
Bjarne Stroustrup

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

C++ is C extended with classes, inline functions, operator overloading, function name overloading, constant types, references, free store management, function argument checking, and a new function definition syntax. This manual was derived from the Unix System V C reference manual, and the general organization and section numbering have been preserved wherever possible. The differences between C++ and C are summarized. Except for details like introduction of new keywords, C++ is a superset of C. An index and a table of contents are also provided.

C++ has been implemented and has been used for non-trivial projects. For a more readable presentation of most of the new features see

Bjarne Stroustrup: “A C++ Tutorial”, AT&T Bell Laboratories CSTR-113, or
1. INTRODUCTION

This manual describes the C++ programming language. C++ is C as described in the C book+ extended with classes, inline functions, operator overloading, function name overloading, constant types, references, free store management, function argument checking, and a new function definition syntax. The differences between C++ and C are summarized in §19. This manual describes the language as of October 1984.

2. LEXICAL CONVENTIONS

There are six classes of tokens: identifiers, keywords, constants, strings, operators, and other separators. Blanks, tabs, new-lines, 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 include the longest string of characters which could possibly constitute a token.

2.1 Comments

The characters /* introduce a comment, which terminates with the characters */. Comments do not nest.

2.2 Identifiers (Names)

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.

2.3 Keywords

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

<table>
<thead>
<tr>
<th>asm</th>
<th>auto</th>
<th>break</th>
<th>case</th>
<th>char</th>
<th>class</th>
<th>const</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>delete</td>
<td>do</td>
<td>double</td>
<td>else</td>
<td>enum</td>
<td>extern</td>
</tr>
<tr>
<td>float</td>
<td>for</td>
<td>friend</td>
<td>goto</td>
<td>if</td>
<td>inline</td>
<td>int</td>
</tr>
<tr>
<td>long</td>
<td>new</td>
<td>operator</td>
<td>overload</td>
<td>public</td>
<td>register</td>
<td>return</td>
</tr>
<tr>
<td>short</td>
<td>sizeof</td>
<td>static</td>
<td>struct</td>
<td>switch</td>
<td>this</td>
<td>typedef</td>
</tr>
<tr>
<td>union</td>
<td>unsigned</td>
<td>virtual</td>
<td>void</td>
<td>while</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2.4 Constants

There are several kinds of constants, as listed below. Hardware characteristics that affect sizes are summarized in §2.6.

2.4.1 Integer constants

An integer constant consisting of a sequence of digits is taken to be octal if it begins with 0 (digit zero), decimal otherwise. The digits 8 and 9 are not octal digits. A sequence of digits preceded by 0x or 0X (digit zero) is taken to be a hexadecimal integer. The hexadecimal digits include a or A through f or F with values 10 through 15. A decimal constant whose value exceeds the largest

signed integer is taken to be long; an octal or hex constant which exceeds the largest unsigned
integer is likewise taken to be long; otherwise integer constants are taken to be int.

2.4.2 Explicit long constants

A decimal, octal, or hexadecimal integer constant immediately followed by 1 (letter ell) or L is a
long constant.

2.4.3 Character constants

A character constant is a character enclosed in single quotes, as in 'x'. The value of a character
constant is the numerical value of the character in the machine's character set. Character constants
are taken to be int.

Certain non-graphic characters, the single quote ', and the backslash \, may be represented
according to the following table of escape sequences:

<table>
<thead>
<tr>
<th>Escape</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>\n</td>
<td>NL (LF)</td>
</tr>
<tr>
<td>\t</td>
<td>HT</td>
</tr>
<tr>
<td>\v</td>
<td>VT</td>
</tr>
<tr>
<td>\b</td>
<td>BS</td>
</tr>
<tr>
<td>\r</td>
<td>CR</td>
</tr>
<tr>
<td>\f</td>
<td>FF</td>
</tr>
<tr>
<td>\</td>
<td>backslash</td>
</tr>
<tr>
<td>'</td>
<td>single quote</td>
</tr>
<tr>
<td>\ddd</td>
<td>bit pattern</td>
</tr>
</tbody>
</table>

The escape \ddd consists of the backslash followed by 1, 2, or 3 octal digits which are taken to
specify the value of the desired character. A special case of this construction is \0 (not followed
by a digit), which indicates the character NUL. If the character following a backslash is not one of
those specified, the backslash is ignored.

2.4.4 Floating constants

A floating constant consists of an integer part, a decimal point, a fraction part, an e or E, and an
optionally signed integer exponent. The integer and fraction parts both consist of a sequence of
digits. Either the integer part or the fraction part (not both) may be missing; either the decimal
point or the e (E) and the exponent (not both) may be missing. A floating constant which cannot
be represented exactly as a single-precision float is taken to be double-precision; see §2.6.

2.4.5 Enumeration constants

Names declared as enumerators (see §8.5) are constants of type int.

2.4.6 Declared constants

An object (§5) of any type can be specified to have a constant value throughout the scope (§4.1) of
its name. For pointers the *const declarator (§8.3) is used to achieve this; for non-pointer
objects the specifier const (§8.2) is used.

2.5 Strings

A string is a sequence of characters surrounded by double quotes, as in "....". A string has type
"array of characters" and storage class static (see §4 below), and is initialized with the given
characters. All strings, even when written identically, are distinct. The compiler places a null byte
\0 at the end of each string so that programs which scan the string can find its end. In a string,
the double quote character " must be preceded by a \; in addition, the same escapes as described
for character constants may be used. Finally, a new-line may occur only immediately following a
\; then both the \ and the new-line are ignored.
2.6 Hardware characteristics

The following table summarizes certain hardware properties that vary from machine to machine.

<table>
<thead>
<tr>
<th></th>
<th>DEC VAX ASCII</th>
<th>Motorola 68000 ASCII</th>
<th>IBM 370 EBCDIC</th>
<th>AT&amp;T 3B ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>8 bits</td>
<td>8 bits</td>
<td>8 bits</td>
<td>8 bits</td>
</tr>
<tr>
<td>int</td>
<td>32</td>
<td>16</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>long</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>float</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>double</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>pointer</td>
<td>32</td>
<td>32</td>
<td>24</td>
<td>32</td>
</tr>
<tr>
<td>float range</td>
<td>$10^{\pm38}$</td>
<td>$10^{\pm38}$</td>
<td>$10^{\pm76}$</td>
<td>$10^{\pm38}$</td>
</tr>
<tr>
<td>double range</td>
<td>$10^{\pm38}$</td>
<td>$10^{\pm38}$</td>
<td>$10^{\pm76}$</td>
<td>$10^{\pm308}$</td>
</tr>
<tr>
<td>field type</td>
<td>signed</td>
<td>unsigned</td>
<td>unsigned</td>
<td>unsigned</td>
</tr>
<tr>
<td>field order</td>
<td>right-to-left</td>
<td>left-to-right</td>
<td>left-to-right</td>
<td>left-to-right</td>
</tr>
<tr>
<td>char</td>
<td>signed</td>
<td>unsigned</td>
<td>unsigned</td>
<td>unsigned</td>
</tr>
</tbody>
</table>

3. 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. An optional terminal or non-terminal symbol is indicated by the subscript "opt," so that

\[
\{ \text{expression}_{opt} \}
\]

indicates an optional expression enclosed in braces. The syntax is summarized in §19.

