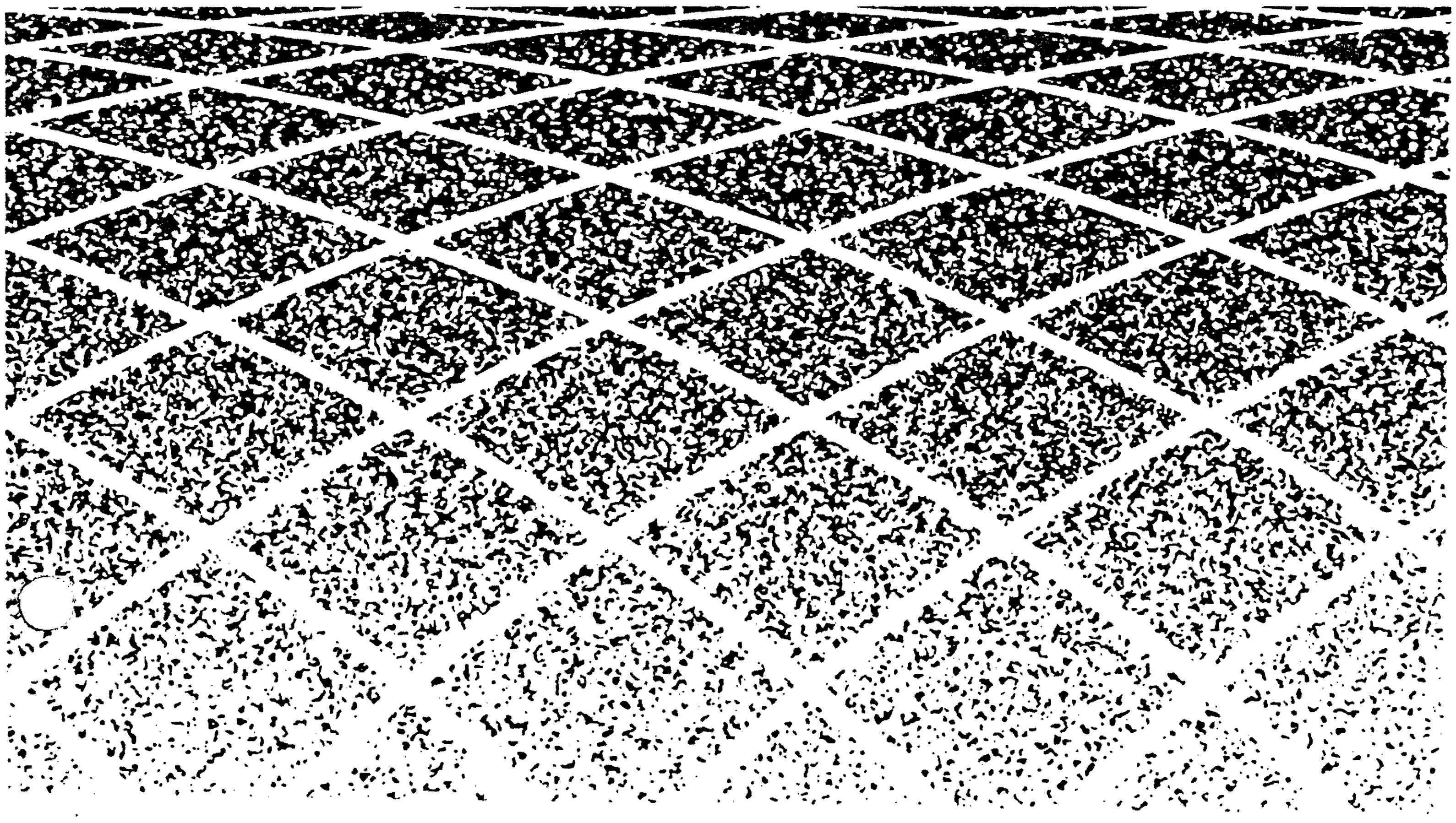
AT&T

UNIX® System V
AT&T C++ Language System
Release 2.0

Product Reference Manual
Select Code 307-146



Preface

Preface

The *AT&T C++ Language System Product Reference Manual* provides a complete definition of the C++ language supported by Release 2.0 of the C++ Language System. The manual is part of a set of four documents that are supplied with your C++ Language System. The other documents are:

- the *Release Notes*, which describe the contents of this release, how to install it, and changes to the language
- the *Selected Readings*, which contains papers describing aspects of the C++ language
- the *Library Manual*, which describes the three C++ class libraries and tells you how to use them

This manual contains 16 sections covering the various aspects of the C++ language:

1. Introduction
2. Lexical Conventions
3. Basic Concepts
4. Standard Conversions
5. Expressions
6. Statements
7. Declarations
8. Declarators
9. Classes
10. Derived Classes
11. Member Access Control
12. Special Member Functions
13. Overloading
14. Templates (experimental)
15. Exception Handing (experimental)
16. Compiler Control Lines

NOTE Sections 14 and 15 are place markers for experimental features that are not implemented in Release 2.0.

The *Reference Manual* proper is followed by appendices that describe grammar, compatibility, and product-specific behavior in Release 2.0:

- Appendix A: Grammar Summary
- Appendix B: Compatibility

- Appendix C: Implementation-Specific Behavior — describes behavior of the Release 2.0 Language System that is implementation specific
- Appendix D: Not Implemented Messages — lists error messages that result when you attempt to use a feature not implemented in Release 2.0, and describes the unimplemented feature

To make the best use of the *Product Reference Manual,* you should be familiar with the C programming language and the C programming environment under the UNIX® operating system. Refer to Appendix B of the *Release Notes* for further sources of information about these topics.

C++ Reference Manual

C++ Reference Manual

NOTE This reference manual is by Bjarne Stroustrup.

NOTE This is the May 1989 draft of the C++ *Reference Manual*

1. Introduction

This manual describes the C++ programming language as of May 1989. C++ is a general purpose programming language based on the C programming language*. In addition to the facilities provided by C, C++ provides classes, templates, exception handling, 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 §B.1. The differences between C++ and ANSI C† are summarized in §B.2. The extensions to C++ since the 1985 edition of this manual are summarized in §B.1.2. The sections related to templates (§14) and exception handling (§15) are placeholders for planned language extensions.

1.1 Overview

This manual is organized like this:

* "The C Programming Language" by Brian W. Kernighan and Dennis M. Ritchie, Prentice Hall, 1978 and 1988.

† Draft Proposed American National Standard X3J11/88-090 dated Dec 7, 1988.

1.2 Syntax Notation

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 non-terminal symbol is indicated by the subscript "*opt*," so that

{ $\textit{expression}_{opt}$ }

indicates an optional expression enclosed in braces.

2. Lexical Conventions

A C++ program consists of one or more *translation units* §3.3. A translation unit is conceptually translated in several phases. The first of these is preprocessing (§16), which performs file inclusion and macro substitution. Preprocessing is controlled by directives introduced by lines where the first non-whitespace character (§2.1) is #. When preprocessing is complete, a translation unit consists of a sequence of tokens. The word *file* is used to refer to a translation unit after preprocessing.

2.1 Tokens

There are six kinds of tokens: identifiers, keywords, constants, strings, operators, and other separators. Blanks, horizontal and vertical tabs, new-lines, 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 constants.

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.

2.2 Comments

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.

2.3 Identifiers

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.

2.4 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

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

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

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 a single token.

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

```
#       ##
```

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 standard header files (§16.2):

```
<stddef.h>  <limits.h>  <stdlib.h>
```

These headers are part of the ANSI C standard.

2.5 Literals

There are several kinds of literal constants:

literal:
> *integer-constant*
> *character-constant*
> *floating-constant*
> *string*

2.5.1 Integer Constants

An integer constant 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 10 through 15. For example, the number twelve can be written `12`, `014`, or `0XC`.

The type of an integer constant 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`.

For example, `100000` is of type `int` on a machine with 32 bit `int`s, but of type `long int` on a machine with 16 bit `int`s and 32 bit `long`s. Similarly, `0XA000` is of type `int` on a machine with 32 bit `int`s, but of type `unsigned int` on a machine with 16 bit `int`s. These implementation dependencies can in many cases be avoided by using suffixes: `100000L` is `long int` on all machines and `0XA000U` is of type `unsigned int` on all machines with at least 16 bits used to represent an `unsigned int`.

2.5.2 Character Constants

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

Certain non-graphic 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.

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 constant is implementation dependent if it exceeds that of the largest `char`.

A character constant immediately preceded by the letter `L`, for example `L'ab'`, is a wide-character constant. A wide-character constant is of type `wchar_t`, an integral type (§3.6.1) defined in the standard header `<stddef.h>`. Wide characters are intended for character sets where a character does not fit into a single byte.

2.5.3 Floating Constants

A floating constant 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 constant is `double` unless explicitly specified by a suffix. The suffixes `f` and `F` specify `float`, the suffixes `l` and `L` specify `long double`.

2.5.4 String Literals

A string literal is a sequence of characters (as defined in §2.5.2) 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 non-overlapping objects) is implementation dependent. The effect of attempting to modify a string literal is undefined.

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'`).

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 \.

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`," where `wchar_t` is an integral type defined in the standard header `<stddef.h>`. Concatenation of ordinary and wide-character string literals is undefined.

3. Basic Concepts

A name denotes an object, a function, a set of functions, 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.

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

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), or it is a class name declaration (§9.1). For example, these 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;
```

3.2 Scopes

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 formal arguments for a function are treated as if they were declared in the outermost block of that function.

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 file 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), after the `->` operator applied to a pointer to an object of its class (§5.2.4), or after the `::` scope resolution operator (§5.1) applied to the name of its class or a class derived from (§10) its class. A class, enumeration (§7.2), or a *typedef-name* declared within a class (§9.7) is not considered a member and its name belongs to the enclosing scope; the same is true for a name declared by a `friend` declaration (§11.4).

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, 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 can still be used when appropriately (§7.1.6) prefixed with `class`, `struct`, or `union`. The scope rules are summarized in §10.4.

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

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

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

3.3 Program and Linkage

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

A name of file scope that is explicitly declared `static` is local to its file and can be used as a name for other objects, functions, etc., in other files. Such names are said to have internal linkage. A name of file scope that is explicitly declared `const` or `inline` and not explicitly declared `extern` is local to its file. So is the name of a class that has not been used in the declaration of an object or function that is not local to its file and has no static members (§9.4) and no non-inline 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 multi-file program refers to the same object (§3.7), function (§8.2.5), or class (§9). Such names are said to be external or 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 in a program that has been used in the declaration of an object or function with external linkage or has a static member or a non-inline member function refers to the same class.

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

Static class members (§9.4) have external linkage.

Local names (§3.2) explicitly declared `extern` have external linkage.

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). There must be exactly one definition for each function, object, and class in a program.

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

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

3.4 Start and Termination

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. It is recommended that the two examples below be allowed on any implementation and that any further arguments required be added after `argv`. The function `main()` may be defined as:

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

or

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

In the latter case `argc` shall be the number of parameters passed to the program from an environment in which the program is run. If `argc` is non-zero these parameters 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`.

`main()` may not be called from within a program.

The value returned by `main()` is returned to the program's environment as the value of the program.

Calling the function

```
void exit(int)
```

declared in `<stdlib.h>` terminates the program. The argument value is returned to the program's environment as the value of the program.

The initialization of static objects (§3.5) from a file is done before the first use of any function or object defined in that file. 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 file. The default initialization of all static objects to zero (§8.4) is performed before any dynamic (i.e., run-time) initialization. No further order is imposed on the initialization of objects from different files.

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. In this case, objects initialized before an `atexit()` call may not be destroyed until after the function specified in the `atexit()` call have 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.

Calling the function

```
void abort()
```

declared in `<stdlib.h>` terminates the program without executing destructors for static objects.

3.5 Storage Classes

There are two declarable storage classes: automatic and static.

Automatic objects are local to each invocation of a block.

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

Local objects are initialized (§12.1) each time their block is entered and destroyed (§12.4) upon exit from it. 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

There are two kinds of types: fundamental types and derived types.

3.6.1 Fundamental Types

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

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`. A plain `char` is either a `signed char`, or an `unsigned char`, but it is implementation dependent which. A `signed char` and an `unsigned char` consume the same amount of space.

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.

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.

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`.

Types `char`, `int` of all sizes, and enumerations (§7.2) are collectively called *integral* types. *Integral* and *floating* types are collectively called *arithmetic* types.

The `void` type specifies an empty set of values. The (nonexistent) value of a `void` object may not be used in any way, and neither explicit nor implicit conversions to a non-void type may be applied. Because a void expression denotes a nonexistent value, such an 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). No object of type `void` may be declared. Any expression may be explicitly converted to type `void` (§5.4).

3.6.2 Derived Types

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 take arguments of given types and return objects of a given type, §8.2.5;

pointers to objects or functions 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 structures capable of containing objects of different types at different times, §9.5;

pointers to class members, which identify members of a given type within objects of a given class, §8.2.3.

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

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`."

Objects of type `void*` (pointer to void), `const void*`, and `volatile void*` can be used to point to objects of unknown type. A `void*` must have sufficient bits to hold any object pointer.

3.6.3 Type Names

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

An *object* is a region of storage; an *lvalue* is an expression referring to an object. 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 below indicates whether it expects lvalue operands and whether it yields an lvalue. An lvalue is *modifiable* if it is not a function name, an array name, or `const`.

4. Standard Conversions

A number of 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. §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

A `char`, 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 may be used. 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`. This process is called *integral promotion*.

4.2 Integral Conversions

When an integer is converted to an *unsigned* type, the value is the least unsigned integer congruent to the signed integer (modulo 2^n where n is the number of bits used to represent the unsigned type). In a 2's complement representation, this conversion is conceptual and there is no actual change in the bit pattern.

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

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

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 will not fit in the space provided.

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

A great many operators cause conversions and yield result types in a similar way. This pattern will be called the "usual arithmetic conversions."

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

The following conversions may be performed wherever pointers (§8.2.1) are assigned, initialized, compared, etc.:

A constant expression (§5.19) that evaluates to 0 is converted to a pointer, commonly called the null pointer. It is guaranteed that this value will produce a pointer distinguishable from a pointer to any object.

A pointer to any object type may be converted to a `void*`.

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 a public base class of that class (§10) provided this conversion can be done unambiguously (§10.1.1). 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.

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 `()`.

4.7 Reference Conversions

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 class may be converted to a reference to a public base class 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

The following conversion may be performed wherever pointers to members (§8.2.3) are initialized, assigned, compared, etc.:

A constant expression (§5.19) that evaluates to 0 is converted to a pointer to member. It is guaranteed that this value will produce a pointer to member distinguishable from any 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 derivation was public and provided this conversion can be done unambiguously (§10.1.1).

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.

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

An expression is a sequence of operators and operands that specifies a computation. An expression may result in a value and it may cause side effects.

This section defines the syntax, order of evaluation, and meaning of expressions.

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). Operator overloading cannot modify the rules for operators applied to types for which they are defined in this section.

The order of evaluation of subexpressions is determined by the precedence and grouping of the operators. The usual mathematical rules for associativity and commutativity of operators may be applied only where the operators really are associative and commutative. Except where noted, the order of evaluation of operands of individual operators is undefined. In particular, if a value is modified twice in an expression, the result of the expression is undefined 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
```

The handling of overflow and divide check in expression evaluation is implementation dependent. Most existing implementations of C++ ignore integer overflows. Treatment of division by 0 and all floating-point exceptions varies among machines, and is usually adjustable by a library function.

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*`. Such usages do not count as standard conversions when considering overloading resolution (§13.2).

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.

User-defined conversions of class objects to and from fundamental types, pointers, etc., can be defined (§12.3). If unambiguous (§13.2), such conversions may be applied 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

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

primary-expression:
- *literal*
- `this`
- `::` *identifier*
- `::` *operator-function-name*
- `(` *expression* `)`
- *name*

A *literal* is a primary expression. Its type depends on its form (§2.5).

The keyword `this` is a local variable in the body of a non-static member function (§9.3); it is a pointer to the object for which the function was invoked. The keyword `this` cannot be used outside of a class member function body.

The operator `::` followed by an *identifier* or an *operator-function-name* is a primary expression. Its type is specified by the declaration of the identifier or *operator-function-name*. The result is the identifier or *operator-function-name*. The result is an lvalue if the identifier is. The identifier or *operator-function-name* must be of file scope. The identifier may not be a class or type name. The identifier may be hidden by a type name. Use of `::` allows an object or a function to be referred to even if its identifier has been hidden (§3.2).

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.

