
DIFFICULTY OF CONVERTING: Semantic transformation
HOW WIDELY USED: Seldom

Section 3.4/2

4 *CHANGE*: Main cannot be called recursively and cannot have its address taken
RATIONALE: The main function may require special actions.
EFFECT ON ORIGINAL FEATURE: Deletion of semantically well-defined feature
DIFFICULTY OF CONVERTING: Trivial: create an intermediary function such as mymain(argc, argv) .
HOW WIDELY USED: Seldom

Section 3.6

5 *CHANGE*: C allows "compatible types" in several places, C++ does not
[Note: It is hoped that future revisions of the C++ Working Paper will resolve some of the incompatibility.]
For example, C has the "initial member rule," by which the following is valid:

```
struct a { int i, j; }       xa;
int *pi = (int*)&xa;
n = *pi;
```

RATIONALE: Stricter type checking is essential for C++. [Some of the "compatibile type" implications are still being discussed by WG21 and X3J16.]

EFFECT ON ORIGINAL FEATURE: Deletion of semantically well-defined feature
DIFFICULTY OF CONVERTING: Semantic transformation
HOW WIDELY USED: Common

Section 3.6.2/4

6 *CHANGE*: There is no guarantee that `char*` and `void*` have the same representation and alignment requirements
RATIONALE: [It is not yet determined whether this difference is intentional.] [Note: It is hoped that future revisions of the C++ Working Paper will resolve some of the incompatibility.]
EFFECT ON ORIGINAL FEATURE: Deletion of semantically well-defined feature
DIFFICULTY OF CONVERTING: Semantic transformation
HOW WIDELY USED: Seldom

19.2.3 Chapter 4: Standard conversions

Section 4.6

1 *CHANGE*: Conversion rules of C involving "incomplete type" are not guaranteed in C++
[Note: It is hoped that future revisions of the C++ Working Paper will resolve some of the incompatibility.] The current draft [June 92] lacks a definition for the C "incomplete type", and therefore does not precisely define the behavior of examples such as this (file-scope) code:

```
extern char a[];
void *b=a;
char a[10];
```

RATIONALE: [To be determined.]
EFFECT ON ORIGINAL FEATURE: [To be determined.]
DIFFICULTY OF CONVERTING: [To be determined.]
HOW WIDELY USED: Incomplete types are used frequently. Often a header contains the incomplete type and the type is completed in a different header or source file. [This technique is used in some C++ constructs, so a proper treatment of "incomplete type" is expected to be achieved.]

Section 4.6

2 *CHANGE*: Converting `void*` to a pointer-to-object type requires casting

```
char a[10];
void *b=a;
void foo() {
char *c=b;
}
```

ISO C will accept this usage of pointer to void being assigned to a pointer to object type. C++ will not.
RATIONALE: C++ tries harder than C to enforce compile-time type safety.
EFFECT ON ORIGINAL FEATURE: Deletion of semantically well-defined feature.
DIFFICULTY OF CONVERTING: Could be automated. Violations will be diagnosed by the C++ translator. The fix is to add a cast. For example:

```
char *c = (char *) b;
```

HOW WIDELY USED: This is fairly widely used but it is good programming practice to add the cast when assigning pointer-to-void to pointer-to-object. Some ISO C translators will give a warning if the cast is not used.

Section 4.6

3 *CHANGE*: Only pointers to non-const and non-volatile objects may be implicitly converted to `void*`
RATIONALE: This improves type safety.
EFFECT ON ORIGINAL FEATURE: Deletion of semantically well-defined feature.
DIFFICULTY OF CONVERTING: Could be automated. A C program containing such an implicit conversion from (e.g.) pointer-to-const-object to void* will receive a diagnostic message. The correction is to add an explicit cast.
HOW WIDELY USED: Infrequent.