4. WHAT'S IN A NAME?

A name denotes an object, a function, a type, or a value. A name can only be used within a region of program text called its scope. A name has a type which determines its use. An object is a region of storage. An object has a storage class which determines its lifetime. The meaning of the values found in an object is determined by the type of the name used to access it.

4.1 Scopes

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

Local: A name declared in a block is local to that block and can only be used in it after the point of declaration and in blocks enclosed by it. Exceptions are labels (§9.12) which can be used anywhere in the function in which they are declared, and function names which belong to the file or program scope. Names of formal parameters for a function are treated as if they were declared in the outermost block of that function.

File: A name declared outside any block (§9.2) or class (§8.5) can be used in the file in which it is declared after the point of declaration. It is not accessible from other files in a multi-file program unless it is explicitly declared extern.

Program: A name declared extern is common to every file in a multi-file program, so that a declaration of that name in another file refers to the same object (§5), function (§10.1), type (§8.7), or value (§8.10).

Class: The name of a class member is local to its class and can only be used either in a member function of that class, for an object of its class using the . operator (§7.1), or for a pointer to an object of its class using the -> operator (§7.1). Static class members (§8.5.1) and function members can also be referred to where the name of their class is in scope by using the :: operator (§7.1).
A name may be hidden by an explicit declaration of that same name in a block or class. A name in a block or class can only be hidden by a name declared in an enclosed block or class. A hidden non-local name can still be used when its scope is specified using the :: operator; see §7.1.

4.2 Storage classes

There are two declarable storage classes: automatic and static.

 Automatic objects are local to each invocation of a block and are discarded upon exit from it.

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

Some objects are not associated with names and their lifetimes are explicitly controlled using the new and delete operators; see §7.2, §9.14, and §17.

4.3 Fundamental types

Objects declared as characters (char) are large enough to store any member of the implementation's character set, and if a genuine character from that character set is stored in a character variable, its value is equivalent to the integer code for that character. Other quantities may be stored into character variables, but the implementation is machine-dependent.

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 host machine architecture; the other sizes are provided to meet special needs.

Each enumeration (§8.9) is a set of named constants. The properties of an enum are identical to those of an int.

Unsigned integers, declared unsigned, obey the laws of arithmetic modulo $2^n$ where $n$ is the number of bits in the representation.

Single-precision floating point (float) and double-precision floating point (double) may be synonymous in some implementations.

Because objects of the foregoing types can usefully be interpreted as numbers, they will be referred to as arithmetic types. Types char, int of all sizes, and enum will collectively be called integral types. float and double will collectively be called floating types.

The void type specifies an empty set of values; see §6.7.

4.4 Derived types

Besides the fundamental arithmetic 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;
- functions which take arguments of given types and return objects of a given type;
- pointers to objects of a given type;
- references to objects of a given type;
- constants which are values of a given type;
- classes containing a sequence of objects of various types, a set of functions for manipulating these objects, and a set of restrictions on the access to these objects and functions;
- structures which are classes without access restrictions;
- unions which are structures capable of containing objects of different types at different times.

In general these methods of constructing objects can be applied recursively.
5. OBJECTS AND 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. There are operators which 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 \( E_1 = E_2 \) in which the left operand \( E_1 \) must be an lvalue expression. The discussion of each operator below indicates whether it expects lvalue operands and whether it yields an lvalue.

6. 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 explains the result to be expected from such conversions. §6.6 summarizes the conversions demanded by most ordinary operators; it will be supplemented as required by the discussion of each operator. §8.5.6 describes user-defined conversions.

6.1 Characters and integers

A character or a short integer may be used wherever an integer may be used. In all cases the value is converted to an integer. Conversion of a shorter integer to a longer always involves sign extension; integers are signed quantities. Whether or not sign-extension occurs for characters is machine dependent; see §2.6. The more explicit type unsigned char forces the values to range from 0 to a machine dependent maximum.

On machines that treat characters as signed, the characters of the ASCII set are all positive. However, a character constant specified with an octal escape suffers sign extension and may appear negative; for example, \"\377\" has the value -1.

When a longer integer is converted to a shorter or to a char, it is truncated on the left; excess bits are simply discarded.

6.2 Float and double

Floating arithmetic is carried out as if in double-precision. Conversions between single-precision and double-precision floating-point numbers are as mathematically correct as the hardware allows.

6.3 Floating and integral

Conversions of floating values to integral type tend to be machine-dependent; in particular 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 well behaved. Some loss of precision occurs if the destination lacks sufficient bits.

6.4 Pointers and integers

An expression of integral type may be added to or subtracted from a pointer; in such a case the first is converted as specified in the discussion of the addition operator.

Two pointers to objects of the same type may be subtracted; in this case the result is converted to an int or a long dependent on the machine; see §7.4.

6.5 Unsigned

Whenever an unsigned integer and a plain integer are combined, the plain integer is converted to unsigned and the result is unsigned. The value is the least unsigned integer congruent to the signed integer (modulo wordsize). In a 2's complement representation, this conversion is conceptual and there is no actual change in the bit pattern.
When an unsigned integer is converted to long, the value of the result is the same numerically as that of the unsigned integer. Thus the conversion amounts to padding with zeros on the left.

6.6 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."

First, any operands of type char, unsigned char, or short are converted to int, and any of type float is converted to double.

Then, if either operand is double, the other is converted to double and that is the type of the result.

Otherwise, if either operand is unsigned long the other is converted to unsigned long and that is the type of the result.

Otherwise, if either operand is long, the other is converted to long and that is the type of the result.

Otherwise, if either operand is unsigned, the other is converted to unsigned and that is the type of the result.

Otherwise, both operands must be int, and that is the type of the result.

6.7 Void

The (nonexistent) value of a void object may not be used in any way, and neither explicit nor implicit conversions may be applied. Because a void expression denotes a nonexistent value, such an expression may be used only as an expression statement (§9.1) or as the left operand of a comma expression (§7.15).

An expression may be converted to type void by use of a cast. For example, this makes explicit the discarding of the value of a function call used as an expression statement.

A object of type void* (pointer to void) can be used to point to objects of unknown type.

7. EXPRESSIONS

The precedence of expression operators is the same as the order of the major subsections of this section, highest precedence first. Thus, for example, the expressions referred to as the operands of + (§7.4) are those expressions defined in §§7.1-7.4. Within each subsection, the operators have the same precedence. Left- or right-associativity is specified in each subsection for the operators discussed therein. The precedence and associativity of all the expression operators is summarized in the grammar of §18.

Otherwise the order of evaluation of expressions is undefined. In particular the compiler considers itself free to compute subexpressions in the order it believes most efficient, even if the subexpressions involve side effects. The order in which side effects take place is unspecified. Expressions involving a commutative and associative operator (*, +, &, |, ^) may be rearranged arbitrarily, even in the presence of parentheses; to force a particular order of evaluation an explicit temporary must be used.