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

```
name:
        identifier
        operator-function-name
        conversion-function-name
        qualified-name
```

An *identifier* is a *name* provided it has been suitably declared (§7). For *operator-function-names* see §13.4. For *conversion-function-names* see §12.3.2.

```
qualified-name:
        class-name :: identifier
        class-name :: operator-function-name
        class-name :: conversion-function-name
        class-name :: class-name
        class-name :: ~ class-name
```

A *class-name* (§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-name*; 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 non-type name; in this case the *class-name* is still found and used. Where *class-name* `::` *class-name* or *class-name* `::` `~` *class-name* is used the two *class-names* must refer to the same class; this notation names constructors (§12.1) and destructors (§12.4), respectively.

5.2 Postfix Expressions

Postfix expressions group left-to-right.

```
postfix-expression:
        primary-expression
        postfix-expression [ expression ]
        postfix-expression ( expression-list_opt )
        simple-type-name ( expression-list_opt )
        postfix-expression . name
        postfix-expression -> name
        postfix-expression ++
        postfix-expression --

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

5.2.1 Subscripting

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 integral type. The type of the result is "`T`." 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

A function call is a postfix expression followed by parentheses containing a possibly empty, comma-separated list of expressions which constitute the actual arguments to the function. The postfix expression must be of type "function returning `T`," "pointer to function returning `T`," or "reference to function returning `T`," and the result of the function call is of type "`T`."

When a function is evaluated, each formal argument is initialized (§8.4.3, §12.8, §12.1) with its actual argument. Standard (§4) and user-defined (§12.3) conversions are performed. A function may change the values of its non-constant formal arguments, but these changes cannot affect the values of the actual arguments except where a formal argument is of a non-`const` reference type (§8.2.2). Where a formal argument is of reference type a temporary variable is introduced if the actual argument is a constant (§7.1.6, §2.5, §2.5.4, §8.2.4) or of a type that requires type conversion (§12.2). In addition, it is possible to modify the values of non-constant objects through pointer arguments.

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

A function can only be called if a declaration of it is accessible from the scope of the call. This implies that, except where the ellipsis (`...`) is used, a formal argument is available for each actual argument.

Any actual argument of type `float` for which there is no formal argument is converted to `double` before the call; any of type `char`, `short`, etc., for which there is no formal argument is converted to `int` or `unsigned` by integral promotion (§4.1). An object of a class for which a constructor is declared cannot be passed as an argument except where a formal argument of an appropriate type is declared. In that case the formal argument is initialized with the actual argument by a constructor call before the function is entered (§12.2, §12.8).

The order of evaluation of arguments is implementation dependent; 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 implementation dependent.

Recursive calls are permitted.

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

5.2.3 Explicit Type Conversion

A *simple-type-name* (§7.1.6) followed by a possibly empty parenthesized expression (or an *expression-list* if the type is a class with a suitably declared constructor (§8.4)) causes the value of the expression to be converted to the named type; see also (§5.4). The parentheses may only be empty if the type is a class with a suitably declared constructor.

5.2.4 Class Member Access

A postfix expression followed by a dot (`.`) followed by a *name* is a postfix expression. The first expression must be a class object, and the *name* must name a member of that class. The result is the named member of the object, and it is an lvalue if the member is an lvalue.

A postfix expression followed by an arrow (`->`) followed by a *name* is a postfix expression. The first expression must be a pointer to a class object and the *name* must name a member of that class. The result is the named member of the object to which the pointer points, and it is an lvalue if the member is an lvalue. Thus the expression `E1->MOS` is the same as `(*E1).MOS`.

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

5.2.5 Increment and Decrement

The value obtained by applying a postfix `++` is the value of the operand. The operand must be a modifiable lvalue. 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 the discussions of addition (§5.7) and assignment operators (§5.17) for information on conversions.

The operand of postfix `--` is decremented analogously to the postfix `++` operator.

5.3 Unary Operators

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-name* `)`
  allocation-expression
  deallocation-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`."

The result of the unary `&` operator is a pointer to its operand. The operand must be a function, an lvalue, or a *qualified-name*. 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 `const T` has type `const T*`; `volatile` is handled similarly. For a *qualified-name*, 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`." The address of an overloaded function (§13) can only be taken in an initialization or assignment where the left hand side uniquely determines which version of the overloaded function is referred to (§13.3).

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.

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.

The operand of the logical negation operator `!` must have arithmetic type or be a pointer; its value is 1 if the value of its operand is 0 and 0 if the value of its operand is non-zero. The type of the result is `int`.

The operand of `~` must have integral type; the result is the 1'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

The operand of prefix `++` is incremented by `1`. The operand must be a modifiable lvalue. 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.

The operand of prefix `--` is decremented analogously to the prefix `++` operator.

5.3.2 Sizeof

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, or an undefined class. A *byte* is undefined by the language except in terms of the value of `sizeof`; `sizeof(char)` is `1`.

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 larger 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. It is illegal to take the size of an array with an unspecified dimension.

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

The `new` operator attempts to create an object of the *type-name* (§8.1) to which it is applied. This type must be an object type; functions cannot be allocated this way, though pointers to functions can.

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

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

restricted-type-name:
  type-specifiers *restricted-declarator*$_{opt}$

restricted-declarator:
  ptr-operator *restricted-declarator*$_{opt}$
  restricted-declarator `[` *expression*$_{opt}$ `]`

The lifetime of an object created by `new` is not restricted to the scope in which it is created. The `new` operator returns a pointer to the object it created. When that object is an array, a pointer to its initial element 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 (§10) is specified all array dimensions but the first must be constant expressions (§5.19) with positive values. The first array dimension can be a general *expression* even when the *type-name* is used (despite the fact that array dimensions in *type-names* are generally restricted to *constant-expressions* (§5.19)).

The `new` operator will call the function `operator new()` to obtain storage. A first argument of `sizeof(T)` is supplied when allocating an object of type `T`. The *placement* syntax can be used to supply additional arguments. For example, `new T` results in a call of `operator new(sizeof(T))` and `new(2,f) T` results in a call `operator new(sizeof(T),2,f)`.

The *placement* syntax can only be used provided an `operatornew()` with suitable argument types (§13.2) has been declared.

When an object of a non-class type (including arrays of class objects) is created with operator `new`, the global `::operator new()` is used. When an object of a class `T` is created with operator `new`, `T::operator new()` is used if it exists (using the usual lookup rules for finding members of a class and its base classes; §10.1.1); otherwise the global `::operator new()` is used. Using `::new` ensures that the global `::operator new()` is used even if `T::operator new()` exists.

An initializer (§8.4) of the form `(` *expression-list*$_{opt}$ `)` may be supplied in an *allocation-expression*. For objects of classes with a constructor (§12.1) this argument list will be used in a constructor call; otherwise the initializer must be of the form `(` *expression* `)` and will be used to initialize the object.

If a class has a constructor an object of it can only be created by `new` if suitable arguments are provided or if the class has a constructor that can be called without arguments.

No initializers can be specified for arrays. Arrays of objects of a class with constructors can only be created by operator `new` if the class has a default constructor (§12.1). In that case, the default constructor will be called for each element of the array.

Initialization is done only if the value returned by `operator new()` is non-zero. If the value returned by the `operator new()` is `0` (the null pointer) the value of the expression is `0`.

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

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

is an error because the binding is

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

Objects of general type can be expressed using the explicitly parenthesized version of the `new` operator. For example:

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

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

When parsing the *restricted-type* in an *allocation-expression* expressions the longest sequence of *restricted-declarators* is used. This prevents ambiguities between declarator operators `*`, and `[]` and their expression counterparts. For example,

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

5.3.4 Delete

The `delete` operator destroys an object created by the `new` operator.

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

The result has type `void`. The operand of `delete` must be a pointer returned by `new`. The effect of applying `delete` to a pointer not obtained from the `new` operator without a placement specification is undefined and usually harmful. However, deleting a pointer with the value zero is guaranteed to be harmless.

The effect of accessing 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 its value is undefined after the deletion.

The `delete` operator will invoke the destructor (if any §12.4) for the object pointed to.

To free the store pointed to, the `delete` operator will call the function `operator delete()`. For objects of a non-class type (including arrays of class objects), the global `::operator delete()` is used. For an object of a class `T`, `T::operator delete()` is used if it exists (using the usual lookup rules for finding members of a class and its base classes; §10.1.1); otherwise the global `::operator delete()` is used. Using `::delete` ensures that the global `::operator delete()` is used even if `T::operator delete()` exists.

The form

  `delete [` *expression* `]` *cast-expression*

is used to delete arrays. The second expression points to an array and the first expression gives the number of elements of that array (see §12.4). The destructors (if any) for the objects pointed to will be invoked.

5.4 Explicit Type Conversion

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

An explicit type conversion can be expressed either using functional notation (§5.2.3) or the *cast* notation. The *cast* notation is needed to express conversion to a type that does not have a *simple-type-name*.

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.

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.

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.

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 cause addressing exceptions upon use 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 size may be converted to a pointer to an object of the same or smaller size and back again without change. 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 pointer to a class B may be explicitly converted to a pointer to a class D that has B as a direct or indirect base class provided an unambiguous conversion from D to B exists (§4.6, §10.1.1) and provided that B is not a virtual base class (§10.1). Such a cast from a base to a derived class assumes that the object of the base class is a sub-object of an object of the derived class; the resulting pointer points to the enclosing object of the derived class. If the object of the base class is not a sub-object of an object of the derived class the cast may cause an exception. The null pointer (0) is converted into itself.

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, etc.

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 refer to the same object as the original (uncast) expression.

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

A pointer to 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.

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

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 functions of classes derived from each other.

A pointer to an object of a const type can be cast into a pointer to a non-const. The resulting pointer will refer to the original object. An object of a const type or a reference to an object of a const type can be cast into a reference to a non-const. The resulting reference will refer to the original object. The result of attempting to modify that object through such a pointer or reference is undefined.

5.5 Pointer-to-Member Operators

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

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

The binary operator .* binds its second operand, which must be of type "pointer to member of class T" to its first operand, which must be of class T or of a class publicly derived from class T. The result is an object or a function of the type specified by the second operand.

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 class publicly derived from T." The result is an object or a function of the type specified by the second operand.

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`.

5.6 Multiplicative Operators

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

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

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.

The binary `*` operator indicates multiplication.

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 is `0` the result is undefined; otherwise `(a/b)*b + a%b` is equal to `a`. If both operands are non-negative then the remainder is non-negative; if not, the sign of the remainder is implementation dependent.

5.7 Additive Operators

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*

The result of the + operator is the sum of the operands. A pointer to an object in an array and a value of any integral type may be added. The result is a pointer of the same type as the original pointer, which points to another object in the same array, appropriately offset from the original object. Thus if `P` is a pointer to an object in an array, the expression `P+1` is a pointer to the next object in the array. If the resulting pointer points outside the bounds of the array, except at the first location beyond the high end of the array, the result is undefined.

The result of the - operator is the difference of the operands. A value of any integral type may be subtracted from a pointer, and then the same conversions apply as for addition.

No further type combinations are allowed for pointers.

If two pointers to objects of the same type are subtracted, the result is a signed integral value representing the number of objects separating the pointed-to objects. Pointers to successive elements of an array differ by `1`. The type of the result is implementation dependent, but is defined as `ptrdiff_t` in the standard header `<stddef.h>`. The value is undefined unless the pointers point to elements of the same array or to the first location beyond the high end of the array.

5.8 Shift Operators

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 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 0-filled. The value of `E1 >> E2` is `E1` right-shifted `E2` bit positions. The right shift is guaranteed to be logical (0-fill) if `E1` has an unsigned type or if it has a non-negative value; otherwise the result is implementation dependent.

5.9 Relational Operators

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 operators < (less than), > (greater than), <= (less than or equal to), and >= (greater than or equal to) all yield `0` if the specified relation is false and `1` if it is true. The type of the result is `int`.

The usual arithmetic conversions are performed on arithmetic operands. Pointer conversions are performed on pointer operands. This implies that any pointer may be compared to a constant expression evaluating to `0` and any pointer can be compared to a pointer of 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.

Two pointers to the same object compare equal. If two pointers point to non-static members of the same object, the pointer to the member defined last compares higher provided the two members are defined using the same *access-specifier* (§11.1) and provided their class is not a union. If two pointers point to non-static members of the same union, they compare equal. 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 highest subscript compares higher. Other pointer comparisons are implementation dependent.

5.10 Equality Operators

> *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.)

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 `0`.

5.11 Bitwise AND Operator

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

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

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

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

The `&&` operator groups left-to-right. It returns 1 if both its operands are non-zero, 0 otherwise. Unlike `&`, `&&` guarantees left-to-right evaluation; moreover the second operand is not evaluated if the first operand evaluates to `0`.

The operands need not have the same type, but each must have arithmetic type or be a pointer. The result is an `int`. All side effects of the first expression happen before the second expression is evaluated.

5.15 Logical OR Operator

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

The `||` operator groups left-to-right. It returns 1 if either of its operands is non-zero, and 0 otherwise. Unlike `|`, `||` guarantees left-to-right evaluation; moreover, the second operand is not evaluated if the first operand evaluates to non-zero.

The operands need not have the same type, but each must have arithmetic type or be a pointer. The result is an `int`. All side effects of the first expression happen before the second expression is evaluated.

5.16 Conditional Operator

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

Conditional expressions group right-to-left. The first expression must have arithmetic type or be a pointer type. It is evaluated and if it is non-zero, 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.

If both the second and the third expressions are of arithmetic type, 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 `0`, pointer conversions are performed to bring them to a common type. Otherwise, if both the second and the third expressions are references, reference conversions are performed to bring them to a common type. Otherwise, if both the second and the third expressions are `void`, the common type is `void`. Otherwise the expression is illegal. 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 operand are of the same type and both are lvalues.

5.17 Assignment Operators

There are a number of 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

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

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 right operand is of an enumeration type the left operand must be of the same type. If the left operand has pointer type, the right operand must be of pointer type or a constant expression that evaluates to `0`; the right operand is converted to the type of the left before the assignment.

A pointer of type `T*const` can be assigned to a pointer of type `T*`, but the reverse assignment is illegal (§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).

If the left operand has pointer to member type, the right operand must be of pointer to member type or a constant expression that evaluates to `0`; the right operand is converted to the type of the left before the assignment.

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 public derivation can be assigned to an `X`.

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 public derivation.

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

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, in which case the (integral) right operand is converted as explained in §5.7; all right operands and all non-pointer left operands must have arithmetic type.

Note that for class objects assignment is not in general the same as initialization (§8.4, §12.1, §12.6, §12.8).

5.18 Comma Operator

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.

In contexts where comma is given a special meaning, for example, in lists of actual arguments to functions (§5.2.2) and lists of initializers (§8.4), the comma operator as described in this section can only appear 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

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 (§2.5), enumerators, `const` values of integral types initialized with constant expressions (§8.4), and `sizeof` expressions. Floating constants (§2.5.3) 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.

6. Statements

Except as indicated, statements are executed in sequence.

statement:
labeled-statement
expression-statement
compound-statement
selection-statement
iteration-statement
jump-statement

dstatement:
statement
declaration-statement

6.1 Labeled Statement

A statement may be labeled.

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

An identifier label declares the identifier. The only use of an identifier label is as a target of a `goto`. The scope of a label is the current function. 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.

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

6.2 Expression Statement

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`.

6.3 Compound Statement, or Block

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

compound-statement:
{ *statement-list*$_{opt}$ }

statement-list:
dstatement
statement-list dstatement

Note that a declaration is a *dstatement* (§6.7).

6.4 Selection Statements

Selection statements choose one of several flows of control.

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

6.4.1 If Statements

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

The expression is evaluated and if it is non-zero, the first substatement is executed. If `else` is used, the second substatement is executed if the expression is zero. The "else" ambiguity is resolved by connecting an `else` with the last encountered `else`-less `if`.

6.4.2 Switch Statement

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

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.

There may also be at most one label of the form

```
default :
```

within a `switch` statement.

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

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 labeled statement. If no case matches and if there is no `default` then none of the statements in the switch is executed.

`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.

Usually the statement that is the subject of a switch is compound. Declarations may appear in this statement. However, it is illegal to jump past a declaration with an explicit or implicit initializer unless the declaration is in an inner block that is not entered (that is, completely bypassed by the transfer of control). This implies that declarations that contain explicit or implicit initializers must be contained in an inner block.

6.5 Iteration Statements

Iteration statements specify looping.

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

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

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

6.5.1 While Statement

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

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

6.5.2 Do statement

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

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

6.5.3 For Statement

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* 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; the second expression often specifies incrementing that is done after each iteration. The first expression must have arithmetic or pointer type or a class type for which an unambiguous conversion to arithmetic or pointer type exists (§12.3).

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

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

A *for-statement* containing a declaration in its *for-init-statement* may not be the *statement* after an `if`, `else`, `switch`, `while`, `do`, or `for`. This restriction follows from the rule against jumping past initialized declarations (§6.6).

6.6 Jump Statements

Jump statements unconditionally transfer control.

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

On exit from a scope (however accomplished), destructors are called for all constructed class objects in that scope that have not yet been destroyed. This applies to both explicitly declared objects and temporaries (§12.2).

6.6.1 Break Statement

The `break` statement must occur 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 Continue Statement

The `continue` statement must occur 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 (...) {      do {              for (...) {
  ...                ...               ...
contin: ;          contin: ;         contin: ;
}                  } while (...);    }
```

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

6.6.3 Return Statement

A function returns to its caller by means of the `return` statement. 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` or 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 is illegal in a value-returning function.