19.2.4 Chapter 5: Expressions

Section 5

1 *CHANGE*: The C language effectively permits read-only type-punning between certain types
That is to say, an object may have its stored value accessed by an lvalue that has a (slightly) differing type. Lvalue types may differ in qualification:

```
char c = 'x';
const char *pcc = (const char *)&c;
f(*pcc);  /* ok to access via lvalue of type ``const char'' */
```

Lvalue types may differ in signedness (the source code for the library function strcmp typically requires this behavior):

```
char c = 0x12;
unsigned char *puc = (unsigned char *)*puc;
f(*puc);  /* ok to access via lvalue of type ``unsigned char'' */
```

One type may be a member of the other (aggregate or union) type:

```
struct x { int i; } xo = [0];
int* i = (int*)&x;
f(*pi);  /* ok to access via lvalue of member type */

union x { int i; short j; } xo = [0];
int* pi = (int *)&x;
f(*pi);  /* ok to access via lvalue of member type */
```

One type may be a character type, where the other is any other type:

```
union arena { align_t a; char buf[N]; } my_arena;
size_t *p = (size_t *)&arena.buf[0];
f(*p);  /* ok to access chars (bytes) via lvalue of other type */
```

Currently, the C++ language permits no such looseness.
2 For example, the memcpy function can be written in portable C, but it is not [yet] certain whether this is true for C++.
3 [Note: It is hoped that future revisions of the C++ Working Paper will resolve some of the incompatibility.]
RATIONALE: The type-safe nature of C++ [or oversight].
EFFECT ON ORIGINAL FEATURE: Deletion of semantically well-defined feature.
DIFFICULTY OF CONVERTING: [Uncertain.]
HOW WIDELY USED:
Not common.

Section 5.2.2

4 *CHANGE*: Implicit declaration of functions is not allowed
RATIONALE: The type-safe nature of C++.
EFFECT ON ORIGINAL FEATURE: Deletion of semantically well-defined feature. Note: the original feature was labeled as “obsolescent” in ISO C.

DIFFICULTY OF CONVERTING: Syntactic transformation. Facilities for producing explicit function declarations are fairly widespread commercially.
HOW WIDELY USED: Very common.

Section 5.2.4

5 *CHANGE*: The "common initial sequence rule" is not guaranteed
The C language effectively permits any degree of type-punning between "common initial sequences" of members of structs that appear in a union. Currently, the C++ working paper is not clear regarding such "layout compatibility" rules. [Note: It is hoped that future revisions of the C++ Working Paper will resolve some of the incompatibility.]

6 For example, the following is valid in C:

```
struct a { int i, j; }        xa;
struct b { int i; char j; }   xb;

union ab { struct a ma; struct b mb; } xab;
xab = xa;
n = xab.mb.i;
```

RATIONALE: The type-safe nature of C++ [or oversight].
EFFECT ON ORIGINAL FEATURE: Deletion of semantically well-defined feature.
DIFFICULTY OF CONVERTING: Semantic transformation.
HOW WIDELY USED: Not uncommon.

Section 5.3.2/4, 5.4/2

7 *CHANGE*: Types must be declared in declarations, not in expressions
In C, a sizeof expression or cast expression may create a new type. For example,

```
p = (void*)(struct x {int i;} *)0;
```

declares a new type, struct x .
RATIONALE: This prohibition helps to clarify the location of declarations in the source code.
EFFECT ON ORIGINAL FEATURE: Deletion of a semantically well-defined feature.
DIFFICULTY OF CONVERTING: Syntactic transformation.
HOW WIDELY USED: Very rare.

Section 5.4/5

8 *CHANGE*: Converting an integral value to an enumeration type, when the value converted does not equal the value of any enumerator of the enumeration type, gives undefined results
[Note: this is still somewhat uncertain.]
RATIONALE: The type-safe nature of C++.
EFFECT ON ORIGINAL FEATURE: Deletion of a semantically well-defined feature.
DIFFICULTY OF CONVERTING: Semantic transformation.
HOW WIDELY USED: Not uncommon.