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

In addition to the standard meanings described in §7.2-7.15 operators may be overloaded, that is given meanings when applied to user-defined types; see §7.16.
7.1 Primary expressions

Primary expressions involving ., ->, ::, subscripting, and function calls group left-to-right.

\[
id:
\begin{align*}
& \text{id} \\
& \text{identifier} \\
& \text{operator-function-name} \\
& \text{typedef-name} \ : \ : \ \text{identifier} \\
& \text{typedef-name} \ : \ : \ \text{operator-function-name}
\end{align*}
\]

\[
\text{primary-expression:}
\begin{align*}
& \text{id} \\
& \text{:: identifier} \\
& \text{constant} \\
& \text{string} \\
& \text{this} \\
& \left( \text{expression} \right) \\
& \text{primary-expression} \ [ \text{expression} ] \\
& \text{primary-expression} \ ( \text{expression-list} \ \text{opt} ) \\
& \text{primary-expression} \ . \ \text{id} \\
& \text{primary-expression} \ \Rightarrow \ \text{id}
\end{align*}
\]

\[
\text{expression-list:}
\begin{align*}
& \text{expression} \\
& \text{expression-list} \ , \ \text{expression}
\end{align*}
\]

An identifier is a primary expression, provided it has been suitably declared as discussed below. Its type is specified by its declaration. If the type of the identifier is "array of . . .", however, then the value of the identifier-expression is a pointer to the first object in the array, and the type of the expression is "pointer to . . .". Moreover, an array identifier is not an lvalue expression. Likewise, an identifier which is declared "function returning . . .", when used except in the function-name position of a call, is converted to "pointer to function returning . . .". An operator-function-name is an identifier with a special meaning; see §7.16 and §8.5.10.

The operator :: followed by an identifier is a primary expression, provided the identifier has been suitably declared in the file or program scope (§4.1). Its type is specified by the declaration of the identifier. It allows an object to be referred to by name even if its identifier has been redefined in a local scope.

A typedef-name (§8.8) followed by :: followed by an identifier is a primary expression. The typedef-name must denote a class (§8.5) and the identifier must denote a member of that class. Its type is specified by the declaration of the identifier.

A constant is a primary expression. Its type may be int, long, or double depending on its form.

A string is a primary expression. Its type is originally "array of char"; but following the same rule given above for identifiers, this is modified to "pointer to char" and the result is a pointer to the first character in the string. (There is an exception in certain initializers; see §8.6.)

The keyword this is a primary expression in the body of a member function (see §8.5). There it refers to the object for which the member function was invoked.

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 primary expression followed by an expression in square brackets is a primary expression. The intuitive meaning is that of a subscript. Usually, the primary expression has type "pointer to . . .", the subscript expression is int, and the type of the result is " . . .". The expression E1[E2] is
identical (by definition) to \(\ast((E_1)+(E_2))\). All the clues needed to understand this notation are contained in this section together with the discussions in §§ 7.1, 7.2, and 7.4 on identifiers, \(\ast\), and + respectively; §14.3 below summarizes the implications.

A function call is a primary expression followed by parentheses containing a possibly empty, comma-separated list of expressions which constitute the actual arguments to the function. The primary expression must be of type "function returning \ldots", and the result of the function call is of type "\ldots". A hitherto unseen identifier followed immediately by a left parenthesis is contextually declared to represent a function returning an integer. Its argument type will be declared to that of the argument list of the call.

The actual arguments are compared with the formal arguments and conversions are performed as if the formal argument were initialized with its actual argument (see §8.6).

In preparing for a call to a function, a copy is made of each actual parameter. A function may change the values of its formal parameters, but these changes cannot affect the values of the actual parameters. On the other hand, it is possible to pass a pointer or a reference on the understanding that the function may change the value of the object to which the pointer or reference points. An array name is a pointer expression.

A function may be declared to accept fewer arguments or more arguments than are specified in the function declaration; see §8.4. Any actual argument of type \texttt{float} for which there is no formal argument are converted to \texttt{double} before the call; any of type \texttt{char} or \texttt{short} are converted to \texttt{int}; and as usual, array names are converted to pointers. The order of evaluation of arguments is undefined by the language; take note that the various compilers differ.

Recursive calls to any function are permitted.

A primary expression followed by a dot followed by an identifier (or an identifier qualified by a typedef-name using the :: operator) is an expression. The first expression must be a class object, and the identifier must name a member of that class. The value is the named member of the object, and it is an lvalue if the first expression is an lvalue. Note that "class objects" can be structures (§8.5.11) and unions (§8.5.12).

A primary expression followed by an arrow (\(\rightarrow\)) followed by an identifier (or an identifier qualified by a typedef-name using the :: operator) is an expression. The first expression must be a pointer to a class object and the identifier must name a member of that class. The result is an lvalue referring to the named member of the class to which the pointer expression points. Thus the expression \(E_1\rightarrow\text{MOS}\) is the same as \((\ast E_1).\text{MOS}\). Classes are discussed in §8.5.

If a primary expression yields a value of type "reference to \ldots" (see §8.4 and §8.6.3) that value is immediately dereferenced so that the value of the expression is the object denoted by the reference. If this object is also a reference, it too will be dereferenced, and so on. A reference can be thought of as a name of an object; see §8.6.3.

7.2 Unary operators

Expressions with unary operators group right-to-left.
unary-expression:
  unary-operator expression
  expression ++
  expression --
  ( type-name ) expression
  simple-type-name ( expression-list )
  sizeof expression
  sizeof ( type-name )
  new type-name
  new ( type-name )

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 ...", the type of the result is "...".

The result of the unary & operator is a pointer to the object referred to by the operand. The operand must be an lvalue. If the type of the expression is "...", the type of the result is "pointer to ...".

The result of the unary - operator is the negative of its operand. The operand must be of integral type. The usual arithmetic conversions are performed. The negative of an unsigned quantity is computed by subtracting its value from $2^n$, where $n$ is the number of bits in an int. There is no unary + operator.

The result of the logical negation operator ! is 1 if the value of its operand is 0, 0 if the value of its operand is non-zero. The type of the result is int. It is applicable to any arithmetic type or to pointers.

The - operator yields the one's complement of its operand. The usual arithmetic conversions are performed. The type of the operand must be integral.

The operand of prefix ++ is incremented. The operand must be an lvalue. The value is the new value of the operand, but is not an lvalue. The expression x++ is equivalent to x=x+1. See the discussions of addition (§7.4) and assignment operators (§7.14) for information on conversions.

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

The value obtained by applying a postfix ++ is the value of the operand. The operand must be an lvalue. After the result is noted, the object is incremented in the same manner as for the prefix ++ operator. The type of the result is the same as the type of the operand.

The value obtained by applying a postfix -- is the value of the operand. The operand must be an lvalue. After the result is noted, the object is decremented in the manner as for the prefix -- operator. The type of the result is the same as the type of the operand.

A simple-type-name (§8.2) followed by a parenthesized expression causes the value of the expression to be converted to the named type. To express conversion to a type that does not have a simple name the type-name (§8.7) must be parenthesized; in this case the expression need not be parenthesized. This construction is called a cast. The method for defining conversions for user-defined types (classes) is described in §8.5.5 and §8.5.6. For user-defined types an expression list, rather than a simple expression, can be used; see §8.5.5.

The sizeof operator yields the size, in bytes, of its operand. (A byte is undefined by the language except in terms of the value of sizeof. However, in all existing implementations a byte is the space required to hold a char.) When applied to an array, the result is the total number of bytes in the array. The size is determined from the declarations of the objects in the expression. This expression is semantically an unsigned constant and may be used anywhere a constant is
required. Its major use is in communication with routines like storage allocators and I/O systems.

The `sizeof` operator may also be applied to a parenthesized type name. In that case it yields the size, in bytes, of an object of the indicated type.