6.6.4 Goto Statement

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. It is illegal to jump past a declaration with an explicit or implicit initializer unless the declaration is in an inner block that is not entered (that is, completely bypassed by the transfer of control).

6.7 Declaration Statement

A declaration statement is used to introduce a new identifier 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.

Any initializations of `auto` or `register` variables are done each time their *declaration-statement* is executed. It is possible to transfer into a block, but not in a way that causes initializations not to be done (§6.6). Destruction of local variables declared in the block is done upon exit from the block (§6.6).

Initialization of an object with storage class `static` (§7.1.1) is done the first time control passes through its declaration (only). The destructor for a local `static` object will be executed if and only if the variable was constructed.

6.8 Ambiguity Resolution

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

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, assume `T` is a *simple-type-name* (§7.1.6):

```
T(a)->m = 7;          // expression-statement
T(a)++;               // expression-statement
T(a,5)<<c;            // expression-statement
T(*d)(double(3));     // expression-statement

T(*e)(int);           // declaration
T(f)[];               // declaration
T(g) = { 1, 2 };      // declaration
```

The remaining cases are *declarations*. For example:

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

The disambiguation is purely syntactic; that is, the meaning of the names, beyond whether they are *type-names* or not, is not used in the disambiguation.

Note that a simple lexical lookahead can help a parser disambiguate most cases. Consider analyzing a *statement* consisting of a sequence of tokens:

type-name `(` *d-or-e* `)` *tail*

Here, *d-or-e* must be a *declarator*, an *expression*, or both for the statement to be legal. This implies that *tail* must be a semicolon, something that can follow a parenthesized *declarator* or something that can follow a parenthesized *expression*, that is, an *initializer*, `const`, `volatile`, `(` or `[` or a postfix or infix operator.

A user can explicitly disambiguate cases that appear obscure. For example:

```
void f()
{
      auto int(*p)();    // explicitly declaration
      (void) int(*p)(); // explicitly expression-statement
      0,int(*p)();       // explicitly expression-statement
      (int(*p)());       // explicitly expression-statement
      int(*p)();         // resolved to declaration
}
```

A slightly different ambiguity between *expression-statement*s and *declaration*s is resolved by requiring a *type-name* 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

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-specifiers$_{opt}$ *declarator-list*$_{opt}$;
  asm-declaration
  function-definition
  linkage-specification

The declarators in the declarator-list (§8) contain the identifiers being declared. Only in function definitions (§8.3) and function declarations may the *decl-specifiers* 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 *declarator-list* be empty. *asm-declaration*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

The specifiers that can be used in a declaration are:

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

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

The longest sequence of *decl-specifier*s that could possibly be a type name is taken as the *decl-specifiers* 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 illegal 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)
```

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

The "storage class" specifiers are:

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

The `auto` or `register` specifiers can be applied only to names of objects declared in a block (§6.3) and for formal arguments (§8.3). The `auto` declarator is always redundant and not often used; one use of `auto` is explicitly to distinguish a *declaration-statement* from an *expression-statement* (§6.2).

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.

An object declaration is a definition unless it contains the `extern` specifier and no initializer (§3.1). A definition causes the appropriate amount of storage to be reserved and any appropriate initialization (§8.4) to be done.

The `static` and `extern` specifiers can be applied only to names of objects or functions. There can be no `static` function declarations within a block, nor any `static` or `extern` formal arguments. Static class members are described in (§9.4); `extern` cannot be used for class members.

A name specified `static` has internal linkage. Functions declared `inline` and 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` or `inline`. 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 specifications
      { /* ... */ }

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 specifications

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

7.1.2 Function Specifiers

Some specifiers can only be used in function declarations:

fct-specifier:
```
        inline
        virtual
```

The `inline` specifier gives the function default internal linkage (§3.3); it is also 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. A function (§5.2.2, §8.2.5) defined within the declaration of a class is `inline` by default.

A global inline function may explicitly be given external linkage provided it is not called between the `extern` declaration and the `inline` definition. For example:

```
extern f();
extern g();

h() { return f(); }

inline f() { /* ... */ }        // error
inline g() { /* ... */ }        // ok
```

Here `g()` has external linkage but can be inlined in the (one) source file where its definition appears.

A class member function need not be declared `inline` in the class declaration to be inline. Where 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 h(X* p)
{
      int i = p->f();        // now X::f() has external linkage
      int j = p->g();
      // ...
}

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

The `virtual` specifier may be used only in declarations of non-static class member functions within a class declaration; see §10.2.

7.1.3 The `typedef` Specifier

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 *fct-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 legal declarations; the type of `distance` is `int`; that of `metricp` is "pointer to `int`."

A `typedef` may be used to re-define a name to refer to the type it already referred to — 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;
```

A `typedef` may be not be used to re-define 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 of 'complex'
```

A *typedef-name* that names a class is a *class-name* (§9.1).

7.1.4 The `template` Specifier

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

7.1.5 The `friend` Specifier

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

7.1.6 Type Specifiers

The type-specifiers are:

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

The words `const` and `volatile` may be added to any legal 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. 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 member of a `const` class object is `const` (§9.3.1).

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 member of a `volatile` class object is `volatile` (§9.3.1).

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

> *simple-type-name:*
> > *class-name*
> > *typedef-name*
> > `char`
> > `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 be mentioned 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 is used to force `char` objects and bit-fields to be signed; it is redundant with other integral types.

class-specifiers and *enum-specifiers* are discussed in §9 and §7.2, respectively.

> *elaborated-type-specifier:*
> > *class-key class-name*
> > *class-key identifier*
> > `enum` *enum-name*
>
> *class-key:*
> > `class`
> > `struct`
> > `union`

If an *identifier* is specified, the *elaborated-type-specifier* declares it to be a *class-name;* see §9.1.

If defined, a name declared using the `union` specifier must be defined as a union. If defined, a name declared using the `class` specifier must be defined using the `class` or `struct` specifier. If defined, a name declared using the `struct` specifier must be defined using the `class` or `struct` specifier.

7.2 Enumeration Declarations

An enumeration is a distinct integral 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*$_{opt}$ `{` *enum-list*$_{opt}$ `}`
>
> *enum-list:*
> > *enumerator*
> > *enum-list* `,` *enumerator*
>
> *enumerator:*
> > *identifier*
> > *identifier* `=` *constant-expression*

The identifiers in an *enum-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 0 and increase by 1 as the declaration is read from left to right. An enumerator with = gives the associated identifier the value indicated; subsequent identifiers without initializers continue the progression from the assigned value. The value of an enumerator must be an int or a value that can be promoted to int by integral promotion (§4.1).

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 0, b and e to be 1, and f to be 3.

Each enumeration defines an integral type that is different from all other integral types. The type of an enumerator is its enumeration. 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 an integral 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 int 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 color to int)
int i = yellow;   // ok: yellow converted to int value 1
                  //     (integral promotion)
```

See also §B.3.

Enumerators defined in a class (§9) are in the scope of that class and can only be referred to outside member functions of that class by explicit qualification with the class name (§5.1). However, the name of the enumeration itself is not 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)
{
      p->f(left);      // error
      p->f(X::right);  // ok
}

direction d = X::left;
```

7.3 Asm Declarations

An asm declaration has the form

asm-declaration:
    `asm (` *string* `) ;`

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

Linkage (§3.3) to non-C++ code fragments can be achieved using a *linkage-specification*:

linkage-specification:
    `extern` *string* `{` *declaration-list*$_{opt}$ `}`
    `extern` *string declaration*

declaration-list:
    declaration
    declaration-list declaration

The *string* is used to indicate what kind of linkage is required. The meaning of the string 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
}
```

Linkage specifications nest. A linkage specification does not establish a scope. A *linkage-specification* may only occur in *file* scope (§3.2). A *linkage-specification* for a class applies to non-member functions first declared within it. A *linkage-specification* for a function also applies to functions first declared within it. A linkage declaration with a string that is unknown to the implementation is an error.

If a function has more than one *linkage-specification*, they must agree; that is, they must specify the same *string*. 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.

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

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. In that case, the linkage directive is ignored for that 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 an error (§7.1.1).

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 sufficiently similar can such linkage be achieved.

Where the name of a programming language is used to name a style of linkage in the *string* 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

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

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

The two components of a *declaration* are the specifiers (*decl-specifiers*; §7.1) and the declarators (*declarator-list*). The specifiers indicate the fundamental type, storage class, etc., 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.

Declarators have the syntax:

> *declarator:*
>> *dname*
>> *ptr-operator declarator*
>> *declarator* `(` *argument-declaration-list* `)` *cv-qualifier-list*$_{opt}$
>> *declarator* `[` *constant-expression*$_{opt}$ `]`
>> `(` *declarator* `)`
>
> *ptr-operator:*
>> `*` *cv-qualifier-list*$_{opt}$
>> `&` *cv-qualifier-list*$_{opt}$
>> *class-name* `::` `*` *cv-qualifier-list*$_{opt}$
>
> *cv-qualifier-list:*
>> *cv-qualifier cv-qualifier-list*$_{opt}$
>
> *cv-qualifier:*
>> `const`
>> `volatile`
>
> *dname:*
>> *name*
>> *class-name*
>> `~` *class-name*
>> *typedef-name*

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

8.1 Type Names

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

> *type-name:*
>> *type-specifier abstract-declarator*$_{opt}$

abstract-declarator:
  ptr-operator abstract-declarator$_{opt}$
  abstract-declarator$_{opt}$ (*argument-declaration-list*) *cv-qualifier-list*$_{opt}$
  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 taking no arguments and returning pointer to integer," and "pointer to function taking a `double` argument and returning an integer."

8.2 Meaning of Declarators

A list of declarators appears after a (possibly empty) list of *decl-specifiers* (§7.1). Each declarator contains exactly one *dname*; it specifies the identifier that is declared. Except for the declarations of some special functions (§12.3, §13.4) a *dname* will be a simple *identifier*. An `auto`, `static`, `extern`, `register`, `friend`, `inline`, `virtual`, or `typedef` specifier applies directly to each *dname* in a *declarator-list*; the type of each *dname* depends on both the *decl-specifiers* (§7.1) and its *declarator*.

Thus, a declaration of a particular identifier has the form

```
T D
```

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

In a declaration where `D` has the form

```
( D1 )
```

the type of `D1` is the same as that of `D`. Parentheses do not alter the type of the embedded *dname*, but they may alter the binding of complex declarators.

8.2.1 Pointers

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

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

the type of the contained identifier is "... *cv-qualifier-list* pointer to `T`." The *cv-qualifier*s apply to the pointer and not to the object pointed to.

For example, the declarations

```
const ci = 10, *pc = &ci, *const cpc = pc;
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; `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 legal operations are:

```
i = ci; *cp = ci; pc++; pc = cpc; pc = p;         // ok
```

Examples of illegal operations are:

```
ci = 1; ci++; *pc = 2; cp = &ci; cpc++; p = pc;   // errors
```

Each is illegal because it would either change the value of an object declared `const` or allow it to be changed through an unqualified pointer later.

`volatile` specifiers are handled similarly.

See also §5.17 and §8.4.

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

8.2.2 References

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

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

the type of the contained identifier is "... *cv-qualifier-list* reference to `T`." The type `void&` is not permitted.

For example,

```
void f(double& a) { a += 3.14; }
// ...
double d = 0;
f(d);
```

declares `a` to be a reference argument of `f` so that 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 that `g(3)=7` will assign `7` to the fourth element of the array `v`.

```
void h(link*& p)
{
      p->next = first;
      first = p;
      p = 0;
}
// ...
link* q = new link;
h(q);
```

declares `p` to be a reference to a pointer to `link` so that `h(q)` will leave `q` with the value `0`. See also §8.4.3.

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), or is a class member (§9.2) declaration within a class declaration, the declaration of an argument type, or the declaration of a return type (§8.2.5).

8.2.3 Pointers to Members

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

> *class-name* `::` `*` *cv-qualifier-list$_{opt}$* `D1`

the type of the contained identifier is "... *cv-qualifier-list* pointer to member of class *class-name* of type `T`."

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
```

Note that a pointer to member cannot point to a static member of a class (§9.4). See also §5.5 and §5.3.

8.2.4 Arrays

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

> `D1 [`*constant-expression$_{opt}$*`]`

then the contained identifier has type "... array of `T`." If the *constant-expression* (§5.19) is present, it must be of integral type and have a value greater than `0`. The constant expression specifies the number of elements in the array. If the constant expression is `N`, the array has `N` elements numbered `0` to `N-1`.

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.

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

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.

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.

A consistent rule is followed in the case of multi-dimensional 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.

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) 5-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 5 integer objects. The results are added and indirection applied to yield an array (of 5 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.

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

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

`D1 (` *argument-declaration-list* `)` *cv-qualifier-list*$_{opt}$

the contained identifier has the type "... *cv-qualifier-list*$_{opt}$ function taking arguments of type *argument-declaration-list* and returning `T`."

argument-declaration-list:
  arg-declaration-list$_{opt}$ `...`$_{opt}$
  arg-declaration-list `, ...`

arg-declaration-list:
  argument-declaration
  arg-declaration-list `,` *argument-declaration*

argument-declaration:
  decl-specifiers declarator
  decl-specifiers declarator `=` *expression*
  decl-specifiers abstract-declarator$_{opt}$
  decl-specifiers abstract-declarator$_{opt}$ `=` *expression*

If the *argument-declaration-list* terminates with an ellipsis, the number of arguments is known only to be equal to or greater than the number of argument types specified; if it is empty, the function takes no arguments. The argument list `(void)` is equivalent to the empty argument list. Except for this special case `void` may not be an argument type (though types derived from `void`, such as `void*`, may). Where legal, "`, ...`" is synonymous with "`...`". The standard header `<stdarg.h>` contains a mechanism for accessing arguments passed using the ellipsis.

A single name may be used for several different functions in a single scope; this is function overloading (§13.1). All declarations for a function taking a given set of arguments must agree exactly both in the type of the value returned and in the number and type of arguments; the presence or absence of the ellipsis is considered part of the function type. Argument types that differ only in the use of typedef names or unspecified argument array bounds agree exactly. The argument types, but not the default arguments (§8.2.6), are part of the function type. A *cv-qualifier-list* can be part of a member function declaration, member function definition, or pointer to member function only; see §9.3.1. It is part of the function type.

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.

The *argument-declaration-list* is used to check and convert actual arguments in calls and to check pointer-to-function assignments and initializations.

An identifier can optionally be provided as an argument name; if present in a function declaration, it cannot be used since it immediately goes out of scope; if present in a function definition (§8.3), it names a formal argument. In particular, argument names are also optional in function definitions and names used for an argument in different declarations and the definition of a function need not be the same.

The declaration

```
int i, *pi, f(), *fpi(int), (*pif)(const char*, const char*);
```

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, and a pointer `pif` to a function which takes two pointers to constant characters and returns an integer. It is especially useful to compare the last two. The binding of `*fpi(int)` is `*(fpi(int))`, so that 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.

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 number and types of arguments. For example:

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

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

8.2.6 Default Arguments

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

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.