19.2.5 Chapter 6: Statements

Section 6.4.2/7.3, 6.7/3.1 (`switch` and `goto` statements

1 *CHANGE*: It is now invalid to jump past a declaration with explicit or implicit initializer (except across entire block not entered)
RATIONALE: Constructors used in initializers may allocate resources which need to be de-allocated upon leaving the block. Allowing jump past initializers would require complicated run-time determination of allocation. Furthermore, any use of the uninitialized object could be a disaster. With this simple compile-

time rule, C++ assures that if an initialized variable is in scope, then it has assuredly been initialized.
EFFECT ON ORIGINAL FEATURE: Deletion of semantically well-defined feature.
DIFFICULTY OF CONVERTING: Semantic transformation.
HOW WIDELY USED: Seldom.

Section 6.6.3/2.1,6.6.3/2.5

2 *CHANGE*: It is now invalid to return (explicitly or implicitly) from a function which is declared to return a value without actually returning a value
RATIONALE: The caller and callee may assume fairly elaborate return-value mechanisms for the return of class objects. If some flow paths execute a return without specifying any value, the compiler must embody many more complications. Besides, promising to return a value of a given type, and then not returning such a value, has always been recognized to be a questionable practice, tolerated only because very-old C had no distinction between void functions and int functions.
EFFECT ON ORIGINAL FEATURE: Deletion of semantically well-defined feature.
DIFFICULTY OF CONVERTING: Semantic transformation. Add an appropriate return value to the source code, e.g. zero.
HOW WIDELY USED: Seldom. For several years, many existing C compilers have produced warnings in this case.

19.2.6 Chapter 7: Declarations

Section 7.1.1/6

1 *CHANGE*: In C++, the `static` or `extern` specifiers can only be applied to names of objects or functions Using these specifiers with type declarations is illegal in C++. In C, these specifiers are ignored when used on type declarations. Example:

```
static struct S {       // valid C, invalid in C++
int i;
// ...
};
```

RATIONALE: Storage class specifiers don't have any meaning when associated with a type. In C++, class members can be defined with the `static` storage class specifier. Allowing storage class specifiers on type declarations could render the code confusing for users.
EFFECT ON ORIGINAL FEATURE: Deletion of semantically well-defined feature.
DIFFICULTY OF CONVERTING: Syntactic transformation.
HOW WIDELY USED: Seldom.

Section 7.1.3/3

2 *CHANGE*: A C++ typedef name must be different from any class type name declared in the same scope (except if the typedef is a synonym of the class name with the same name). In C, a typedef name and a struct tag name declared in the same scope can have the same name (because they have different name spaces)
Example:

```
typedef struct name1 { /*...*/ } name1; // valid C and C++
struct name { /*...*/ };
typedef int name;                  // valid C, invalid C++
```

RATIONALE: For ease of use, C++ doesn't require that a type name be prefixed with the keywords `class`, `struct` or `union` when used in object declarations or type casts. Example:

```
class name { /*...*/ };
name i;                 // i has type 'class name'
```

EFFECT ON ORIGINAL FEATURE: Deletion of semantically well-defined feature.
DIFFICULTY OF CONVERTING: Semantic transformation. One of the 2 types has to be renamed.
HOW WIDELY USED: Rare.

Section 7.1.6/1 [See also 3.3/2]

3 *CHANGE*: const objects must be initialized in C++ but can be left uninitialized in C
RATIONALE: A const object cannot be assigned to so it must be initialized to hold a useful value.
EFFECT ON ORIGINAL FEATURE: Deletion of semantically well-defined feature.
DIFFICULTY OF CONVERTING: Semantic transformation.
HOW WIDELY USED: Seldom.