The `new` operator creates an object of the `type-name` (see §8.7) to which it is applied. 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 applied to an object of type `T` it therefore generally returns a value of type `T*`. However, the type yielded for an array type `T[ ]` is `T*`. For example, both `new int` and `new int[10]` return an `int*`. See §17 for details of how the free store is managed.

7.3 Multiplicative operators

The multiplicative operators `*`, `/`, and `%` group left-to-right. The usual arithmetic conversions are performed.

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

The binary `*` operator indicates multiplication. The `*` operator is associative and expressions with several multiplications at the same level may be rearranged by the compiler.

The binary `/` operator indicates division. When positive integers are divided truncation is toward 0, but the form of truncation is machine-dependent if either operand is negative. On all machines covered by this manual, the remainder has the same sign as the dividend. It is always true that `(a/b)*b + a%b` is equal to `a` (if `b` is not 0).

The binary `%` operator yields the remainder from the division of the first expression by the second. The usual arithmetic conversions are performed. The operands must not be floating.

7.4 Additive operators

The additive operators `+` and `-` group left-to-right. The usual arithmetic conversions are performed. There are some additional type possibilities for each operator.

```
additive-expression:
  expression + expression
  expression - 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 latter is in all cases converted to an address offset by multiplying it by the length of the object to which the pointer points. The result is a pointer of the same type as the original pointer, and 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.

No further type combinations are allowed for pointers.

The `+` operator is associative and expressions with several additions at the same level may be rearranged by the compiler.

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

If two pointers to objects of the same type are subtracted, the result is converted (by division by the length of the object) to an integer representing the number of objects separating the pointed-to objects. Depending on the machine the resulting integer may be of type `int` or type `long`; see §2.6. This conversion will in general give unexpected results unless the pointers point to objects in
the same array, since pointers, even to objects of the same type, do not necessarily differ by a
multiple of the object-length.

7.5 Shift operators

The shift operators << and >> group left-to-right. Both perform the usual arithmetic conversions
on their operands, each of which must be integral. Then the right operand is converted to int;
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 of the object in bits.

shift-expression:
  expression << expression
  expression >> expression

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 is unsigned; otherwise it may be arithmetic (fill by a copy of the sign
bit).

7.6 Relational operators

The relational operators group left-to-right, but this fact is not very useful; a<b<c does not mean
what it seems to.

relational-expression:
  expression < expression
  expression > expression
  expression <= expression
  expression >= 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. Two pointers may be compared; the result
depends on the relative locations in the address space of the pointed-to objects. Pointer
comparison is portable only when the pointers point to objects in the same array.

7.7 Equality operators

equality-expression:
  expression == expression
  expression != 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.)

A pointer may be compared to an integer only if the integer is the constant 0. A pointer to which
0 has been assigned is guaranteed not to point to any object, and will appear to be equal to 0; in
conventional usage, such a pointer is considered to be null.

7.8 Bitwise AND operator

and-expression:
  expression & expression

The & operator is associative and expressions involving & may be rearranged. The usual arithmetic
conversions are performed; the result is the bitwise AND function of the operands. The operator
applies only to integral operands.
7.9 Bitwise exclusive OR operator

\[ \text{exclusive-or-expression:} \]
\[ \text{expression ^ expression} \]

The ^ operator is associative and expressions involving ^ may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise exclusive OR function of the operands. The operator applies only to integral operands.

7.10 Bitwise inclusive OR operator

\[ \text{inclusive-or-expression:} \]
\[ \text{expression | expression} \]

The | operator is associative and expressions involving | may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise inclusive OR function of its operands. The operator applies only to integral operands.

7.11 Logical AND operator

\[ \text{logical-and-expression:} \]
\[ \text{expression & expression} \]

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 is 0.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always int.

7.12 Logical OR operator

\[ \text{logical-or-expression:} \]
\[ \text{expression | expression} \]

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 value of the first operand is non-zero.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always int.

7.13 Conditional operator

\[ \text{conditional-expression:} \]
\[ \text{expression ? expression : expression} \]

Conditional expressions group right-to-left. The first expression is evaluated and if it is non-zero, the result is the value of the second expression, otherwise that of third expression. If possible, the usual arithmetic conversions are performed to bring the second and third expressions to a common type; otherwise, if both are pointers of the same type, the result has the common type; otherwise, one must be a pointer and the other the constant 0, and the result has the type of the pointer. Only one of the second and third expressions is evaluated.

7.14 Assignment operators

There are a number of assignment operators, all of which group right-to-left. All require an lvalue as their left operand, and the type of an assignment expression is that of its left operand. The value is the value stored in the left operand after the assignment has taken place.
Assignment-expression:
expression assignment-operator expression

Assignment-operator: one of

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

In the simple assignment with =, the value of the expression replaces that of the object referred to by the left hand operand. If both operands have arithmetic type, the right operand is converted to the type of the left preparatory to the assignment. Both operands may be class objects of the same type or pointer objects of the same type. Objects of some derived classes cannot be assigned; see §8.5.3. A pointer to a class may be assigned to a pointer to a public base class of that class; see §8.5.3. Any pointer may be assigned to a pointer of type void*. The constant 0 may be assigned to a pointer, and it is guaranteed that this value will produce a null pointer distinguishable from a pointer to any object.

Since a reference is implicitly dereferenced, assignment to a object of type “reference to ...” assigns to the object denoted by the reference.

The behavior of an expression of the form E1 op = E2 may be inferred by taking it as equivalent to E1 = E1 op (E2); however, 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 §7.4; all right operands and all non-pointer left operands must have arithmetic type.

7.15 Comma operator

Comma-expression:
expression , expression

A pair of expressions separated by a comma is evaluated left-to-right and the value of the left expression is discarded. The type and value of the result are the type and value of the right operand. This operator groups left-to-right. In contexts where comma is given a special meaning, for example in lists of actual arguments to functions (§7.1) and lists of initializers (§8.6), 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.

7.16 Overloaded operators

Most operators can be declared to accept class objects as operands (see §8.5.10). It is not possible to change the precedence of operators. It is not possible to change the meaning of operators when applied to non-class objects. The pre-defined meaning of the operators = and (unary) & when applied to class objects can be changed.

The meanings of some operators are defined to be equivalent to some combination of other operators on the same arguments. For example, ++a means a++1. Such relations do not hold for defined operators unless the user defines them that way. Some operators, for example assignment, require an operand to be an lvalue; this is not required for defined operators.

7.16.1 Unary operators

A unary operator, whether prefix or postfix, can be defined by either by a member function (see §8.5.4) taking no arguments or a friend function (see §8.5.9) taking one argument. Thus, for any unary operator @, both x@ and @x can be interpreted as either x.operator@() or operator@x(x). If both operator@ functions are defined the former interpretation is used. When the operators ++ and -- are overloaded, it is not possible to distinguish prefix application from postfix application.
7.16.2 Binary operators
A binary operator can be defined either by a member function taking one argument or by a friend 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 operator@ functions are defined the former interpretation is used.

7.16.3 Special operators
Function call

\[
\text{primary-expression} \ ( \text{expression-list}_{\text{opt}} )
\]

and subscripting

\[
\text{primary-expression} \ [ \ \text{expression} \ ]
\]

are considered binary operators. The names of the defining functions are operator() and operator[], respectively. Thus, a call x(arg) is interpreted as x.operator()(arg) for a class object x. The type of the argument list is defined by the operator() function. A subscripting x[y] is interpreted as x.operator[](y). The type of the argument is defined by the operator[] function.

8. DECLARATIONS
Declarations are used to specify the interpretation given to each identifier; they do not necessarily reserve storage associated with the identifier. Declarations have the form

\[
declaration: \\
\quad \text{decl-specifiers}_{\text{opt}} \ \text{declator-list}_{\text{opt}} ; \\
\quad \text{name-declaration} \\
\quad \text{asm-declaration}
\]

The declarators in the declator-list contain the identifiers being declared. Only in external function definitions (§10.1) or external function declarations may the decl-specifiers be omitted. Only when declaring a class (§8.5) or enumeration (§8.10), that is when the decl-specifiers is a class-specifier or enum-specifier, may the declator-list be empty. Name-declarations are described in §8.8; asm declarations are described in §8.11.

\[
decl-specifiers: \\
\quad \text{decl-specifier} \ \text{decl-specifiers}_{\text{opt}}
\]

\[
decl-specifier: \\
\quad \text{ss-specifier} \\
\quad \text{type-specifier} \\
\quad \text{fct-specifier} \\
\quad \text{friend} \\
\quad \text{typedef}
\]

The list must be self-consistent in a way described below.

8.1 Scope and storage class specifiers
The ss-specifiers are:

\[
\quad \text{ss-specifier:} \\
\quad \text{auto} \\
\quad \text{static} \\
\quad \text{extern} \\
\quad \text{register}
\]
Declarations using the auto, static, and register specifiers also serve as definitions in that they cause an appropriate amount of storage to be reserved. If an extern declaration is not a definition (that is, neither a data declaration with an initializer nor a function declaration with a body) there must be an external definition (§10) for the given identifiers somewhere else.

A register declaration is best thought of as an auto declaration, together with a hint to the compiler that the variables declared will be heavily used. The hint may be ignored. The address-of operator & cannot be applied to them.

The auto or register specifiers can only be used for names declared in a block and for formal parameters. There can be no static functions within a block, nor any static formal arguments. The scope of name declared with the static specifier outside a function or a class is file (§4.1). The scope of the name of an object or function declared extern is program (§4.1).

At most one ss-specifier may be given in a declaration. If the ss-specifier is missing from a declaration, the storage class is taken to be automatic inside a function and static outside. Exception: functions are never automatic. If the ss-specifier is missing from a declaration, the scope of the name is taken to be local in a block, class in a class declaration, and file elsewhere. Exception: The scope of the name of a function that is declared but not defined (§10) is taken to be program unless it is explicitly declared static.

Some specifiers can only be used in function declarations and definitions:

    fct-specifiers:
        overload
        inline
        virtual

The overload specifier enables a single name to be used to denote several functions; see §8.9.

The inline specifier is only a hint to the compiler, does not affect the meaning of a program, and can be ignored. It is used to indicate that when the function is called inline substitution of the function body is to be preferred to the usual function call implementation. It can only be used in function definitions (§10.1). A function (§8.5.2 and §8.5.9) defined within the declaration of its class is inline by default.

The virtual specifier can only be used in declarations of class members; see §8.5.4.

The friend specifier is used to override the name hiding rules for class members and can only be used within a class declaration; see §8.5.9.

The typedef specifier is used to introduce a name for a type; see §8.8.

### 8.2 Type specifiers

The type-specifiers are

    type-specifier:
        simple-type-name
        class-specifier
        enum-specifier
        const