Default argument expressions have their names bound and their types checked at the point of declaration, and are evaluated at each point of call. In the following example, `g` will be called with the value `f(2)`:

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

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

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

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

Similarly, the declaration of `X::mem1()` in the following example is illegal because no object is supplied for the non-static member `X::a` used as an initializer:

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

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

A default argument is not part of the type of a function:

```
int f(int = 0);

h()
{
        f(1);
        f();            // fine, means f(0)
}

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

8.3 Function Definitions

Function definitions have the form

fct-definition:
    decl-specifiers$_{opt}$ *declarator* *ctor-initializer*$_{opt}$ *fct-body*

fct-body:
    compound-statement

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

`D1 (` *argument-declaration-list* `)` *cv-qualifier-list*$_{opt}$

as described in §8.2.5.

The formal arguments are in the scope of the outermost block of the *fct-body*.

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-specifiers*; `max(int a, int b, int c)` is the *declarator*; `{ ... }` is the *fct-body*.

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

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

Note that unused formal arguments need not be named. For example:

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

8.4 Initializers

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

initializer:
  = *assignment-expression*
  = { *initializer-list* , *opt* }
  (*expression-list*)

initializer-list:
  expression
  initializer-list , *expression*
  { *initializer-list* , *opt* }

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

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

A pointer of type `const T*`, that is, a pointer to constant `T`, can be initialized with a pointer of type `T*`, but the reverse initialization is illegal. 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 non-const
                       // point to a const

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

The reason for the two errors is the same: had those initializations been allowed they would have allowed the value of something declared `const` to be changed through an unqualified pointer.

Default argument expressions are more restricted; see §8.2.6.

Variables with storage class static (§3.5) that are not initialized are guaranteed to start off as `0` converted to the appropriate type. The initial values of automatic and register variables that are not initialized are undefined.

When an initializer applies to a pointer or an object of 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.

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`.

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 §6.3.

8.4.1 Aggregates

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-list* 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 0's.

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 `0`.

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).

Braces may be elided as follows. If the *initializer-list* 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-list* or a sub-aggregate 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.

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 0's. 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).

The initializer for `y` begins with a left brace, but that 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 `0`.

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

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
```

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 an error.

8.4.2 Character Arrays

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

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

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

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

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

is an error since there is no space for the implied trailing `'\0'`.

8.4.3 References

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:

```
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'
```

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.

The initializer may be left out for a reference only in an argument 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
```

If the initializer for a reference to type `T` is an lvalue of type `T` or of a type publicly derived (§10) from `T` the reference will refer to the initializer; otherwise, an object of type `T` will be created and initialized with the initializer. The reference then becomes a name for that object. The lifetime of an object created in this way is the scope in which it is created (§3.5). For example:

```
double& rd = 1;
```

is legal and `rd` will point to a `double` containing the value `1.0`. Also,

```
char c = 'c';
char& rc = c;
signed char& rsc = c;
unsigned char& ruc = c;
```

Here, either `rsc` or `ruc` (but not both) will be initialized to a temporary dependent on whether 'plain' `char` is signed or unsigned. The initialization of `rc` does not require the use of a temporary.

Note that a reference to a class `B` can be initialized by an object of a class `D` provided `B` is a public base class of `D` (in that case a `D` is a `B`); see §4.7.

9. Classes

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

> *class-name:*
>   *identifier*

*Class-specifier*s and *elaborated-type-specifier*s 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-list*$_{opt}$ }
>
> *class-head:*
>   *class-key identifier*$_{opt}$ *base-spec*$_{opt}$
>   *class-key class-name base-spec*$_{opt}$
>
> *class-key:*
>   `class`
>   `struct`
>   `union`

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

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.

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

A class declaration 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 an error because it defines `S` twice.

A class declaration introduces the class name into the scope where it is declared and hides any class, object, function, etc., 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 only be referred to using an *elaborated-type-specifier* (§7.1.6). For example,

```
struct stat {
      // ...
};

int stat(struct stat*);

void f()
{
      struct stat s;  // 'struct' needed to name struct
      stat(&s);       // call of the function stat()
}
```

This re-use of a class name is not legal for classes with constructors (§12.1). An *elaborated-type-specifier* with a *class-key* used without declaring an object or function introduces a class name exactly like a class declaration but without declaring 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 declaration of classes that refer to each other. For example:

```
class vector;

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

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

Declaration of `friend`s is described in §11.4, operator functions in §13.4. If a class mentioned as a `friend` has not been declared its name is entered in the same scope as the name of the class containing the friend declaration (§11.4).

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()
{
        struct s* p;     // refer to global 's'
        p->a = 1;

}
```

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.

A *typedef-name* (§7.1.3) that names a class is a *class-name*.

9.2 Class Members

member-list:
    member-declaration member-list$_{opt}$
    access-specifier : member-list$_{opt}$

member-declaration:
    decl-specifiers$_{opt}$ member-declarator-list$_{opt}$;
    function-definition ;$_{opt}$
    qualified-name ;

member-declarator-list:
    member-declarator
    member-declarator-list member-declarator

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

pure-specifier:
    = 0

A *member-list* may declare data, function, class, enum (§7.2), bit-field members (§9.6), friends (§11.4), and type names (§7.1.3, §9.1). A *member-list* may also contain declarations adjusting the access to member names; see §11.3. A member may not be declared twice in the *member-list*. The *member-list* defines the full set of members of the class. No member can be added elsewhere.

Note that a single name can denote several function members provided their types are sufficiently different (§13) and that a *member-declarator* cannot contain an *initializer* (§8.4).

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

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

Members that are class objects must be objects of previously declared classes. In particular, a class `c1` may not contain an object of class `c1`, but it may contain a pointer or reference to an object of class `c1`.

A simple example of a class declaration is:

```
struct tnode {
      char tword[20];
      int count;
      tnode *left;
      tnode *right;
};
```

which contains an array of 20 characters, an integer, and two pointers to similar structures. Once this declaration 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` field 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`.

Data members of a class within the same *access-specifier* are allocated so that later members have higher addresses within a class object. The order of allocation across *access-specifier*s is implementation dependent. 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.

9.3 Member Functions

A function declared as a member (without the `friend` specifier; §11.4) is called a member function, and is called 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, n2;
      n1.set("abc",&n2,0);
      n2.set("def",0,0);
}
```

The definition of a member function is considered to be within the scope of its class. This means that it (provided it is non-static §9.4) can use names of members of its class directly. If the definition of a member function is lexically outside the class declaration, 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);
        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.

Members may only be defined (§3.1) outside their class declaration (provided they have already been declared in the class declaration); they may not be redeclared.

The effect of calling a non-static 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

In a non-static (§9.3) member function, the keyword `this` is a pointer to the object for which the function is called. The type of `this` in a member function of a class `X` is `X *const` unless the member function is declared `const` or `volatile`; in those cases, the type of `this` is `const X *const` and `volatile X *const`, respectively. See also §B.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 an error 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`.

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 for a non-`const` object only. For example,

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

The call `y.g()` is an error 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.

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

Constructors (§12.1) and destructors (§12.4) need not be `const` member functions to be used for `const` objects.

9.3.2 Inline Member Functions

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

Member functions can be defined even in local or nested class declarations where this rewriting would be syntactically illegal. In these cases the function definition is handled as if it occurred outside any function or class. It is still considered within the scope of its class, however. For example:

```
void f()
{
        int b;
        struct x {
                int mem;
                int  f() { return mem; }  // ok
                int  g() { return b; }    // error
        };
        // ...
}
```

The `b` used in `x::g()` is undefined because, appearances to the contrary, `x::g()` is not in the scope of `::f()` (§3.2).

The following rule limits the context sensitivity of the rewrite rules for inline functions, typedefs and nested classes (§9.7), and for class member declarations in general. A *class-name* or a *typedef-name* may not be redefined in a class declaration after being used in the class declaration, nor may a name that is not a *class-name* or a *typedef-name* be redefined to a *class-name* or a *typedef-name* in a class declaration after being used in the class declaration. For example:

```
typedef int c;
class X {
      int f() { return sizeof(c); }
      char c;       // error: typedef name redefined after use
};
```

9.4 Static Members

A data or function member of a class may be declared `static` in the class declaration. There is only one copy of a static data member, shared by all objects of the class in a program. A static member is not part of objects of a class. Static members, except inline functions, of a global class have external linkage (§3.3). The declaration of a static member in its class declaration is *not* a definition. A definition is required elsewhere; see also §B.3.

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

Static members of a class declared local to some function have no linkage and cannot be initialized. It follows that a local class cannot have static members that require initialization (except for the default "all zeros" initialization (§8.4)).

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); in this case, the expression on the left hand side of `.` or `->` is not evaluated. It exists even if no objects of class `cl` have been created. For example,

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

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

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 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

A union may be thought of as a structure whose member objects all begin at offset 0 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). 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 cannot be a member of a union. A union can have no `static` data members.

A union of the form

    union { member-list } ;

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 = "asdf";
        // ...
}
```

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

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

A union may not have function members.

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 illegal since the member name is not associated with any particular object.

9.6 Bit-Fields

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.

An unnamed bit-field is useful for padding to conform to externally-imposed layouts. As a special case, an unnamed bit-field with a width of 0 specifies alignment of the next bit-field at an allocation unit boundary.

A bit-field must have integral type (§3.6.1). It is implementation dependent whether a "plain" `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. There are no references to bit-fields either.

9.7 Nested Class Declarations

A class may be declared within another class. That, however, is only a notational convenience since the inner class has the same scope as the enclosing class. For example:

```
int x;

class enclose {
      int x;
      class inner {
            int y;
            void f(int);
      };
      int g(inner*);
};

inner a;
void inner::f(int i) { x = i; } // assign to ::x
int enclose::g(inner* p) { return p->y; } // error
```

The definition of `enclose::g()` is an error since `inner::y` is private and `enclose::g()` has no special access rights to it despite class `inner` being declared within class `enclose`.

Similarly, typedefs (§7.1.3) and enumeration names (§7.2) declared within a class can be used outside. For example,

```
class X {
      typedef int I;
      I a;
};

I b;
```

The "exporting" of classes and typedefs and enumeration names into an enclosing scope continues until the global or a local (block) scope is reached.

10. Derived Classes

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

> *base-spec:*
> : *base-list*
>
> *base-list:*
> *base-specifier*
> *base-list* , *base-specifier*
>
> *base-specifier:*
> *class-name*
> `virtual` *access-specifier*$_{opt}$ *class-name*
> *access-specifier* `virtual`$_{opt}$ *class-name*
>
> *access-specifier:*
> `private`
> `protected`
> `public`

The *class-name* in a *base-specifier* must denote a previously declared class (§9), which is called a base class for the class being declared. A class is said to be derived from its base classes. For the meaning of *access-specifier* see §11. `protected` may not be used as the *access-specifier* in a *base-specifier* for a base class. Unless re-defined 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. A base member can also be referred to explicitly using the `::` operator (§5.1). This allows a name that has been re-defined in the derived class to be accessed. 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 a public base class (§4.6). A reference to a derived class may be implicitly converted to a reference to a public base class (§4.6).

For example:

```
class base {
public:
        int a, b;
};

class derived : public base {
public:
        int b, c;
};

void f()
{
        derived d;
        d.a = 1;
        d.base::b = 2;
        d.b = 3;
        d.c = 4;
        base* bp = &d;  // standard conversion: derived* to base*
}
```

assigns to the four members of `d` and makes `bp` a pointer to `d`.

A class is called a *direct base* if it is mentioned in the *base-list*, and an *indirect base* if it is not but is a base class of one of the classes mentioned in the *base-list*.

Note that in the *class-name* :: *name* notation, *name* may be a name of a member of an indirect base class; the notation simply specifies a class in which to start looking for *name*. For example:

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

int C::f()
{
        B::f(); // call A's f()
}
```

Here, `A::f()` is called since it is the only `f()` in `B`.

10.1 Multiple Base Classes

A class may be derived from any number of base classes (i.e., multiple inheritance):

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

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). The order in which storage is allocated for base classes is implementation dependent.

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 : B, B { ... };   // illegal

class X : public B { ... };
class Y : public B { ... };
class Z : public X, public Y { ... };    // legal
```

In this case an object of class `Z` will have two sub-objects of class `B`. Note that every access to a member of `B` in a `D` would have been ambiguous (§10.1.1).

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 X : virtual public B { ... };
class Y : virtual public B { ... };
class Z : public X, public Y { ... };
```

Here class `Z` has only one sub-object of class `B`.

A class may have both virtual and non-virtual base classes of a given type:

```
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`.

10.1.1 Ambiguities

Access to base class members must be unambiguous. Access to a base class member is ambiguous if the expression used refers to more than one function, object, or enumerator. The check for ambiguity takes place before access control (§11) and before type checking.

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

class B {
    int a;
public:
    int f;
};

class C : public A, public B {};

void g(C* pc)
{
    pc->a = 1;  // error, ambiguous: A::a or B::a ?
    pc->f();    // error, ambiguous: A::f or B::f ?
}
```

Ambiguities can be resolved by qualifying a name with its class name:

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

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

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

When virtual base classes are used, a function, object, or enumerator may be reached through more than one path through the directed acyclic graph of base classes. This is not an ambiguity. The identical use with non-virtual base classes is an ambiguity; in that case more than one sub-object is involved. For example:

```
class V { public: int v; };
class A { public: int a; };
class B : public A, public virtual V {};
class C : public A, public virtual V {};

class D : public B, public C { public: void f(); };

void D::f()
{
      v++;  // ok: only one 'v' in 'D'
      a++;  // error, ambiguous: two 'a's in 'D'
}
```

When virtual base classes are used, more than one function, object, or enumerator may be reached through paths through the directed acyclic graph of base classes. For objects and enumerators this is an ambiguity. For functions it is not in all cases.

A function `B::f()` *dominates* a function `A::f()` if its class `B` has `A` as a base. If a function dominates another no ambiguity exists between the two and the dominant function is used where there is a choice. For example:

```
class A { public: int f(); int x; };
class B : public virtual A { public: int f(); int x; };
class C : public virtual A { };

class D : public B, public C { void g(); };

void D::g()
{
        x++;  // error, ambiguous: A::x or B::x ?
        f();  // ok: B::f() dominates A::f()
}
```

An explicit or implicit conversion from a derived class to one of its base classes must unambiguously refer to the same 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

If a class `base` contains a `virtual` (§7.1.1) function `vf`, and a class `derived` derived from it also contains a function `vf` of the same type then a call of `vf` for an object of class `derived` invokes `derived::vf` (even if the access is through a pointer or reference to `base`). The derived class function is said to *override* the base class function. If the function types (§8.2.5) are different, however, the functions are considered different and the `virtual` mechanism is not invoked (see also §13.1). It is an error for a derived class function to differ from a virtual base class function in the return type only. For example:

```
struct base {
        virtual void vf1();
        virtual void vf2();
        virtual void vf3();
        void f();
};
```

```
class derived : public base {
public:
      void vf1();
      void vf2(int);
      char vf3();      // error: differs in return type only
      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
}
```

The calls invoke `derived::vf1`, `base::vf2`, and `base::f`, respectively, for the class `derived` object named `d`. That is, the interpretation of the call of a virtual function depends on the type of the object for which it is called, whereas the interpretation of a call of a non-virtual member function depends only on the type of the pointer or reference denoting that object.

The `virtual` specifier implies membership, so a `virtual` function cannot be a global (non-member) (§7.1.1) function. A virtual function cannot be a `static` member either 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. An overriding function is itself considered virtual. The `virtual` specifier may be used for an overriding function in the derived class, but such use is redundant. A virtual function in a base class must be defined or declared pure (§10.3). A virtual function that has been defined in a base class need not be defined in a derived class. In that case, the function defined for the base class is used in all calls.

10.3 Abstract Classes

An *abstract class* is a class that can only be used as a base class of some other class; no objects of an abstract class may be created except as sub-objects of some other class. A class is abstract if it has at least one *pure virtual function*. 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 only be defined if explicitly called with the *qualified-name* syntax (§5.1). For example:

```
class shape {          // abstract class
      point center;
      // ...
public:
      where() { return center; }
      move(point p) { center=p; draw(); }
      virtual void rotate(int) = 0;  // pure virtual function
      virtual void draw() = 0;       // pure virtual function
      // ...
};
```

An abstract class may not be used as an argument type or as a function return type. 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
```

However, references that require the use of a temporary in the initialization (§8.4.3) are illegal.

A class with an abstract class `A` as its immediate base class must define or declare *pure* every *pure* virtual function in `A`.

```
class circle : public shape {
      int radius;
public:
      void rotate(int) {}
      void draw();    // circle::draw() must be defined somewhere
};
```

Member functions can be called from a constructor of an abstract class; calling a pure virtual function directly or indirectly from such a constructor causes a run-time error.

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.

10.4 Summary of Scope Rules

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).

When looking for the object, function, type, etc., that a name refers to, at first the name only is considered and ambiguities are detected (§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, etc., considered.

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.

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`.

A name that is not qualified in any of the ways described above and that is used in a function that is not a non-static class member must be declared in the block in which it occurs or in an enclosing block or be a global name. The declaration of a local name hides declarations of the same name in enclosing blocks and global names. In particular, no overloading occurs of names in different scopes (§13.4).

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

A function argument name in a function definition (§8.3) is in the scope of the outermost block of the function. A function argument name in a function declaration (§8.2.5) that is not a function definition is in no scope at all. A default argument is in the scope determined by the point of declaration (§3.2) of its argument, but may not access local variables or non-static class members; it is evaluated at each point of call (§8.2.6).

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 argument names.

11. Member Access Control

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.

Members of a class declared with the keyword `class` are `private` by default. Members of a class declared with the keyword `struct` 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
};
```

All members of a union are `public` and no *access-specifier* (§11.1) is allowed in a union.