Section 7.2/3

4 *CHANGE*: C++ objects of enumeration type can only be assigned values of the same enumeration type. In C, objects of enumeration type can be assigned values of any integral type
Example:

```
enum color { red, blue, green };
color c = 1;   // valid C, invalid C++
```

RATIONALE: There is no guarantee that the integral value assigned to the object of enumeration type can be represented by one of the enumerators of the enumeration.
EFFECT ON ORIGINAL FEATURE: Deletion of semantically well-defined feature.
DIFFICULTY OF CONVERTING: Syntactic transformation. (The type error produced by the assignment can be automatically corrected by applying an explicit cast. But see the earlier note about 5.4/5 regarding conversions to enumeration type.)
HOW WIDELY USED: Common.

Section 7.2/3

5 *CHANGE*: In C++, the type of an enumerator is its enumeration. In C, the type of an enumerator is int
Example:

```
enum e { A };
sizeof(A) == sizeof(int)  // in C
sizeof(A) == sizeof(e)    // in C++
/* and sizeof(int) is not necessary equal to sizeof(e) */
```

RATIONALE: In C++, an enumeration is a distinct type.
EFFECT ON ORIGINAL FEATURE: Change to semantics of well-defined feature.
DIFFICULTY OF CONVERTING: Semantic transformation.
HOW WIDELY USED: Seldom. The only time this affects existing C code is when the size of an enumerator is taken. Taking the size of an enumerator is not a common C coding practice.

19.2.7 Chapter 8: Declarators

Section 8.2.5/2

1 *CHANGE*: In C++, a function declared with an empty parameter list takes no arguments.
In C, an empty parameter list means that the number and type of the function arguments are unknown"
Example:

```
int f();  // means   int f(void)      in C++
             //         int f(unknown)   in C
```

RATIONALE: This is to avoid erroneous function calls (i.e. function calls with the wrong number or type of arguments).

EFFECT ON ORIGINAL FEATURE: Change to semantics of well-defined feature. This feature was marked as “obsolescent” in C.
DIFFICULTY OF CONVERTING: Syntactic transformation. The function declarations using C incomplete declaration style must be completed to become full prototype declarations. A program may need to be updated further if different calls to the same (non-prototype) function have different numbers of arguments or if the type of corresponding arguments differed.
HOW WIDELY USED: Common.

Section 8.2.5/5 [See 5.3.2/4]

2 *CHANGE*: In C++, types may not be defined in return or parameter types. In C, these type definitions are allowed
Example:

```
void f( struct S { int a; } arg ) {}      // valid C, invalid C++
enum E { A, B, C } f() {}                 // valid C, invalid C++
```

RATIONALE: When comparing types in different compilation units, C++ relies on name equivalence when C relies on structural equivalence. Regarding parameter types: since the type defined in an parameter list would be in the scope of the function, the only legal calls in C++ would be from within the function itself.
EFFECT ON ORIGINAL FEATURE: Deletion of semantically well-defined feature.
DIFFICULTY OF CONVERTING: Semantic transformation. The type definitions must be moved to file scope, or in header files.
HOW WIDELY USED: Seldom. This style of type definitions is seen as poor coding style.

Section 8.3/1

3 *CHANGE*: In C++, the syntax for function definition excludes the “old-style” C function. In C, “old-style” syntax is allowed, but deprecated as “obsolescent.”
RATIONALE: Prototypes are essential to type safety.
EFFECT ON ORIGINAL FEATURE: Deletion of semantically well-defined feature.
DIFFICULTY OF CONVERTING: Syntactic transformation.
HOW WIDELY USED: Frequent in old programs, but already known to be obsolescent.

Section 8.4.2/2

4 *CHANGE*: In C++, when initializing an array of character with a string, the number of characters in the string (including the terminating `'\0'`) must not exceed the number of elements in the array. In C, an array can be initialized with a string even if the array is not large enough to contain the string terminating `'\0'`
Example:

```
char array[4] = "abcd";    // valid C, invalid C++
```

RATIONALE: When these non-terminated arrays are manipulated by standard string routines, there is potential for major catastrophe.
EFFECT ON ORIGINAL FEATURE: Deletion of semantically well-defined feature.
DIFFICULTY OF CONVERTING: Semantic transformation. The arrays must be declared one element bigger to contain the string terminating `'\0'`.
HOW WIDELY USED: Seldom. This style of array initialization is seen as poor coding style.