```

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

The words `long`, `short`, and `unsigned` may be thought of as adjectives. They can be applied to `int`; `unsigned` can also be applied to `char`, `short`, and `long`. The word `const` may be added to any legal type-specifier. Otherwise, at most one type-specifier may be given in a declaration. An object of `const` type is not an lvalue. If the type-specifier is missing from a declaration, it is taken to be `int`.

Class and enumeration specifiers are discussed in §8.5 and §8.10, respectively.

8.3 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
   init-declarator , declarator-list
```

```
init-declarator:
   declarator initializer opt
```

Initializers are discussed in §8.6. The specifiers in the declaration indicate the type and storage class of the objects to which the declarators refer. Declarators have the syntax:

```
declarator:
   dname
   ( declarator )
   * const opt declarator
   & const opt declarator
   declarator { argument-declaration-list }
   declarator { constant-expression opt }
```

```
dname:
   simple-dname
   typedef-name . simple-dname
```

```
simple-dname:
   identifier
   typedef-name
   - typedef-name
   operator-function-name
   conversion-function-name
```

The grouping is the same as in expressions.
8.4 Meaning of declarators

Each declarator is taken to be an assertion that when a construction of the same form as the declarator appears in an expression, it yields an object of the indicated type and storage class. Each declarator contains exactly one dname; it specifies the identifier that is declared. Except for the declarations of some special functions (see §8.5.2) a dname will be a simple identifier.

If an unadorned identifier appears as a declarator, then it has the type indicated by the specifier heading the declaration.

A declarator in parentheses is identical to the unadorned declarator, but the binding of complex declarators may be altered by parentheses; see the examples below.

Now imagine a declaration

\[
T D1
\]

where \( T \) is a type-specifier (like int, etc.) and \( D1 \) is a declarator. Suppose this declaration makes the identifier have type \( \ldots T, \ldots \) where the \( \ldots \) is empty if \( D1 \) is just a plain identifier (so that the type of \( x \) in \( \text{int } x \) is just \text{int}). Then if \( D1 \) has the form

\[
* D
\]

the type of the contained identifier is \( \ldots \) pointer to \( T \).""

If \( D1 \) has the form

\[
* \text{ const } D
\]

the type of the contained identifier is \( \ldots \) constant pointer to \( T \); that is, the same type as \( * D \), but the contained identifier is not an lvalue.

If \( D1 \) has the form

\[
& D
\]

or

\[
& \text{ const } D
\]

the type of the contained identifier is \( \ldots \) reference to \( T \)."" Since a reference by definition cannot be an lvalue, use of const is redundant. It is not possible to have a reference to void (a void*).

If \( D1 \) has the form

\[
D(\text{argument-declaration-list})
\]

then the contained identifier has the type \( \ldots \) function taking arguments of type \( \text{argument-declaration-list} \) and returning \( T \)."

\[
\text{argument-declaration-list:}
\begin{align*}
\text{arg-declaration-list}_{\text{opt } \ldots \text{opt}} \\
\text{arg-declaration-list:}
\begin{align*}
\text{arg-declaration-list }, \text{argument-declaration} \\
\text{argument-declaration}
\end{align*}
\end{align*}
\]

\[
\text{argument-declaration:}
\begin{align*}
decl-specifiers declarator \\
decl-specifiers declarator = \text{constant-expression}
\end{align*}
\]

If the \( \text{argument-declaration-list} \) terminates with an ellipsis the number of arguments is only known to be equal to or greater than the number of argument types specified; if it is empty the function takes no arguments. All declarations for a function must agree exactly both in the type of the
value returned and in the number and type of arguments. The keyword void may be used to indicate that a function takes no arguments, thus (void) is equivalent to ()

The argument-declaration-list is used to check and convert actual arguments in calls and to check pointer-to-function assignments. If a constant expression is specified as initializer for an argument this value is used as a default argument value. A default value for an argument cannot be redefined by a later declaration. However, a declaration may add default values for arguments not given such values in previous declarations. Default argument values will be used in calls where trailing arguments are missing. In an argument-declaration the identifier in the declarator may be left out (as in an abstract-declarator (§8.7)). If present, the identifier can in fact never be used since it goes out of scope at the end of the function declaration.

If D1 has the form

D[ constant-expression ]

or

D[ ]

then the contained identifier has type "... array of T." In the first case the constant expression is an expression whose value is determinable at compile time, and whose type is int. (Constant expressions are defined in §15.) When several "array of" specifications are adjacent, a multi-dimensional array is created; the constant expressions which specify the bounds of the arrays may be missing only for the first member of the sequence. This elision is useful 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 initialization. In this case the size is calculated from the number of initial elements supplied.

An array may be constructed from one of the basic types, from a pointer, from a structure or union, or from another array (to generate a multi-dimensional array).

Not all the possibilities allowed by the syntax above are permitted. The restrictions are as follows: functions may not return arrays or functions, although they may return pointers to such things; there are no arrays of functions, although there may be arrays of pointers to functions.

As an example, the declaration

int i, *ip, f(), *fip(), (*pfi());

declares an integer i, a pointer ip to an integer, a function f returning an integer, a function fip returning a pointer to an integer, and a pointer pfi to a function which returns an integer. It is especially useful to compare the last two. The binding of *fip() is *(fip()), so that the declaration suggests, and the same construction in an expression requires, the calling of a function fip, and then using indirection through the (pointer) result to yield an integer. In the declarator (*pfi()), the extra parentheses are necessary, as they are in an expression, to indicate that indirection through a pointer to a function yields a function, which is then called. The functions f and fip are declared to take no arguments, and pfi to point to a function which takes no argument.

The declaration

const a = 10, *pc = &a, *const cpc = pc;
int b, *const cp = &b;