11.1 Access Specifiers

Member declarations may be labelled by an *access-specifier* (§10):

access-specifier : *member-list*$_{opt}$

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
};
```

The order of allocation of data members with separate *access-specifier*s is implementation dependent (§9.2).

11.2 Access Specifiers for Base Classes

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 `public` members of the derived class and `protected` members of the base class are `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 `private` members of the derived class. Private members of a base class remain inaccessible even to derived class members unless `friend` declarations within the base class declaration are used to explicitly grant access. `protected` cannot be used as the *access-specifier* in a *base-specifier* for a base class.

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 D1 : private base { ... };
class D2 : public base { ... };
class D3 : base { ... };    // 'base' private by default
struct D4 : public base { ... };
struct D5 : private base { ... };
struct D6 : base { ... };   // 'base' public by default
```

Here `base` is a public base of `D2` and `D4`, and `D6`, and a private base of `D1`, `D3`, and `D5`.

11.3 Access Declarations

The access to a member of a base class in a derived class can be adjusted by mentioning its *qualified-name* in the `public` or `protected` part of a derived class declaration. Such mention is called an *access declaration*.

For example:

```
class base {
    int a;
public:
    int b, c;
    int bf();
};

class derived : private base {
    int d;
public:
    base::c;  // adjust access to 'base::c'
    int e;
    int df();
};

int ef(derived&);
```

The external function `ef` can use only the names `c`, `e`, and `df`. Being a member of `derived`, the function `df` can use the names `b`, `c`, `bf`, `d`, `e`, and `df`, but not `a`. Being a member of `base`, the function `bf` can use the members `a`, `b`, `c`, and `bf`.

This notation may not be used to prevent 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 B {
public:
      int a;
private:
      int b;
protected:
      int c;
};

class D : private B {
public:
      B::a;  // make 'a' a public member of D
      B::b;  // make 'b' a public member of D
             // error: attempt to grant access
protected:
      B::c;  // make 'a' a protected member of D
      B::a;  // make 'a' a protected member of D
             // error: attempt to reduce access
};
```

An access declaration for the name of an overloaded function adjusts the access to all functions of that name in the base class.

11.4 Friends

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. A friend is not in the scope of the class and 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; }

X obj;
friend_set(&obj,10);
obj.member_set(10);
```

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

When a `friend` declaration refers to an overloaded name or operator, only the function specified by the argument types becomes a friend. A member function of a class `X` can be a friend of a class `Y`. For example,

```
class X {
      friend char* Y::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* (§9.1):

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

If a class mentioned as a friend has not been declared its name is entered in the same scope as the name of the class containing the friend declaration (§9.1).

A `friend` function defined in a class declaration is `inline` and the re-writing rule specified for member functions (§9.3.2) is applied.

Friendship is inherited but not transitive. For example:

```
class A {
friend B;
      int a;
};

class B {
friend C;
};

class C  {
      void f(A* p) { p->a++; } // error
};
```

11.5 Protected Member Access

A friend or a member function of a derived class can access any public or protected static member of a publicly derived base class. A friend or a member function of a derived class can access a protected non-static member of a base class only through a pointer to, reference to, or object of the derived class of which it is a friend or member (and pointers to, references to, and objects of classes publicly derived from this 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)
}
```

```
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

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 base {
public:
      virtual f();
};

class derived : public base {
private:
      f();
};

void f()
{
      derived d;
      base* pb = &d;
      derived* pd = &d;

      pb->f();          // ok: base::f() is public,
                        // derived::f() is invoked
      pd->f();          // error: derived::f() is private
}
```

Access is checked at the call point.

11.7 Multiple Access

If a name can be reached through several paths through a multiple inheritance graph, the access is that of the path that gives most access. For example:

```
class W { public: f(); };
class A : private W { };
class B : public W { };
class C : public A, public B { f() { W::f(); } };   // ok
```

Since `W::f()` is available to `C::f()` along the public path through `B` access is legal.

12. Special Member Functions

Some member functions are special in that they affect the rules for how objects of a class are created, copied, and destroyed, and how values may be converted to values of other types. In many cases such special functions are called implicitly.

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

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.

A constructor can be invoked for a `const` or `volatile` object even if it is not declared to be a `const` or `volatile` member function (§9.3.1). A constructor may not be `virtual`.

A *default constructor* for a class `X` is a constructor that can take no argument, that is, of the form `X::X()`. A default constructor will not be generated for a class `X` if any constructor has been declared for class `X`. A constructor that can be called with no arguments because of default arguments, for example `X::X(int=0)`, is not a default constructor.

A *copy constructor* for a class `X` is a constructor that can be called to copy an object of class `X`; that is, one that 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 only generated if needed (§12.8) and if no copy constructor is declared.

Constructors are not inherited. However, default constructors and copy constructors are generated (by the compiler) where needed (§12.8). Generated constructors are `public`.

A constructor for a class `X` may not take an argument of type `X`. For example, `X::X(X)` is illegal.

Constructors for array elements are called in increasing subscript order.

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.

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

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.

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

typedef-name (*argument-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.

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

12.2 Temporary Objects

When compiling C++ it is sometimes necessary and sometimes just convenient (for the compiler) to introduce a temporary variable. The use of such temporary variables is implementation dependent. When a compiler induces a temporary variable of a class that has a constructor then it must ensure that a constructor is called for the temporary variable. Similarly, the destructor must be called for an object of a class where a destructor is defined. For example,

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

The compiler must ensure that a temporary object is destroyed. There are only two things that can be done with a temporary: fetch its value (implicitly copying it) to use in some other expression, or bind a reference to it. If the value of a temporary is fetched, that temporary is then dead and can be destroyed immediately. If a reference is bound to a temporary, the temporary must not be destroyed until the reference is. This destruction must take place before exit from the scope in which the temporary is created.

Another form of temporaries is discussed in §8.4.3.

12.3 Conversions

Type conversions of class objects can be specified by constructors and by conversion functions.

Such conversions, often called *user-defined conversions*, are used implicitly in addition to standard conversions (§4). For example, an assignment to an object of class `X` is legal not only if the type `T` of the assigned value is `X`, but also if a conversion has been declared from `T` to `X`. 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 (§6.5, §6.4), and explicit type conversions (§5.2.3, §5.4).

See §13.2 for a discussion of the use of conversions in function calls as well as examples below.

A conversion may not be defined by both a constructor and a conversion function. For example:

```
class B;

class A {
      // ...
      A(const B&);
};

class B {
      // ...
      operator A();    // error: a conversion of a B to an A
                       //        is already declared
};
```

12.3.1 Conversion by Constructor

A constructor taking a single argument specifies a conversion from its argument type to the type of its class. For example:

```
class X {
      // ...
      X(int);
      X(const char*, int = 0);
};

f(X arg) {
      X a = 1;        // a = X(1)
      X b = "asdf";   // b = X("asdf",0)
      a = 2;          // a = X(2)
      f(3);           // f(X(3))
}
```

When no constructor for class `X` accepts the assigned type, no attempt is made to find other constructors 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

A member function of a class `X` with a name of the form

conversion-function-name:
  `operator` *conversion-type-name*

conversion-type-name:
  type-specifiers ptr-operator$_{opt}$

specifies a conversion from `X` to the type specified by the *conversion-type-name*. Neither argument types nor return type may be specified. The type of a conversion function from a type `F` to a type `T` is `T*F::()`.

Conversion operators are inherited.

An example:

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

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:

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

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

```
class X { ... operator int(); };
class Y { ... 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()
```

12.4 Destructors

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. It takes no arguments, and no return type can be specified for it. A destructor can be invoked for a `const` object even if it is not declared to be a `const` member function (§9.3.1).

Base class destructors are implicitly executed after the destructors for their derived classes. Destructors for non-virtual base classes are executed in reverse order of their declaration in the derived class. Destructors for virtual base classes are executed after destructors for non-virtual base classes in the reverse order of their appearance in a depth-first left-to-right traversal of the directed acyclic graph of base classes. Member destructors are implicitly executed in reverse order of their declaration in the class they are members of. A destructor for a member object is executed after the destructor for the object it is a member of. This rule applies recursively for virtual bases of virtual bases.

Destructors are not inherited. A destructor calling destructors for bases and members are generated for a derived class. Generated destructors are `public`.

Destructors for elements of an array are called in reverse order of their construction.

A destructor may be `virtual`.

It is not possible to take the address of a destructor.

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

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

Destructors are invoked implicitly when an `auto` (§3.5) or temporary (§12.2, §8.4.3) object goes out of scope and for `static` (§3.5) objects at program termination (§3.4). They can be invoked through use of the `delete` operator (§5.3.4) for objects allocated by the `new` operator (§5.3.3) and explicitly using the destructor's fully qualified name. When invoked by the `delete` operator, memory is freed by the destructor for the most derived class (§12.6.2) of the object using an operatordelete() (§5.3.4). For example,

```
class X { /* ... */ ~X(); };

void* operator new(size_t, void* p) { return p; }

void f(X* p);

static char buf[sizeof(X)];

void g()
{
        X* p = new X(args);      // allocate and initialize
        f(p);
        delete p;                // cleanup and deallocate

        p = new(&buf) X(args);   // use buf[] and initialize
        f(p);
        p->X::~X();              // cleanup
}
```

Explicit calls of destructors are only necessary for objects placed at specific addresses using a new operator as shown in the example. Such use of explicit placement and destruction of objects can be necessary to cope with dedicated hardware resources and for writing memory management facilities.

12.5 Free Store

When a class object is created with the new operator, an operator new() function is (implicitly) used to obtain the store needed (§5.3.3).

An X::operator new() for a class X is a static member (even if not explicitly declared static). Its first argument must be of type size_t, an implementation dependent integral type defined in the standard header <stddef.h>; it must return void*. For example:

```
class X {
        // ...
        void* operator new(size_t);
        void* operator new(size_t, arena*);
};
```

See §5.3.3 for the rules for selecting an operator new().

An X::operator delete() for a class X is a static member (even if not explicitly declared static) and must have its first argument of type void*; a second argument of type size_t may be added. It cannot return a value; its return type must be void. For example

```
class X {
        // ...
        void operator delete(void*);
};

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

If the two argument style is used, operator delete() will be called with a second argument indicating the size of the object being deleted. The size passed is determined by the (static) type of the pointer being deleted; that is, it will be correct either if the type of the pointer argument to the `delete` operator is the exact type of the object (and not, for example, just the type of base class) or if the type is that of a base class with a virtual destructor and a destructor was defined for the derived class.

The return type of the `operator delete()` used determines the type of a `delete` expression (§5.3.4).

The global `operator new()` and `operator delete()` are used for arrays of class objects. The number of elements must be specified when deleting an array (§5.3.3, §5.3.4). For example:

```
class X { ... ~X(); };
X* p = new X[size];
delete[size] p;
```

The `size` argument is assumed to be the number of elements in the array. What happens if this is not the case is undefined.

Since `X::operator new()` and `X::operator delete()` are `static` they cannot be `virtual`.

12.6 Initialization

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 be either 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

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. In this case, its value is used as the initial value for the object if it is of the object's class or of a class publicly derived from that class; otherwise it is used as an argument to a constructor. For example:

```
class complex {
public:
        complex();
        complex(double);
        complex(double,double);
        // ...
};

complex sqrt(complex,complex);
complex a(1);            // initialize by a call of
                         // complex::complex(double)
complex b = a;           // initialize by a copy of 'a'
complex c = complex(1,2); // initialize by a call of
                         // complex::complex(double,double)
complex d = sqrt(b,c);   // initialize by a call of
                         // sqrt(complex,complex);
complex e;               // initialize by a call of
                         // complex::complex()
complex z = 3;           // initialize by a call of
                         // complex::complex(double)
```

Overloading of the assignment operator = has no effect on initialization. If a copy constructor (§12.1) exists, it will be invoked when an object is initialized with another object of that class (as for `b` above), but not when an object is initialized with a constructor (as for `c` above).

Arrays of objects of a class with constructors use constructors in the 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-list* 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]` are initialized with `complex::complex(double,double)`, and `v[2]`, `v[4]`, and `v[5]` are initialized with `complex::complex()`.

An object of class `M` can be a member of a class `X` only (1) if `M` does not have a constructor, or (2) if `M` has a default constructor, or (3) if `X` has a constructor that 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.

Constructors for non-local 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

Initializers for base classes and for members 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:
  class-name (*expression-list*$_{opt}$)
  identifier (*expression-list*$_{opt}$)

The argument list is used to initialize the named non-static member or base class object. This is the only way to initialize `const` and reference members. For example:

```
struct base1 { base1(int); ... };
struct base2 { base2(int); ... };

struct derived : base1, base2 {
      derived(int);
      base1 b;
      const c;
};

derived::derived(int a)
: base2(a+1), base1(a+2), c(a+3), b(a+4)
{ /* ... */ }

derived d(10);
```

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

Virtual base classes constitute a special case. A virtual base is constructed before any of its derived classes. Virtual bases are constructed before any non-virtual bases and in the order they appear on a depth-first left-to-right traversal of the directed acyclic graph of base classes. This rule applies recursively for virtual bases of virtual bases.

A *complete object* is an object that is not a sub-object representing a base class. Its class is said to be the *most derived* class for 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-initializers* 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 { 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()
```

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

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 redefining it in a derived class. This ensures that unconstructed 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;
        void f() { r++; }  // disaster if r is uninitialized
        Y(int& rr) :r(rr) {}
};
```

A constructor for an abstract class may not call a member function for the object for which it was invoked.

12.8 Copying Class Objects

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). The programmer may define one or both of these. If not defined by the programmer, they will be defined as member-wise assignment and member-wise initialization of the members of `X`, respectively.

If all bases and members of a class `X` have copy constructors accepting `const` arguments the generated copy constructor for `X` will take a single argument of type `const X&`:

```
X::X(const X&)
```

otherwise it will take a single argument of type `X&`:

```
X::X(X&)
```

and copying of `const X` objects will not be possible.

Similarly, if all bases and members of a class `X` have assignment operators accepting `const` arguments the generated assignment operator for `X` will take a single argument of type `const X&`:

```
X& X::operator=(const X&)
```

otherwise it will take a single argument of type `X&`:

```
X& X::operator=(X&)
```

and copying of `const X` objects will not be possible. The default assignment operator will return a reference to the object for which is is invoked. Generated assignment operators are `public`.

Using member-wise assignment and member-wise initialization implies that if a class `X` has a member of a class `M`, `M`'s assignment operator and `M`'s copy constructor are used to implement assignment and initialization of the member, respectively. If a class has a constant member, a reference member, or a member of a class with a private `operator=()`, the default assignment operation cannot be generated. Similarly, if a member of a class M has a private copy constructor then the default copy constructor cannot be generated.

The default assignment and copy constructor will be declared, but they will not be defined (that is, a function body generated) unless actually needed. That is, `X::operator=()` will be generated only if no assignment operation is explicitly declared and an object of class X is actually 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.

If generated, assignment and 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.

If a class X has any `X::operator=()` defined, even one that takes an argument of a type unrelated to X, `X::operator=(const X&)` will not be generated. If a class has any copy constructor defined, `X(const X&)` will not be generated. For example:

```
class X {
      // ...
      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);
```

Assignment of class objects of class X is defined in terms of `X::operator(const X&)`. This implies 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.

Copying one object into another using the default copy constructor of 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
}
```

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`).

An object of a class `X` with a constructor cannot be passed as an argument to a function that does not supply an appropriate argument declaration (§8.2.5). For example,

```
class X {
      // ...
      X(X&);
};

int f(int ...);

void g(X a)
{
      f(1,a);     // error: no formal argument for object
              //       of class with copy constructor
}
```

13. Overloading

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 actual arguments with the formal argument types. 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 initializers, functions with argument 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 initializers, functions with argument 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.

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

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).

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;

f(int i) { /* ... */ }
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 };

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

Argument types that differ only in a pointer `*` vs an array `[]` are identical. Note that only the second and subsequent array dimensions are significant in argument 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

Two function declarations of the same name refer to the same function if they are in the same scope and have identical argument 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. So:

```
void h(D* pd)
{
      pd->f(1);        // error: D::f(char*) hides B::f(int)
      pd->B::f(1);     // ok
      pd->f("asdf");   // 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
}
```

Different versions of an overloaded member function may be given different access rules. For example:

```
class buffer {
private:
      char* p;
      int size;
protected:
      void buffer(int sz, char* store) { size = sz; p = store; }
      // ...
public:
      void buffer(int sz) { p = new char[size = sz]; }
      // ...
};
```

13.2 Argument Matching

A call of a given function name chooses, from among all functions by that name that are in scope, the function that best matches the actual 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 illegal.