19.2.8 Chapter 9: Classes

Section 9.1/2 [See also 7.1.3/3]

1 *CHANGE*: In C++, a class declaration introduces the class name into the scope where it is declared and hides any object, function or other declaration of that name in an enclosing scope. In C, an inner scope declaration of a struct tag name never hides the name of an object or function in an outer scope
Example:

```
int x[99];
void f()
{
        struct x { int a; };
        sizeof(x);  /* size of the array in C    */
        /* size of the struct in C++ */
}
```

RATIONALE: This is one of the few incompatibilities between C and C++ that can be attributed to the new C++ name space definition where a name can be declared as a type and as a nontype in a single scope causing the nontype name to hide the type name and requiring that the keywords `class`, `struct`, `union` or `enum` be used to refer to the type name. This new name space definition provides important notational conveniences to C++ programmers and helps making the use of the user-defined types as similar as possible to the use of built-in types. The advantages of the new name space definition were judged to outweigh by far the incompatibility with C described above.
EFFECT ON ORIGINAL FEATURE: Change to semantics of well-defined feature.
DIFFICULTY OF CONVERTING: Semantic transformation. If the hidden name that needs to be accessed is at global scope, the **::** C++ operator can be used. If the hidden name is at block scope, either the type or the struct tag has to be renamed.
HOW WIDELY USED: Seldom.

Section 9.7/1

2 *CHANGE*: In C++, the name of a nested class is local to its enclosing class. In C the name of the nested class belongs to the same scope as the name of the outermost enclosing class
Example:

```
struct X {
        struct Y { /* ... */ } y;
};
struct Y yy;      // valid C, invalid C++
```

RATIONALE: C++ classes have member functions which require that classes establish scopes. The C rule would leave classes as an incomplete scope mechanism which would prevent C++ programmers from maintaining locality within a class. A coherent set of scope rules for C++ based on the C rule would be very complicated and C++ programmers would be unable to predict reliably the meanings of nontrivial examples involving nested or local functions.
EFFECT ON ORIGINAL FEATURE: Change of semantics of well-defined feature.
DIFFICULTY OF CONVERTING: Semantic transformation. To make the struct type name visible in the scope of the enclosing struct, the struct tag could be declared in the scope of the enclosing struct, before the enclosing struct is defined. Example:

```
struct Y;  // struct Y and struct X are at the same scope
struct X {
        struct Y { /* ... */ } y;
};
```

All the definitions of C struct types enclosed in other struct definitions and accessed outside the scope of the enclosing struct could be exported to the scope of the enclosing struct.
HOW WIDELY USED: Not common. NOTE: This is a consequence of the difference in scope rules, which is documented at section 3.2 above.

Section 9.9/2

3 *CHANGE*: In C++, a typedef name may not be redefined in a class declaration after being used in the declaration
Example:

```
typedef int I;
struct S {
        I i;
        int I;       // valid C, invalid C++
};
```

RATIONALE: When classes become complicated, allowing such a redefinition after the type has been used can create confusion for C++ programmers as to what the meaning of 'I' really is.
EFFECT ON ORIGINAL FEATURE: Deletion of semantically well-defined feature.
DIFFICULTY OF CONVERTING: Semantic transformation. Either the type or the struct member has to be renamed.
HOW WIDELY USED: Seldom.

19.2.9 Chapter 16: Preprocessing directives

Section 16.10/5 (Predefined names)

1 *CHANGE*: Whether __STDC__ is defined and if so, what its value is,
are implementation-defined"
RATIONALE: C++ is not identical to ISO C. Mandating that `__STDC__` be defined would require that translators make an incorrect claim. Each implementation must choose the behavior that will be most useful to its marketplace.
EFFECT ON ORIGINAL FEATURE: Change to semantics of well-defined feature.
DIFFICULTY OF CONVERTING: Semantic transformation.
HOW WIDELY USED: Programs and headers that reference `__STDC__` are quite common.