declares a: a constant integer, pc: a pointer to a constant integer, cpc: a constant pointer to a constant integer, b: an integer, and cp: a constant pointer to integer. The value of a, 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 illegal operations are:
a = 1; a++; *pc = 2; cp = &a; cpc++;  
Examples of legal operations are:  
b = a; *cp = a; pc++; pc = cpc;

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 (§8.2). The declaration

point(int = 0, int = 0);

declares a function which can be called with zero, one or two arguments of type int. For
example:

point(1,2);
point(1); /* meaning point(1,0); */
point();   /* meaning point(0,0); */

The declaration

printf(char* ...);

declares a function which can be called with varying number and types of arguments. For
example:

printf("hello world");
printf("a=%d b=%d", a, b);
printf("string = %s", st);

However, it must always have a char* as its first argument.

As another example,

float fa[17], *afp[17];
declares an array of float numbers and an array of pointers to float numbers. Finally,

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. The first three have type "array," the last has type int.

8.5 Class declarations

A class specifies a type. Its name becomes a typedef-name (see §8.8) which can be used even
within the class specifier itself. Objects of a class consist of a sequence of members.

class-specifier:
   class-head ( member-list_opt )
   class-head ( member-list_opt, public : member-list_opt )

class-head:
   aggr identifier_opt
   aggr identifier_opt : public_opt, typedef-name

aggr:
   class
   struct
   union
A structure is a class with all members public; see §8.5.8. A union is a structure which holds only one member at a time; see §8.5.12. A member-list may declare data, function, class, typedef, enum, and field members. Fields are discussed in §8.5.13. A member-list may also contain declarations adjusting the visibility of member names; see §8.5.8.

```
member-list:
    member-declaration member-list_opt

member-declaration:
    decl-specifiers member-declarator initializer_opt ;
    function-definition ; opt

member-declarator:
    declarator
    identifier_opt : constant-expression
```

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 to an object of class c1.

The member names in different classes do not conflict with each other or with ordinary variables. Only declarations of static members (§8.5.1) may contain initializers.

A simple example of a struct 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 first character of the tword member of the right subtree of s.

8.5.1 Static members

A data member of a class may be static; function members may not. Members may not be auto, register, or extern. There is only one copy of a static member shared by all objects of the class in a program. A static member mem of class c1 can be referred to as c1::mem, that is without referring to an object. It exists even if no objects of class c1 have been created.

8.5.2 Member functions

A function declared as a member (without the friend specifier (§8.5.9) is called a member function, and is called using the class member syntax (§7.1). For example:
struct tnode {
  char tword[20];
  int count;
  tnode *left;
  tnode *right;
  void set(char*, tnode* l, tnode* r);
};

tnode n1, n2;
set("asdf", &n1, 0);
set("ghjk", 0, 0);

The definition of a member function is considered to be within the scope of its class. This means that it can use names 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 typedef-name.simple-name notation; see §8.3.3. Function definitions are discussed in §10.1. For example:

```c
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 function name tnode.set specifies that the function set is a member of class tnode. This enables the member names `word`, `count`, `left`, and `right` to be used. In a member function a member name refers to the object for which the function was called. Thus, in the call `n1.set(...)`, `tword` refers to `n1.tword`, and in the call `n2.set(...)` it refers to `n2.tword`. In this example, the functions `strlen`, `error`, and `strcpy` are assumed to be declared elsewhere; see §10.1.

In a member function, the keyword `this` points to the object for which the function is called. The type of `this` in a function which is the member of a class `cl` is `cl*`. If `mem` is a member of class `cl`, `mem` and `this->mem` are synonymous in a class `cl` member function (unless `mem` has been used as the name of a local variable in an intermediate scope).

A member function may be defined (§10.1) in the class declaration. Placing a member function definition in the class declaration is just a shorthand for declaring it in the class declaration and then defining it inline (§8.1) just after the class declaration. For example:

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

means

```c
int b;
struct x {
  int f();
  int b;
};
inline x.f() { return b; }
```
The specifier overload (§8.2) need not be used for member functions: if a name is declared to
denote several functions in a class it is overloaded (see §8.9).

It is legal to apply the address-of operator to a member function. However, the type of the
resulting pointer to function is undefined, so that any use of it is implementation dependent.

8.5.3 Derived classes

In the construct

```cpp
aggr identifier : public_opt typedef-name
```

the `typedef-name` must denote a previously declared class, which is called a base class for the class
being declared. The latter is said to be derived from the former. For the meaning of `public` see
§8.5.8. The members of the base class can be referred to as if they were members of the derived
class itself, except when a base member name has been re-defined in the derived class; in this case
the `typedef-name::identifier` notation (§7.1) can be used to refer to the hidden name. For example:

```cpp
struct base {
    int a, b;
};

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

derived d;

d.a = 1;

d.base::b = 2;

d.b = 3;

d.c = 4;
```

assigns to the four members of `d`.

A derived class can itself be used as a base class. It is not possible to derive from a union
(§8.5.12). Assignment is not implicitly defined (see §7.14 and §14.1) for a objects of a class
derived from a class for which `operator=`() has been defined (§8.5.10).

8.5.4 Virtual functions

If a base class `base` contains a virtual (§8.1) function `vf`, and a derived class `derived` also
contains a function `vf` then a call of `vf` for an object of class `derived` invokes `derived::vf`.
For example:

```cpp
struct base {
    virtual void vf();
    void f();
};

struct derived : public base {
    void vf();
    void f();
};
```
derived d;
base* bp = &d;

bp->vf();
bp->f();

The calls invoke derived::vf 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 denoting that object. This implies that the type of objects of classes with virtual functions and objects of classes derived from such classes can be determined at run time.