For purposes of argument matching, a function with n default arguments (§8.2.6) is considered to be $n+1$ functions with different numbers of arguments.

For purposes of argument matching, a non-static member function is considered to have an extra argument specifying the object for which it is called. This extra argument requires a match by either 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). Where the member function is explicitly called for a pointer using the `->` operator, this extra argument is assumed to have type `const X*` for `const` members and `X*` for others. Where the member function is explicitly called for an object using the `.` operator or the function is invoked for the first operand of an overloaded operator (§13.4), this extra argument is assumed to have type `const X&` for `const` members and `X&` for others.

An ellipsis in a formal argument list (§8.2.5) is a match for an actual argument of any type except a type `T` for which `T(T&)` exists.

A function matches on an argument only if there exists an *admissible sequence* of conversions leading from the type of the actual argument to the type of the corresponding formal argument. A sequence is admissible unless it contains an admissible subsequence or more than one user-defined conversion. A best-matching sequence is an admissible sequence for which no better admissible sequence exists.

For example, `int->float->double` is a sequence of conversions from `int` to `double`, but it is not an admissible sequence because it contains the shorter admissible sequence `int->double`; thus the rule against admissible sub-sequences ensures that only shortest sequences of conversions are ever considered.

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`

Sequences of trivial conversions that differ only in order are indistinguishable.

A temporary variable is needed for a formal argument of type `T&` if the actual argument is not an lvalue, has a type different from `T`, or is a `const` and `T` isn't. In this case, a conversion of a `T` to a `T&` is not a trivial conversion but a conversion requiring a temporary (see [4] below).

Sequences of conversions are considered according to these rules:

[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 publicly 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&`.

[4] Match with conversions requiring temporaries: Of sequences not mentioned in [3], those that involve only conversions to `T&` requiring the use of temporary values (§8.4.3), user-defined (§12.3), standard (§4) and trivial conversions are better than all other sequences.

[5] Match with user-defined conversions: Of sequences not mentioned in [4], those that involve only user-defined conversions (§12.3), conversions to `T&` requiring the use of temporary values (§8.4.3), standard (§4) and trivial conversions are better than all other sequences.

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

13.2.1 Examples

Consider

```
f(char&);
f(short);
f(char*);

void g()
{
        f('c');      // call f(char&);
        char v[10];
        f(v);        // call f(char*);
        const char ch = 'c';
        f(ch);       // call f(char&);
        f(3);        // call f(short);
}
```

Here, temporary variables are needed for the first call of `f()` because `'c'` is a literal (of type `char`) and also for the third call because `ch` is a `const char` rather than the required `char`. Since `f(3)` can be interpreted as a call of `f(short)` without the use of a temporary variable, `f(short)` is preferred to `f(char&)` which requires the use of a temporary.

Note that `0` is of type `int` so that it is an exact match on an `int` argument but a match with standard conversions on arguments of types `short`, `double`, `char*`, etc.:

```
int f(char);
int f(double);

void g(short si)
{
        f('a');     // matches f(char)
        f(0);       // ambiguous: the int 0 can be converted
                    // to either char or double
        f(si);      // ambiguous: si can be converted
                    // to either char or double
        f(1.0f);    // matches f(double)
}
```

The last call is not ambiguous because the promotion from `float` to `double` is better than the standard conversion of a `float` to a `char`.

Integral promotions (§4.1) are implementation dependent:

```
int f(int);
int f(unsigned);

void g(unsigned short us)
{
        int i = f(us);
}
```

Here `f(int)` is called if `sizeof(short)<sizeof(int)`; otherwise `f(unsigned)` is called.

Enumerators are of the type of their enumeration (§7.2) and can be used in overloaded function calls:

```
enum e { A, B } ee;

void f(int);
void f(e);

void g()
{
        f(0);    // matches f(int)
        f(A);    // matches f(e)
        f(A+1); // matches f(int)
        f(ee);   // matches f(e)
}
```

It is possible to declare two functions that differ only in `const` in a pointer or reference argument:

```
void f(const int*);
void f(int*);

void g(const int a, int b) {
        f(&a);        // calls f(const int*)
        f(&b);        // calls f(int*)
}
```

This is especially important in the case of `const` objects and `const` member functions:

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

void g(const X& a, X b)
{
        a.f();        // calls X::f() const
        b.f();        // calls X::f()
}
```

An inheritance hierarchy defines a preference order for the standard pointer and reference conversions (§4.6, §4.7):

```
class A {};
class B : public A {};
class C : public B {};

void g(A*);
void g(B*);

C cc;

void f()
{
        g(&cc);       // ok: call g(B*)
}
```

In a sense, `void*` is the root of such hierarchies:

```
void h(void*);
void h(A*);

void hh()
{
        h(&cc);  // ok: call h(A*)
        h(0);    // error: ambiguous
}
```

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(); };

int f(long), f(char*);
int g(S), g(char*);
int h(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)))
}
```

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);
};

extern f(x);
extern f(y);

void g()
{
        f(1); // ambiguous
}
```

The call `f(1)` is ambiguous despite `f(y(long(1)))` needing one more standard conversion than `f(x(1))`.

If different combinations of standard or user-defined conversions are possible, the call is ambiguous:

```
extern f(int,long);
extern f(long,int);

void g()
{
      f(3,4);            // ambiguous
}
```

A call needing only standard conversions is preferred over one requiring user-defined conversions:

```
extern h(int,complex);
extern h(double,double);

void hh()
{
      h(3,4);            // ok: h(double(3),double(4))
}
```

However, where identical conversions exist for an argument, matching is determined by other arguments (if any):

```
extern k(int,complex);
extern k(double,complex);

void kk() {
      k(1,2);          // ok: k(1,complex(2)),
                       // that is, k(int,complex);

      k(1.0,2);        // ok: k(1.0,complex(2)),
                       // that is, k(double,complex);
}
```

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()
}
```

Default arguments can cause ambiguities:

```
int f();
int f(int i = 1);

void g()
{
      f(11);  // fine
      f();    // error, ambiguous: f() or f(1)
}
```

The ellipsis (...) can also cause ambiguities:

```
f(int);
f(int ...);
f(int, char* ...);

void g()
{
      f(1);              // error, ambiguous: f(int) or f(int ...)
      f(1,2);            // fine: f(int ...)
      f(1,"asdf");       // fine: f(int, char* ...)
      f(1,"asdf",2);     // fine: f(int, char* ...)
}
```

13.3 Address of Overloaded Function

A use of a function name without arguments picks out, among all functions of that name that are in scope, the (only) function that matches the target, which may be

an object being initialized (§8.4)

a parameter of a function (§5.2.2)

a parameter of a user-defined operator (§13.4)

the right side of an assignment (§5.17)

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)`.

For example:

```
int f(double);
int f(int);
int (*pfd)(double) = &f;
int (*pfi)(int) = &f;
int (*pfa)(...) = &f; // error: type mismatch
```

The last initialization is an error because no `f()` with type `int(...)` has been defined, and not because of any ambiguity.

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

Most operators can be overloaded.

operator-function-name:
```
    operator operator
```

operator: one of
```
    new  delete
    +    -    *    /    %    ^    &    |    ~
    !    =    <    >    +=   -=   *=   /=   %=
    ^=   &=   |=   <<   >>   >>=  <<=  ==   !=
    <=   >=   &&   ||   ++   --   ,    ->*  ->
    ()   []
```

The last two operators are function call (§5.2.2) and subscripting (§5.2.1).

Both the unary and binary forms of

```
    +    -    *    &
```

can be overloaded.

The following operators cannot be overloaded:

```
    .    .*   ::   ?:
```

nor can the preprocessing symbols # and ## (§16).

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);
```

The operators `new` and `delete` are described in §5.3.3 and §5.3.4 and the rules described below in this section do not apply to them.

An operator function must either be a member function or take at least one argument of class type. It is not possible to change the precedence, grouping, or number of operands of operators. The pre-defined meaning of the operators = and (unary) & applied to class objects may be changed. With the exception of `operator=()` operator functions are inherited; see §12.8 for a description of the rules for `operator=()`.

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 assignment, require an operand to be an lvalue when applied to basic types; this is not required when the operators are declared for class types.

13.4.1 Unary Operators

A unary operator, whether prefix or postfix, may be declared by a non-static member function (§9.3) taking no arguments or a non-member function taking one argument. Thus, for any unary operator `@`, both `x@` and `@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. When the operators ++ and -- are overloaded, it is not possible to distinguish prefix application from postfix application from within the overloading function.

13.4.2 Binary Operators

A binary operator may be declared either by a non-static member function (§9.3) taking one argument or by a non-member function taking two arguments. 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

The assignment function `operator=()` must be a non-static 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

Function call

primary-expression (*expression-list*$_{opt}$)

is considered a binary operator with the *primary-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 non-static member function.

13.4.5 Subscripting

Subscripting

primary-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 non-static member function.

13.4.6 Class Member Access

Class member access using ->

primary-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 either return a pointer to a class object or an object of a class for which `operator->()` is defined. `operator->` must be a non-static member function.

14. Templates (experimental)

<< The template design is experimental; see Bjarne Stroustrup: *Parameterized Types for C++*, Proc. USENIX C++ Conference, Denver, October, 1988. >>

15. Exception Handling (experimental)

<< The exception handling design is experimental >>

16. Compiler Control Lines

This section will be replaced with one based on the preprocessing defined for ANSI C. The obvious adjustment for the additional C++ keywords and tokens will be made. C++ style comments will be added so that the two styles of comments can be used to comment out each other as described in (§2.2), so that macro expansion will not take place within // comments, and so that // style comments appearing in macro definitions will be turned into whitespace before the macro is expanded.

The compiler contains a preprocessor capable of macro substitution, conditional compilation, and inclusion of named files.

The name `__cplusplus` is defined when compiling a C++ program.

Lines beginning with # communicate with this preprocessor. These lines have syntax independent of the rest of the language; they may appear anywhere and have effect which lasts (independent of scope) until the end of the source program file.

Note that `const` and `inline` definitions provide alternatives to many uses of `#define`.

16.1 Token Replacement

A compiler-control line of the form

`#define` *identifier token-string*

causes the preprocessor to replace subsequent instances of the identifier with the given string of tokens. Semicolons in, or at the end of, the token-string are part of that string. A line of the form

`#define` *identifier*`(` *identifier* `,` ... `,` *identifier* `)` *token-string*

where there is no space between the first identifier and the `(`, is a macro definition with arguments. Subsequent instances of the first identifier followed by a `(`, a sequence of tokens delimited by commas, and a `)` are replaced by the token string in the definition. Each occurrence of an identifier mentioned in the formal argument list of the definition is replaced by the corresponding token string from the call. The actual arguments in the call are token strings separated by commas; however commas in quoted strings or protected by parentheses do not separate arguments. The number of formal and actual arguments must be the same. Strings and character constants in the token-string are scanned for formal arguments, but strings and character constants in the rest of the program are not scanned for defined identifiers.

In both forms the replacement string is rescanned for more defined identifiers. In both forms a long definition may be continued on another line by writing \ at the end of the line to be continued. A control line of the form

`#undef` *identifier*

causes the identifier's preprocessor definition to be forgotten.

16.2 File Inclusion

A compiler control line of the form

`#include "`*filename*`"`

causes the replacement of that line by the entire contents of the file *filename*. The named file is searched for first in the directory of the original source file, and then in a sequence of specified or standard places. Alternatively, a control line of the form

```
    #include <filename>
```

searches only the specified or standard places, and not the directory of the source file. (How the places are specified is not part of the language.)

`#include`'s may be nested.

16.3 Conditional Compilation

A compiler control line of the form

```
    #if expression
```

checks whether the expression evaluates to non-zero. The expression must be a constant expression (§12). In addition to the usual C++ operations a unary operator `defined` can be used. When applied to an identifier, its value is non-zero if that identifier has been defined using `#define` and not later undefined using `#undef`; otherwise its value is `0`. A control line of the form

```
    #ifdef identifier
```

checks whether the identifier is currently defined in the preprocessor; that is, whether it has been the subject of a `#define` control line. A control line of the form

```
    #ifndef identifier
```

checks whether the identifier is currently undefined in the preprocessor.

All three forms are followed by an arbitrary number of lines, possibly containing a control line

```
    #else
```

and then by a control line

```
    #endif
```

If the checked condition is true then any lines between `#else` and `#endif` are ignored. If the checked condition is false then any lines between the test and a `#else` or, lacking a `#else`, the `#endif`, are ignored.

These constructions may be nested.

16.4 Line Control

For the benefit of other preprocessors that generate C++ programs, a line of the form

    #line *constant* "*filename*"$_{opt}$

causes the compiler to believe, for purposes of error diagnostics, that the line number of the next source line is given by the constant and the current input file is named by the identifier. If the identifier is absent the remembered file name does not change.

Appendix A: Grammar Summary

This appendix is not part of the C++ reference manual proper and does not define C++ language features.

This summary of C++ syntax is intended to be an aid to comprehension. It is not an exact statement of the language. In particular, this grammar accepts a superset of valid C++ constructs and the disambiguation rules (§6.8, §10.1.1) must be used to distinguish expressions from declarations and access control, ambiguity, and type rules must be used to weed out syntactically valid, but meaningless constructs.

A.1 Key Words

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

enum-name:
    identifier

typedef-name:
    identifier

Note that a *typedef-name* naming a class is also a *class-name* (§9.1).

A.2 Expressions

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

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

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

conditional-expression:
    logical-or-expression
    logical-or-expression ? *expression* : *conditional-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-name* `)` *cast-expression*

unary-expression:
  postfix-expression
  `++` *unary-expression*
  `--` *unary-expression*
  unary-operator *cast-expression*
  `sizeof` *unary-expression*
  `sizeof` `(` *type-name* `)`
  allocation-expression
  deallocation-expression

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

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

placement:
(*expression-list*)

restricted-type-name:
type-specifiers restricted-declarator$_{opt}$

restricted-declarator:
ptr-operator restricted-declarator$_{opt}$
restricted-declarator [*expression*$_{opt}$]

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

postfix-expression:
primary-expression
postfix-expression [*expression*]
postfix-expression (*expression-list*$_{opt}$)
simple-type-name (*expression-list*$_{opt}$)
postfix-expression . *name*
postfix-expression -> *name*
postfix-expression ++
postfix-expression --

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

primary-expression:
literal
`this`
`::` *identifier*
`::` *operator-function-name*
(*expression*)
name

name:
identifier
operator-function-name
conversion-function-name
qualified-name

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

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

A.3 Declarations

declaration:
 decl-specifiers$_{opt}$ *declarator-list*$_{opt}$;
 asm-declaration
 function-definition
 linkage-specification

decl-specifier:
 sc-specifier
 type-specifier
 fct-specifier
 template-specifier
 `friend`
 `typedef`

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

sc-specifier:
 `auto`
 `register`
 `static`
 `extern`

fct-specifier:
 `inline`
 `virtual`

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

simple-type-name:
 class-name
 typedef-name
 `char`
 `short`
 `int`
 `long`
 `signed`
 `unsigned`
 `float`
 `double`
 `void`

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

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

enum-specifier:
  `enum` *identifier*$_{opt}$ `{` *enum-list*$_{opt}$ `}`

enum-list:
  enumerator
  enum-list `,` *enumerator*

enumerator:
  identifier
  identifier `=` *constant-expression*

constant-expression:
  conditional-expression

linkage-specification:
  `extern` *string* `{` *declaration-list*$_{opt}$ `}`
  `extern` *string* *declaration*

declaration-list:
  declaration
  declaration-list `;` *declaration*

A.4 Declarators

declarator-list:
  init-declarator
  declarator-list `,` *init-declarator*

init-declarator:
  declarator *initializer*$_{opt}$

declarator:
  dname
  ptr-operator declarator
  declarator `(` *argument-declaration-list* `)` *cv-qualifier-list*$_{opt}$
  declarator `[` *constant-expression*$_{opt}$ `]`
  `(` *declarator* `)`

ptr-operator:
  `*` *cv-qualifier-list*$_{opt}$
  `&` *cv-qualifier-list*$_{opt}$
  class-name `::` `*` *cv-qualifier-list*$_{opt}$

cv-qualifier-list:
  cv-qualifier *cv-qualifier-list*$_{opt}$

cv-qualifier:
  `const`
  `volatile`

dname:
  name
  class-name
  `~` *class-name*
  typedef-name

type-name:
  type-specifier *abstract-declarator*$_{opt}$

abstract-declarator:
  ptr-operator abstract-declarator$_{opt}$
  abstract-declarator$_{opt}$ (*argument-declaration-list*) *cv-qualifier-list*$_{opt}$
  abstract-declarator$_{opt}$ [*constant-expression*$_{opt}$]
  (*abstract-declarator*)

argument-declaration-list:
  arg-declaration-list$_{opt}$ ...$_{opt}$
  arg-declaration-list , ...

arg-declaration-list:
  argument-declaration
  arg-declaration-list , *argument-declaration*

argument-declaration:
  decl-specifiers declarator
  decl-specifiers declarator = *expression*
  decl-specifiers abstract-declarator$_{opt}$
  decl-specifiers abstract-declarator$_{opt}$ = *expression*

fct-definition:
  decl-specifiers$_{opt}$ *declarator ctor-initializer*$_{opt}$ *fct-body*

fct-body:
  compound-statement

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

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

A.5 Class Declarations

class-specifier:
  class-head { *member-list*$_{opt}$ }

class-head:
  class-key identifier$_{opt}$ *base-spec*$_{opt}$
  class-key class-name base-spec$_{opt}$

member-list:
  member-declaration member-list$_{opt}$
  access-specifier : *member-list*$_{opt}$

member-declaration:
  decl-specifiers$_{opt}$ *member-declarator-list*$_{opt}$;
  function-definition ;$_{opt}$
  qualified-name ;

member-declarator-list:
  member-declarator
  member-declarator-list member-declarator

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

pure-specifier:
  `= 0`

base-spec:
  : *base-list*

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

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

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

conversion-function-name:
  `operator` *conversion-type-name*

conversion-type-name:
  type-specifiers ptr-operator$_{opt}$

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

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

mem-initializer:
  class-name (*argument-list*$_{opt}$)
  identifier (*argument-list*$_{opt}$)

operator-function-name:
  `operator` *operator*

operator: one of