19.3 Anachronisms

1 The extensions presented here may be provided by an implementation to ease the use of C programs as C++ programs or to provide continuity from earlier C++ implementations. Note that each of these features has undesirable aspects. An implementation providing them should also provide a way for the user to ensure that they do not occur in a source file. A C++ implementation is not obliged to provide these features.

2 The word `overload` may be used as a *decl-specifier* (7) in a function declaration or a function definition. When used as a *decl-specifier*, `overload` is a reserved word and cannot also be used as an identifier.

3 The definition of a static data member of a class for which initialization by default to all zeros applies (8.4, 9.4) may be omitted.

4 An old style (that is, pre-ANSI C) C preprocessor may be used.

5 An `int` may be assigned to an object of enumeration type.

6 The number of elements in an array may be specified when deleting an array of a type for which there is no destructor; 5.3.4.

7 A single function `operator++()` may be used to overload both prefix and postfix ++ and a single function `operator--()` may be used to overload both prefix and postfix `--`; 13.4.6.

8

19.3.1 Old style function definitions

1 The C function definition syntax

old-function-definition:
  decl-specifiers$_{opt}$ old-function-declarator declaration-seq$_{opt}$ function-body

old-function-declarator:
  declarator (*parameter-list$_{opt}$*)

parameter-list:
identifier
parameter-list , *identifier*

For example,

```
max(a,b) int b; { return (a<b) ? b : a; }
```

may be used. If a function defined like this has not been previously declared its parameter type will be taken to be `(...)`, that is, unchecked. If it has been declared its type must agree with that of the declaration.

2 Class member functions may not be defined with this syntax.

19.3.2 Old style base class initializer

1 In a *mem-initializer*(12.6.2), the *class-name* naming a base class may be left out provided there is exactly one immediate base class. For example,

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

class D : public B {
    // ...
    D(int i) : (i) { /* ... */ }
};
```

causes the `B` constructor to be called with the argument `i`.

19.3.3 Assignment to `this`

1 Memory management for objects of a specific class can be controlled by the user by suitable assignments to the `this` pointer. By assigning to the `this` pointer before any use of a member, a constructor can implement its own storage allocation. By assigning the null pointer to `this`, a destructor can avoid the standard deallocation operation for objects of its class. Assigning the null pointer to `this` in a destructor also suppressed the implicit calls of destructors for bases and members. For example,

```
class Z {
    int z[10];
    Z()  { this = my_allocator( sizeof(Z) ); }
    ~Z() { my_deallocator( this ); this = 0; }
};
```

2 On entry into a constructor, `this` is nonnull if allocation has already taken place (as it will have for `auto`, `static`, and member objects) and null otherwise.

3 Calls to constructors for a base class and for member objects will take place (only) after an assignment to `this`. If a base class's constructor assigns to `this`, the new value will also be used by the derived class's constructor (if any).

4 Note that if this anachronism exists either the type of the `this` pointer cannot be a `*const` or the enforcement of the rules for assignment to a constant pointer must be subverted for the `this` pointer.

19.3.4 Cast of bound pointer

1 A pointer to member function for a particular object may be cast into a pointer to function, for example, `(int(*)())p->f`. The result is a pointer to the function that would have been called using that member function for that particular object. Any use of the resulting pointer is − as ever − undefined.

19.3.5 Nonnested classes

1 Where a class is declared within another class and no other class of that name is declared in the program that class can be used as if it was declared outside its enclosing class (exactly as a C `struct`). For example,

```
struct S {
    struct T {
        int a;
    };
    int b;
};

struct T x;      // meaning 'S::T x;'
```