If a derived class has a member of the same name as a virtual function in a base class the its type must be the same in both classes. A virtual function cannot be a friend (§8.5.9). A function f in a class derived from a class which has a virtual function f is itself considered virtual. A virtual function in a base class must be defined. A virtual function which 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.

8.5.5 Constructors

A member function with the same name as its class is called a constructor. A constructor has no return value type; it is used to construct values of its class type. A constructor can be used 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) );

Objects created in this way are unnamed (unless the constructor was used as an initializer as for zz above), with their lifetime limited to the scope in which they are created. They can often be considered constants of their type. If a class has a constructor it is called for each object of that class before any use is made of the object; see §8.6. A constructor may be overload, but not virtual or friend.

If a class has a base class with a constructor, the constructor for the base class is called before the constructor for the derived class. The constructors for member objects, if any, are executed after the constructor for the base class and before the constructor for the object containing them. See §10.1 for an explanation of how arguments can be specified for a base class constructor, and see §17 for an explanation of how constructors can be used for free storage management.

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

8.5.6 Conversions

A constructor taking a single argument specifies a conversion from its argument type to the type of its class. Such conversions are used implicitly in addition to the usual arithmetic conversions. An assignment to an object of class x is therefore legal if either the assigned value is an x, or if a conversion has been declared from the type of the assigned value to x. Constructors are used similarly for conversion of function arguments (§7.1) and initializers (§8.6). For example:
```cpp
class X { ... X(int); };  
f(X arg) {
    X a = 1;       /* a = X(1) */
    a = 2;         /* a = X(2) */
    f(3);          /* f(X(3)) */
}
```

When no constructor for class X is found which accepts the assigned type, no attempt is made to find other constructors to convert the assigned value into a type which would be acceptable to a constructor for class X. For example:

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

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

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

specifies a conversion from type to X. It will be used implicitly like the constructors above, or it can be called explicitly using the cast notation. For example:

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

X a;
int i = int(a);
i = (int)a;
i = a;
```

In all three cases the value assigned will be found by a call of the function `X::operator int()`.

A user defined type conversion is implicitly applied only if it is unique; see §8.9. Note that if a class X has a conversion to an integral or pointer type declared, an X can be used wherever an expression of such a type is required. For example

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

### 8.5.7 Destructors

A member function of class `cl` named `-cl` is called a destructor. A destructor has no return value and takes no arguments; it is used to destroy values of type `cl` immediately before the object containing them is destroyed. A destructor cannot be overload or friend.

The destructor for a base class is executed after the destructor for its derived class. Destructors for member objects are executed before the destructor for the object they are members of. See §17 for an explanation of how destructors can be used for free storage management.

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

### 8.5.8 Visibility of member names

The members of a class declared with the keyword `class` are private, that is, their names can only be used by member functions (§8.5.2) and friends (see §8.5.10), unless they appear after the
"public:" label; in that case they are public. A public member can be used in any function. A struct is a class with all members public; see §8.5.11.

If the keyword public precedes the base class name in the declaration of a derived class the public members of the base class are public for the derived class; if not, they are private. A public member mem of a private base class base can be declared to be public for the derived class by a declaration of the form

    typedef-name . identifier ;

where the typedef-name denotes the base class and the identifier is the name of a member of the base class. Such a declaration must occur in the public part of the derived class.

Consider

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

    class derived : base {
    int d;
    public:
        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.

8.5.9 Friends

A friend of a class is a non-member function which may use the private member names from the class. A friend is not in the scope of a class and is not called using the member selection syntax (unless it itself is the member of some class). The following example illustrates the differences between members and friends:

    class private {
        int a;
        friend void friend_set(private*, int);
    public:
        void member_set(int);
    };

    void friend_set(private* p, int i) { p->a = i; }

    void private_member_set(int i) { a = i; }

    private obj;

    friend_set(&obj, 10);

    obj.member_set(10);
When a friend declaration refers to an overloaded name or operator only the function specified by the argument types becomes a friend. A member of a class cl1 can be the friend of a class cl2. For example

```cpp
class cl2 {
    friend char* cl1::foo(int);
    ...
};
```

All the functions of a class cl1 can be made friends of a class cl2 by a single declaration

```cpp
class cl2 {
    friend cl1;
    ...
};
```

Placing the definition of a friend function in a class declaration is a shorthand for declaring it and then defining it inline just as for member functions; see §8.5.2.

8.5.10 Operator functions

Most operators can be overloaded to take class object operands.

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

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

The last two operators are function call and subscripting. An operator function (except `operator new()` and `operator delete()`; see §17) must either be a member function or take at least one argument of class type. See also §7.16.

8.5.11 Structures

A structure is a class with all members public. That is

```cpp
struct s { ... };
```

is equivalent to

```cpp
class s { public: ... };
```

A structure may have member functions (including constructors and destructors).

8.5.12 Unions

A union may be thought of as a structure all of whose member objects 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). It is not possible to derive a class from a union. An object of a class with a constructor or a destructor cannot be a member of a union.

A union of the form

```cpp
union { member-list };
```

is called an anonymous union; it defines an unnamed object. The names of the members of an
anonymous union must be distinct from other names in the scope where the union is declared; they can be used directly in that scope without using the usual member access syntax (§8.5). For example

```c
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.

8.5.13 Bit fields

A **member-declarator** of the form

```c
identifier_opt : constant-expression
```

specifies a field; its length is set off from the field name by a colon. Fields are packed into machine integers; they do not straddle words. A field which does not fit into the space remaining in an integer is put into the next word. No field may be wider than a word. Fields are assigned right-to-left on some machines, left-to-right on other machines; see §2.6.

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

Implementations are not required to support any but integer fields. Moreover, even `int` fields may be considered to be unsigned. For these reasons, it is recommended that fields be declared as `unsigned`. The `address-of` operator `&` may not be applied to them, so that there are no pointers to fields.

Fields may not be union members.

8.5.14 Nested classes

A class may be declared within another class. In this case, the scope of the name of the inner class and its public names is restricted to the enclosing class. Except for this restriction the inner class could have been declared outside its enclosing class. Declaring a class within another does not affect the rules for access to private members, nor does it place the member functions of the inner class in the scope of the enclosing class. For example:

```c
int x;

class enclose {
    int x;
    class inner {
        int y;
        f() { x = 1; }
        ...
    };
    g(inner*);
    ...
};

int inner;

enclose.g(inner* p) { ... }
```

In this example, the `x` in `f` refers to the `x` declared before class `enclose`. Since `y` is a private member of `inner`, `g` can not use it. Since `g` is a member of `enclose`, names used in `g` are
resolved in the scope of class `enclose`. Therefore `inner` in the argument declaration for `g` refers to the enclosed type `inner`, and not to the `int`.

### 8.6 Initialization

A declarator may specify an initial value for the identifier being declared. The initializer is preceded by `=`, and consists of an expression or a list of values nested in braces.

```
initializer:
  = expression
  = { initializer-list }
  = { initializer-list , }
  ( expression-list )

initializer-list:
  expression
  initializer-list , initializer-list
  { initializer-list }
```

All the expressions in an initializer for a static or external variable must be constant expressions, which are described in §15, or expressions which reduce to the address of a previously declared variable, possibly offset by a constant expression. Automatic or register variables may be initialized by arbitrary expressions involving constants, previously declared variables and functions.