```
new  delete
+    -    *    /    %    ^    &    |    ~
!    =    <    >    +=   -=   *=   /=   %=
^=   &=   |=   <<   >>   >>=  <<=  ==   !=
<=   >=   &&   ||   ++   --   ,    ->*  ->
()   []
```

A.6 Statements

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

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

dstatement:
  statement
  declaration-statement

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

compound-statement:
  { *statement-list*$_{opt}$ }

statement-list:
  dstatement
  statement-list dstatement

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

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

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

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

declaration-statement:
  declaration

A.7 Preprocessor

```
#define identifier token-string
#define identifier( identifier , ... , identifier ) token-string
#else
#endif
#if expression
#ifdef identifier
#ifndef identifier
#include "filename"
#include <filename>
#line constant "filename"opt
#undef identifier
```

Appendix B: Compatibility

This appendix is not part of the C++ reference manual proper and does not define C++ language features.

C++ is based on C (K&R78) and adopts most of the changes specified by the draft ANSI C report. 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 two exceptions, programs that are both C++ and ANSI C have the same meaning in both cases:

In C , `sizeof('a')` equals `sizeof(int)`; in C++, it equals `sizeof(char)`. In C, given

```
enum e { A };
```

`sizeof(A)` equals `sizeof(int)`; in C++, it equals `sizeof(e)`, which need not equal `sizeof(int)`.

A structure 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++ */
}
```

B.1 Extensions

This section summarizes the major extensions to C provided by C++.

B.1.1 C++ Features Available in 1985

This subsection summarizes the extensions to C provided by C++ in the 1985 version of this manual:

The types of function arguments 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.

Single-precision floating point arithmetic may be used for `float` expressions; §3.6.1 and §4.3. This is also in ANSI C.

Function names can be overloaded; §13.

Operators can be overloaded; §13.4.

Functions can be inline substituted; §7.1.1.

Data objects can be `const`; §7.1.6. This is also in ANSI C.

Objects of reference type can be declared; §8.2.2 and §8.4.3.

A free store is provided by the `new` and `delete` operators; §5.3.3, §5.3.4.

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).

The name of a class or enumeration is a type name; §9.

Any pointer can be assigned to a `void*` without use of a cast; §4.6. This is also in ANSI C.

A declaration within a block is a statement; §6.7.

Anonymous unions can be declared; §9.5.

B.1.2 C++ Features Added Since 1985

This subsection summarizes the major extensions of C++ since the 1985 version of this manual:

A class can have more than one direct base class (multiple inheritance); §10.1.

Class members can be `protected`; §11 .

Pointers to class members can be declared and used; §8.2.3, §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; §B.3.3.

Objects can be explicitly destroyed; §12.4.

Assignment and initialization are defined as member-wise assignment and initialization; §12.8.

The `overload` keyword was made redundant and moved to the anachronism section; §B.3.

General expressions are allowed as initializers for static objects; §8.4.

Data objects can be `volatile`; §7.1.6. Also in ANSI C.

Initializers are allowed for `static` class members; §9.4.

Member functions can be `static`; §9.4.

Member functions can be `const` and `volatile`; §9.3.1.

Linkage to non-C++ program fragments can be explicitly declared; §7.4.

Operators `->`, `->*`, and `,` can be overloaded; §13.4.

Classes can be abstract; §10.3.

Templates (experimental); §14.

Exception handling (experimental); §15.

B.2 C++ and ANSI C

In general, C++ provides more language features and fewer restrictions than ANSI C so most constructs in ANSI C are legal in C++ with their meanings unchanged. The exceptions are:

ANSI C programs using any of the C++ keywords

```
asm      catch    friend   new      private    public    this
class    delete   inline   operator protected  template  virtual
```

as identifiers are not C++ programs; §2.4.

Though deemed obsolescent in ANSI C, a C implementation may impose Draconian limits on the length of identifiers; a C++ implementation is not permitted to; §2.3.

In C++, a function must be declared before it can be called; §5.2.2.

The function declaration `f();` means that `f` takes no arguments (§8.2.5); in C it means that `f` can take any number of arguments of any type at all. Such use is deemed obsolescent in ANSI C.

In ANSI C a global data object may be declared several times without using the `extern` specifier; in C++ it must be defined exactly once; §3.3.

In C++, a class may not have the same name as a typedef declared to refer to a different type in the same scope; §9.1.

In ANSI C a `void*` may be used as the right-hand operand of an assignment to or initialization of a variable of any pointer type; in C++ it may not; §7.1.6.

C allows jumps to bypass an initialization; C++ does not.

In ANSI C, a global `const` by default has external linkage; in C++ it does not; §3.3.

"Old style" C function definitions and calls of undeclared functions are considered anachronisms in C++ and may not be supported by all implementations; §B.3.1. This is deemed obsolescent in ANSI C.

Enumerators declared within a struct are in the scope of the struct in C++ (§7.2) and in the scope enclosing the struct in ANSI C.

Assignment to an object of enumeration type with a value that is not of that enumeration type is considered an anachronism in C++ and may not be supported by all implementations; §7.2. ANSI C recommends a warning for such assignments.

Surplus characters are not allowed in strings used to initialize character arrays; §8.4.2.

The type of a character constant is `char` in C++ (§2.5.2) and `int` in C.

The type of an enumerator is the type of its enumeration in C++ (§7.2) and `int` in C.

In addition, the ANSI C standard allows conforming implementations to differ considerably; this may lead to further incompatibilities between C and C++ implementations. In particular, some C implementations may consider certain incompatible declarations legal. C++ requires consistency even across compilation boundaries; §3.3.

B.2.1 How to Cope

In general, a C++ program uses many features not provided by ANSI C. In this case, the minor differences of §B.2 don't matter since they are dwarfed by the C++ extensions. Where ANSI C and C++ need to share header files, more care must be taken so that such headers are written in the common subset of the two languages:

No advantage must be taken of C++ specific features such as classes, overloading, etc.

A name should not be used both as a structure tag and as the name of a different type.

Functions taking no arguments should be declared `f(void)` and not simply `f()`.

Global `const`s must be declared explicitly `static` or `extern`.

Conditional compilation using the C++ predefined name `__cplusplus` may be used to distinguish information to be used by an ANSI C program from information to be used by a C++ program.

Functions that are to be callable from both languages must be explicitly declared to have C linkage.

B.3 Anachronisms

The extensions presented here may be provided by an implementation to make it easier to use 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.

The word `overload` may be used as a *decl-specifier*(§7) in a function declaration or a function definition. In this case `overload` is a reserved word and cannot be used as an identifier.

The definition of a static data member of a class for which the default "all zeros initialization" (§9.4) applies may be omitted.

An old-style (i.e., pre-ANSI C) C preprocessor may be used.

An `int` may be assigned to an object of enumeration type.

The number of elements in an array may be left unspecified when deleting an array of a type for which there is no destructor; §5.3.4.

B.3.1 Old Style Function Definitions

The C function definition syntax

old-function-definition:
    decl-specifiers$_{opt}$ *old-function-declarator declaration-list*$_{opt}$ *fct-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 argument type will be taken to be `(...)`, that is, unchecked. If it has been declared its type must agree with that of the declaration.

Class member functions may not be defined with this syntax.

B.3.2 Old Style Base Class Initializer

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 base {
        // ...
        base (int);
};

class derived : public base {
        // ...
        derived(int i) : (i) { /* ... */ }
};
```

causes the `base` constructor to be called with the argument `i`.

B.3.3 Assignment to `this`

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 a zero value to this, a destructor can avoid the standard de-allocation operation for objects of its class. Assigning a zero value to this in a destructor also suppressed the implicit calls of destructors for bases and members. For example:

```
class cl {
      int v[10];
      cl()  { this = my_allocator( sizeof(cl) ); }
      ~cl() { my_deallocator( this ); this = 0; }
}
```

On entry into a constructor, this is non-zero if allocation has already taken place (as is the case for auto, static, and member objects) and zero otherwise.

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).

Note that if this anachronism exists the type of the this pointer either cannot be a *const or the enforcement of the rules for assignment to a constant pointer must be subverted for the this pointer.

B.3.4 Cast of Bound Pointer

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.

Appendix C: Implementation Specific Behavior

This appendix describes implementation specific behavior of the AT&T C++ Language System. Implementation specific behaviors can be categorized as follows:

1. behavior that the *Product Reference Manual* defines as "implementation dependent"
2. behavior that depends on the underlying C compiler or preprocessor used with Release 2.0
3. properties that are defined in the standard header files `stddef.h`, `limits.h`, and `stdlib.h`
4. translation limits
5. language constructs that are not implemented in this release

This appendix addresses categories 1, 2, 4, and 5. For details about properties defined in the standard header files (category 3), see the headers themselves. Additional information about constructs that are not implemented is provided in Appendix D, which contains an alphabetical listing of the "not implemented" messages.

The ordering and numbering of sections in this appendix corresponds to the order and numbering of the related sections in the *Reference Manual.* The section entitled "Translation Limits", which does not have a corresponding section in the *Reference Manual,* precedes the numbered sections.

Translation Limits

Release 2.0 of the AT&T C++ Language System imposes the following translation limits:

- 50 nesting levels of compound statements
- 127 nesting levels of `#included` files
- 10 nesting levels of linkage declarations
- 4088 characters in a token
- 22222 virtual functions in a class
- 15000 identifiers generated by the implementation

These limits can be changed by recompiling the translator. Additional translation limits may be inherited from the underlying C compiler and preprocessor.

2.3 Identifiers (Names)

Identifiers reserved by Release 2.0: In addition to identifiers that contain a sequence of two underscores, Release 2.0 also reserves the identifiers reserved in the proposed ANSI C standard.

2.5.2 Character Constants

Value of multicharacter constants: The *Reference Manual* states that the value of a multicharacter constant, such as `'abcd'`, is implementation dependent. Release 2.0 passes these constants to the underlying C compiler, which determines their values. A multicharacter constant containing more characters than `sizeof(int)` is reported as an error by Release 2.0.

Value of (single) character constants: The *Reference Manual* states that the value of a character constant is implementation dependent if it exceeds that of the largest `char`. Release 2.0 accepts octal and hexadecimal character literals that do not fit in a `char`. It uses the low order bits that make up the value of the constant. Thus, the octal character constant `'\777'`, for example, will be treated as `'\377'`. The hexadecimal character constant `'\x123'` will be treated as `'\x23'`.

Wide character constants: Release 2.0 does not implement wide character constants, such as `L'ab'`. A "not implemented" message is reported.

2.5.3 Floating Constants

Long double floating constants: When compiling with the `+a0` option, Release 2.0 will remove an `l` or `L` suffix from a floating constant before passing the constant to the underlying C compiler. Under the `+a1` option such a constant will be passed unchanged to the underlying C compiler. In either case, the constant will be considered to be of type `long double` for purposes of resolving overloaded function calls.

2.5.4 String Literals

Wide character strings: Release 2.0 does not implement wide character strings, such as `L"abcd"`. A "not implemented" message is reported.

3.4 Start and Termination

Type of `main()`: The *Reference Manual* states that the type of `main()` is implementation dependent. Release 2.0 itself does not impose any restrictions on the type of `main()`, but the underlying C compiler or the target environment may impose such restrictions.

3.6.1 Fundamental types

Signed integral types: Release 2.0 does not implement the type specifier `signed`; it issues a warning and proceeds as though the specifier `signed` had not appeared.

Long double type: When Release 2.0 is invoked with the `+a0` option, the type `long double` will be the same size and precision as type `double` in the underlying C compiler. Under the `+a1` option, `long double` will passed to the underlying C compiler as `long double`. In either case, type `long double` will be considered a distinct type for purposes of resolving overloaded function declarations and invocations.

5. Expressions

Overflow and divide check: The *Reference Manual* states that the handling of overflow and divide check in expression evaluation is implementation dependent. Release 2.0 detects division by zero when the value of the denominator is known at compile time and issues a warning, but overflow and other divide check conditions are handled by the underlying C compiler and execution environment.

5.4 Explicit Type Conversion

Explicit conversions between pointer and integral types: The *Reference Manual* states that the value obtained by explicitly converting a pointer to an integral type large enough to hold it is implementation dependent. This behavior is defined by the underlying C compiler. Similarly, the behavior when explicitly converting an integer to a pointer depends on the underlying C compiler.

5.6 Multiplicative Operators

Sign of the remainder: The *Reference Manual* states that the sign of the result of the `%` operator is non-negative if both operands are non-negative; otherwise, the sign of the result is implementation dependent. This behavior depends on the underlying C compiler except when the values of both operands are known at compile time. In this case, the sign of the result is the same as the sign of the numerator.

5.8 Shift Operators

Result of right shift: The *Reference Manual* states that the result of a right shift when the left operand is a signed type with a negative value is implementation dependent. This behavior depends on the underlying C compiler.

5.9 Relational Operators

Pointer comparisons: According to the *Reference Manual*, certain pointer comparisons are implementation dependent. For Release 2.0, the results of these comparisons depend on the underlying C compiler.

7.1.1 Storage Class Specifiers

Inline functions: The *Reference Manual* states that the `inline` specifier is a hint to the compiler. Release 2.0 performs inline substitution of `inline` functions wherever possible. Inline calls are not, however, generated in the following cases:

- For an inline function with a non-`void` return type, the presence of an iteration, `switch`, labeled, or `goto` statement within the inline function precludes generating code inline. In these cases, a warning message is issued and code to call the function out-of-line is generated.
- For an inline function with `void` return type, the presence of a `return` statement precludes generating code inline. If a call to such a function is seen, a "not implemented" message is issued.
- For an inline function containing a static object, a "not implemented" message is issued.
- For an inline function containing an automatic array object, a "not implemented" message is issued if the function is called.
- For an inline function containing a recursive call, out-of-line code is generated.
- In general, if a call to an inline function precedes (lexically) the definition of the inline function, an out-of-line call is generated.

These restrictions may be relaxed in future releases of the AT&T C++ Language System.

7.1.6 Type Specifiers

`Volatile`: Release 2.0 does not implement the type specifier `volatile`; it is ignored and a warning message is issued.

`Signed`: Release 2.0 does not implement the type specifier `signed`; it is ignored and a warning message is issued.

7.3 Asm Declarations

Effect of an asm declaration: Release 2.0 passes `asm` declarations to the underlying C compiler without modification.

7.4 Linkage Specifications

Languages supported: Release 2.0 supports linkage to C and C++.

Linkage to functions: The effect of a `"C"` linkage specification (`extern "C"`) on a function that is not a member function is that the function name is not encoded with type information, as is otherwise done for C++ functions. Member functions are not affected by linkage specifications.

Linkage to non-functions: The `"C"` linkage specification, (`extern "C"`) when applied to a non-function declaration, does not affect the C code generated.

9.3 Member Functions

Calling a non-static member function: The *Reference Manual* states that the effect of calling a non-static member function of a class `X` for something that is not an object of class `X` is implementation dependent. In such cases Release 2.0 generates code to call the member function and assigns to `this` the address of the object for which the function is called. This behavior is not guaranteed to be supported in future releases of the AT&T C++ Language System.

9.6 Bit Fields

Allocation and alignment of bit fields: The *Reference Manual* states that the allocation and alignment of bit fields within a class object is implementation dependent. Responsibility for the allocation and alignment of bit fields rests with the underlying C compiler.

13.2 Argument Matching

Integral arguments: The type of the result of an integral promotion (§4.1) depends on the execution environment, as does the type of an unsuffixed integer constant (§2.5.1). Consequently, the determination of which overloaded function to call may also depend on the execution environment, as illustrated by the example in §13.2 of the *Reference Manual.*

14. Templates (experimental)

Release 2.0 does not implement templates. The keyword `template` is reserved for future use. A "not implemented" message is reported if `template` is seen.

15. Exception Handling (experimental)

Release 2.0 does not implement exception handling. The keyword `catch` is reserved for future use. A "not implemented" message is reported if `catch` is seen.

16. Compiler Control Lines

Behavior of the preprocessor: Lines beginning with `#` communicate with the preprocessor. The effect of these lines depends on the underlying preprocessor.

16.1 Token Replacement

Predefined macro names: The following macro names are defined by Release 2.0:

`__cplusplus`	The decimal constant 1.
`c_plusplus`	The decimal constant 1. This macro name is provided for compatibility with previous releases and is not guaranteed to be supported in future releases.

B.3 Anachronisms

The anachronisms supported in Release 2.0 are not guaranteed to be supported in future releases of the AT&T C++ Language System.

The word `overload` is reserved. It may be used as a *decl-specifier* in a function declaration or a function definition; it cannot be used as an identifier.

The definition of a static data member of a class may be omitted. When a definition is not given for a static data member, its initial value will be zero.

The AT&T C++ Language System Release 2.0 does not include a preprocessor. Nothing in Release 2.0 precludes the use of an old-style (that is, pre-ANSI C) C preprocessor.

An `int` may be assigned to an object of enumeration type.

The number of elements in an array may be left unspecified when deleting an array of a nonclass type.

B.3.1 Old Style Function Definitions

Programs may use the old C function definition syntax.

B.3.2 Old Style Base Class Initializer

The name of the base class may be omitted in a member initializer list when there is exactly one immediate base class.

B.3.3 Assignment to `this`

Release 2.0 allows assignment to `this` pointers in constructors and destructors to control memory management of class objects.

B.3.4 Cast of Bound Pointer

Release 2.0 allows casting a pointer to member function for a particular object into a pointer to function. Any use of the resulting pointer is undefined.

Ap endix D: "[illegible]mented" Messages

This appendix contains [illegible] nation for all "not implemented" messages produced by the AT& C++ Language S[illegible]. They are listed here in alphabetical order.

Each message is preced[illegible] nd line number. The line number is usually the line on whi a problem has be[illegible]

"No mplemented" me[illegible] command to fail; that [illegible] code is not generated, nor is the prog m linked. Release [illegible] attempt to examine the rest of your program for other err[illegible]

& of *op*

lvalue *op* too com[illegible]

An operand requir[illegible] or the operand of the unary & operator is an [illegible] that has side effect[illegible] operand of & might be, for example an assign[illegible] sion, an increment [illegible] or a decrement operation.

```
int i, j;
int *p1 = &(++(i =j));
```

```
"file", line 2: not implemented: ++ of %=
"file", line 2: not implemented: & of ++
```

++ of *op*

The operand of the [illegible] operator contains a ?: operator, a comma operator, or an operator that produces side effects [illegible]h as =,or ++.

```
int i, j;
int k = ++(i+ j);
```

```
"file", line 2: n[illegible] plemented: ++ of +=
```

1st operand of [illegible]o complicated

The first operand of [illegible]tion call expression involves a pointer to a member function and is an expression that may [illegible] side effects or may require a temporary.

```
struct S { v    al int f(); };
int (S::*pmf    = &S::f;
S *f();
int i = (f()    f)();
```

```
"file", line 5: not implemented: 1st operand of .* too complicated
```

- `2nd operand of .* too complicated`

  The second operand of a pointer to member operator is an expression that has side effects.

```
struct S { int f(); };
int (S::*pmf)() = &S::f;
S *sp = new S;
int i = 5;
int j = (sp->*(i+=5, pmf))();
```

```
"file", line 5: not implemented: 2nd operand of .* too complicated
```

- `anonymous unions nested deeper than 2 levels`

  Anonymous unions are nested to more than two levels.

```
static union {
      union {
            int i;
            union {
                  char c;
                  int j;
            };
      };
};
```

```
"file", line 7: not implemented: anonymous unions nested deeper than 2 levels
```

- `cannot expand inline function` *function* `with for statement`

  A `for` statement appears in the declaration of an inline function.

```
struct S {
      int s[100];
      S() { for (int i = 0; i < 100; i++) s[i] = i; }
};
```

```
"file", line 1: not implemented: cannot expand inline function S::S() with for statement
```

- `cannot expand inline function` *function* `with return statement`

  A `return` statement without an expression appears in the declaration of an inline function.

```
inline void f() {
      return;
}
```

```
"file", line 2: not implemented: cannot expand inline function f() with return statement
```

- `cannot expand inline function` *function* `with statement after "return"`

  A value-returning inline function contains a statement following a `return` statement.

```
inline int f(int i) {
      if (i) return i;
      return 0;
}
```

```
"file", line 4: not implemented: cannot expand inline function f() with statement
after "return"
```

- `cannot expand inline function` *function* `with two local variables with the same name (`*name*`)`

  Two variables with the same name and different types are declared within the body of a value-returning inline function.

```
inline int f(int i) {
      { int x = i; }
      { double x = i; }
      return 0;
}
```

```
"file", line 5: not implemented: cannot expand inline function f() with two local
variables with the same name (x)
```

- `cannot expand inline function needing temporary variable in non function context`

  This message should not be produced.

- `cannot expand inline function needing temporary variable of vector type`

  An inline function that contains a local declaration of an array object is called.

```
inline int f(int i) {
      int a[1];
      a[0] = i;
      return i;
}
int v = f(0);
```

```
"file", line 6: not implemented: cannot expand inline function needing
temporary variable of vector type
```

- `cannot expand inline function with return in if statement`

  This condition occurs as a result of implementation-generated statements. It can be circumvented by avoiding unnecessary `return` statements in inline functions.

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

```
"file", line 1: not implemented: cannot expand inline function S::S() with statement
after "return"
"file", line 1: not implemented: cannot expand inline function with return in if statement
```

- `cannot expand inline function with static` *name*

  An inline function contains the declaration of a static object.

```
inline void f() {
      static int i = 5;
};
```

```
"file", line 2: not implemented: cannot expand inline function with static i
```

- `cannot expand inline void` *function* `called in comma expression`
- `cannot expand inline void` *function* `called in for expression`

  A call of an `inline void` function that cannot be translated to an expression (that is, one that includes a loop, a `goto`, or a `switch` statement) appears as the first operand of a comma operator or appears in the second expression of a `for` statement.

```
int i;
void inline f1() { for (;;) ; }
void inline f2() { i += 5; }
void g() { for ( f1(), i=i;;) ; }
void h() { for ( f2(), i=i;;) ; }
void j() { for (;; f1()); }
void k() { for (;; f2()); }
```

```
"file", line 4: not implemented: cannot expand inline void f1() called in comma expression
"file", line 6: not implemented: cannot expand inline void f1() called in for expression
```

- `cannot expand value-returning inline` *function* `with call of ...`

  A value-returning inline function is defined, and it contains a call to another inline function that is not value-returning.

```
inline void f() { for(;;); }
inline int g() { f(); return 0; }
```

```
"file", line 2: not implemented: cannot expand value-returning inline g() with call of
non-value-returning inline f()
```

- `cannot merge lists of conversion functions`

  A derived class with multiple bases is declared and there are conversion operators declared in more than one of the base classes.

```
struct B1 {
      operator int();
};
struct B2 {
      operator float();
};
struct D : public B1, public B2 { };
```

```
"file", line 7: not implemented: cannot merge lists of conversion functions
```

- `class defined within sizeof`

  A class definition appears as the type name in a `sizeof` expression.

```
int i = sizeof (struct S { int i; });
```

```
"file", line 1: not implemented: class defined within sizeof
```

- `class hierarchy too complicated`

  This message should not be produced.

- `conditional expression with` *type*

  The second and third operands of a conditional expression are member functions or pointers to member functions.

```
struct S {
      int t;
      f() { return t; }
      g() { return 1; }
      h() { return (t?f:g)(); }
};
```

```
"file", line 5: not implemented: conditional expression with int S::()
"file", line 5: error:  object or pointer missing for ? of type  int S::()
```

- `constructor as default argument`
- `constructor needed for argument initializer`

  The default value for an argument is a constructor or is an expression that invokes a constructor.

```
struct S { S(int); };
int f(S = S(1));
int g(S = 5);
```

```
"file", line 2: not implemented: constructor as default argument
"file", line 3: not implemented: constructor needed for argument initializer
```

- `copy of` *member*`[], no memberwise copy for` *class*

  An implementation-generated copy operation for a class `X` is required, but the operation cannot be generated because `X` has an array member of one of the following kinds of types: a class with a virtual base class, a class with a programmer-defined `X(X&)`, or a class with a programmer-defined `operator=()`.

```
struct S1 {};
struct S2 : virtual S1 {};
struct X { S2 m[1]; };
X var1;
X var2 = var1;
```

```
"file", line 5: not implemented: copy of S2[], no memberwise copy for S2
```

- `default argument too complicated`

  A default argument in a declaration not at file scope requires the generation of a temporary.

```
struct S { int f(int &r = (int) 'x'); };
```

```
"file", line 1: not implemented: default argument too complicated
"file", line 1: not implemented: needs temporary variable to evaluate argument initializer
```

- `default arguments for constructor for array of class` *name*

  A constructor with default arguments is invoked to initialize an array of class objects without an explicit initializer.

```
struct S { S(int=5); };
S s[2];
S *sp = new S[2];
```

```
"file", line 2: not implemented: default arguments for constructor for array of class S
"file", line 3: not implemented: default arguments for constructor for array of class S
```

- `delete array through pointer or array of arrays`

  The object deleted by a `delete` operator has the type "pointer to array of class." This can be either a single- or multi-dimensional array. This error does not occur if the type is "pointer to class."

```
struct S {};
typedef S A1[1];
void f() {
      A1 *p = (A1 *) new A1;
      delete p;
}
```

```
"file", line 6: not implemented: delete array through pointer or array of arrays
```

- `expression in for statement needs temporary of class with destructor`

  An expression in a `for` statement requires a temporary of a class type for which there is a destructor.

```
struct S { S(int); ~S(); } s(0);
main() {
      for( int i = 10 ; i ; s = --i );
}
```

```
"file", line 3: not implemented: expression in for statement needs temporary of class S
with destructor
```

- `general initializer in initializer list`

  The initializer list in a declaration contains an expression that is not constant.

```
int f();
int i[1] = { f() };
```

```
"file", line 2: not implemented: general initializer in initializer list
```

- `catch`

  The keyword `catch` appears; `catch` is reserved for future use.

```
int catch;
```

```
"file", line 1: not implemented: catch
"file", line 1: warning: name expected in declaration list
```

- `initialization of automatic aggregate`

  An aggregate at local scope is initialized. This message is not issued if the `+a1` option (produces declarations acceptable to an ANSI C compiler) is specified.

```
void f() {
        int i[1] = {1};
}
```

```
"file", line 2: not implemented: initialization of automatic aggregate
```

- `initialization of union with initializer list`

  An object of union type is initialized with an initializer list. This message is not issued if the `+a1` option (produces declarations acceptable to an ANSI C compiler) is specified.

```
union U { int i; float f; };
U u = { 1 };
```

```
"file", line 2: not implemented: initialization of union with initializer list
```

- `initializer for local static too complicated`

  This message should not be produced.

- `initializer for multi-dimensional array of objects of class` *class* `with constructor` *name*

  A multi-dimensional array of a class with a constructor has an explicit initializer.

```
struct S { S(int); };
S s[2][2] = {1,2,3,4};
```

```
"file", line 2: not implemented: initializer for multi-dimensional
array of objects of class S with constructor ::s
```

- `label in block with destructors`

  A labeled statement appears in a block in which an object with a destructor exists.

```
struct S { S(int); ~S(); };
void f() {
      S s(5);
xyz:  ;
}
```

```
"file", line 5: not implemented: label in block with destructors
```

- `lvalue` *op* `too complicated`

  See "`&` `of` *op*."

- `needs temporary variable to evaluate argument initializer`

  A default argument requires a temporary variable.

```
void g() {
      int h(int & = 5);
}
```

```
"file", line 2: not implemented: needs temporary variable to evaluate argument initializer
```

- `non-trivial declaration in switch statement`

  A "non-trivial" declaration appears within a switch statement. Such a declaration might declare an object of reference type, a static object, a `const` object, an object of a class type with constructor or destructor, an object with an initializer list, or an object initialized with a string literal.

```
void f(int i) {
      switch (i) {
      default:
            int& j = i;
      }
}
```

```
"file", line 2: not implemented: non-trivial declaration in switch statement
   (try enclosing it in a block)
```

Note that since it is illegal to jump past a declaration with an explicit or implicit initializer unless the declaration is in an inner block that is not entered, most declarations in `switch` statements and not contained in inner blocks will be errors.

- `pointer expression too complicated for delete`
- `pointer expression too complicated for delete[]`

The `delete` operator is applied to an object or an array of objects of a type with a virtual destructor, and the operand is not a simple expression.

```
struct S {
      S();
      virtual ~S();
};
S *f();
void g() {
      delete f();
      delete [2] f();
}
```

```
"file", line 9: not implemented: pointer expression too complicated for delete
"file", line 9: not implemented: pointer expression too complicated for delete[]
```

A variable can be introduced to circumvent the unimplemented functionality. For example,

```
void g2() {
      S* sp;
      sp = f();
      delete sp;
}
```

- `pointer to member function` *type* `too complicated`

This message should not be produced.

- `pointer to member of not first base`

  A pointer to member function is initialized with or assigned the address of a member of a base class other than its lexically first base.

```
struct B1 {};
struct B2 { int f(); };
struct D : B1, B2 {};
int (D::*pmf)() = D::f;
```

```
"file", line 4: not implemented: int B2::() assigned to int (D::*)() (too complicated)
"file", line 4: not implemented: pointer to member of not first base
```

- `public specification of overloaded` *function*

  The base class member in an access declaration refers to an overloaded function. A similar message is issued for illegal `private` and `protected` access declarations.

```
struct B { int f(); int f(int); };
class D : private B {
public:
       B::f;
};
```

```
"file", line 2: not implemented: public specification of overloaded B::f()
```

- `struct` *name* `member` *name*

  This message should not be produced.

- `template`

  The keyword `template` appears; `template` is reserved for future use.

```
int template;
```

```
"file", line 1: not implemented: template
"file", line 1: warning: name expected in declaration list
```

- `temporary of class` *name* `with destructor needed in` *expr* `expression`

  An expression containing a `?:`, `||`, or `&&` operator requires a temporary object of a class that has a destructor.

```
struct S { S(int); ~S(); };
S f(int i) {
        return i ? S(1) : S(2) ;
}
```

```
"file", line 3: not implemented: temporary of class S with destructor needed in ?: expression
```

- `too few initializers for` *name*

  The initializer list for an array of class objects has fewer initializers than the number of elements in the array.

```
struct S { S(int); };
S a[2] = {1};
```

```
"file", line 2: not implemented: too few initializers for ::a
```

- *type1* `assigned to` *type2* `(too complicated)`

  A pointer is initialized or assigned with an expression whose type is too complicated.

```
struct S1 {};
struct S2 { int i; };
struct S3 : S1, S2 {};
int S3::*pmi = &S2::i;
```

```
"file", line 4: not implemented: int S2::* assigned to int S3::* (too complicated)
```

- `visibility declaration for conversion operator`

  An access declaration is specified for a conversion operator.

```
struct B { operator int(); };
class D : private B {
public:
        B::operator int;
};
```

```
"file", line 1: not implemented: visibility declaration for conversion operator
```

- `wide character constant`
- `wide character string`

A wide character constant or a wide character string is used.

```
int wc = L'ab';
char *ws = L"abcd";
```

```
"file", line 1: not implemented: wide character constant
"file", line 2: not implemented: wide character string
```

I Index

Index