Static and external variables which are not initialized are guaranteed to start off as 0; automatic and register variables which are not initialized are guaranteed to start off as garbage.

When an initializer applies to a `scalar` (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`.

### 8.6.1 Initializer lists

When the declared variable is an `aggregate` (a class or an array) then the initializer may consist of a brace-enclosed, comma-separated list of initializers for the members of the aggregate, written in increasing subscript or member order. If the array contains subaggregates, this rule applies recursively to the members of the aggregate. If there are fewer initializers in the list than there are members of the aggregate, then the aggregate is padded with 0's.

Braces may be elided as follows. If the initializer 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 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 1-dimensional array which has three members, since no size was specified and there are three initializers.
```c
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. Precisely the same effect could have been achieved by
```c
float y[4][3] = {
    1, 3, 5,
    2, 4, 6,
    3, 5, 7
};
```
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,
```c
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.

### 8.6.2 Class objects

An object with private members cannot be initialized by simple assignment to its members as described above; neither can a union object. An object of a class with a constructor must be initialized. If a class has a constructor which does not take arguments that constructor is used for objects which are not explicitly initialized.

The arguments for a constructor can also be presented as a parenthesized list; this style must be used when creating objects on the free store. For example:
```c
struct complex {
    float re, im;
    complex(float r, float i) { re=r; im=i; }
    complex(float r) { re=r; im=0; }
};

complex zz(1, 2.3);
complex* zp = new complex(1, 2.3);
```
Initialization can also be performed by explicit assignment; conversions are performed. For example,
```c
complex zz1 = complex(1, 2.3);
complex zz2 = complex(123);
complex zz3 = 123;
complex zz4 = zz3;
```
If a constructor taking a reference to an object of its own class exists, it will be invoked when an object is initialized with another object of that class, but not when an object is initialized with a constructor.

An object can be a member of an aggregate only if the object's class does not have a constructor or if one of its constructors takes no arguments. In the latter case that constructor is called when the aggregate is created. If a member of an aggregate is of a class with a destructor then that destructor is called when the aggregate is destroyed.
8.6.3 References

When a variable is declared to be a T&, that is "reference to type T", it can be initialized either by
a pointer to type T, or by an object of type T. In the latter case the address-of operator & will be
implicitly applied. For example:

```c
int i;
int& r1 = i;
int& r2 = &i;
```

Both r1 and r2 will reference i.

Initialization of a reference is treated very differently from assignment to it. As described in §7.1 a
reference is implicitly dereferenced when used. For example

```c
r1 = r2;
```

means copy the integer referenced by r2 into the integer referenced by r1.

A reference must be initialized. Because of the implicit dereferencing the value of a reference
cannot be changed after initialization. A reference can therefore be thought of as a name of an
object.

The expression &r1 yields the address of the object referenced by r1. Thus to get a pointer pp to
denote the same object as r1 one can write pp=&r1.

If the initializer for a reference to type T is not an lvalue an object of type T will be created and
initialized with the initializer using the usual initialization rules. The address of that object then
becomes the value of the reference. The lifetime of an object created in this way is the scope in
which it is created. For example:

```c
double& rr = 1;
```

is legal and rr will point to a double containing the value 1.0.

References are particularly useful as formal argument types.

8.6.4 Character arrays

A final abbreviation allows a char array to be initialized by a string. In this case successive
characters of the string initialize the members of the array. For example,

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

shows a character array whose members are initialized with a string.

8.7 Type names

Sometimes (to specify type conversions explicitly, and as an argument of sizeof or new) it is
desired to supply the name of a data type. This is accomplished using a "type name," which in
essence is a declaration for an object of that type which omits the name of the object.

```c
type-name:
    type-specifier abstract-declarator

abstract-declarator:
    empty
    * abstract-declarator
    abstract-declarator ( argument-declaration-list )
    abstract-declarator [ 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 *
int *[3]
int *(*)
int (*)(*)

name respectively the types "integer," "pointer to integer," "pointer to an array of three integers," "function returning pointer to integer," and "pointer to function returning an integer."

8.8 Typedef

Declarations containing the decl-specifier typedef define identifiers which can be used later as if they were type keywords naming fundamental or derived types.

typedef-name:
    identifier

Within the scope of a declaration involving typedef, each identifier appearing as part of any declarator therein becomes syntactically equivalent to the type keyword naming the type associated with the identifier in the way described in §8.4. The name of a class or an enum is also a typedef-name. For example, after

    typedef int MILES, *KLICKSP;
    struct complex { double re, im; };

the constructions

    MILES distance;
    extern KLICKSP metricp;
    complex z, *zp;

are all legal declarations; the type of distance is int, that of metricp is "pointer to int."

typedef does not introduce brand new types, only synonyms for types which could be specified in another way. Thus in the example above distance is considered to have exactly the same type as any other int object.

A class declaration, however, does introduce 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.

A declaration of the form

    name-declaration:
        aggr identifier ;
        enum identifier ;

specifies that an identifier is the name of some (possibly not yet defined) class or enumeration. Such declarations allows declaration of classes which refer to each other. For example:
class vector;

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

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

8.9 Overloaded function names

When several (different) function declarations are specified for a single name, 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 argument types in the function declarations.

Of the usual arithmetic conversions defined in §6.6 only the conversions char->short->int, int->double, int->long, and float->double are performed for a call of an overloaded function. To overload the name of a non-member function an overload declaration must precede any declaration of the function; see §8.2. For example,

    overload abs;
    int     abs(int);
    double  abs(double);

When an overloaded name is called, the list of functions is scanned in order to find one which can be invoked. For example abs(12) will invoke abs(int) and abs(12.0) will invoke abs(double). Had the order of declarations been reversed, both calls would have invoked abs(double).

If, for a call of an overloaded name, no function is found by the method above, the set of user-defined conversions (§8.5.6) is examined. If there is a unique set of user-defined conversions which makes the call legal, it is implicitly applied. For example:

    class X { ... X(int); };  
class Y { ... Y(int); };  
class Z { ... Z(char*); };

    overload int f(X), f(Y);
    overload int g(X), g(Z);

    f(1);    /* illegal: ambiguous f(X(1)) or f(Y(1)) */
    g(1);    /* g(X(1)) */
    g("asdf");    /* g(Z("asdf")) */

All operator function names are automatically overloaded.

The address-of operator & may only be applied to an overloaded name in an assignment or an initialization where the type expected determines which function to take the address of. For example:

    int operator=(matrix&, matrix&);
    int operator=(vector&, vector&);
    int (*pfm)(matrix&, matrix&) = &operator=;
    int (*pfv)(vector&, vector&) = &operator=;
    int (*pfx)(...) = &operator=;    /* error */
8.10 Enumeration declarations

Enumerations are int types with named constants.

\[
\text{enum-specifier: }
\]
\[
\text{enum identifier opt } \{ \text{enum-list } \}
\]

\[
\text{enum-list:}
\]
\[
\text{enumerator}
\]
\[
\text{enum-list , identifier}
\]

\[
\text{enumerator:}
\]
\[
\text{identifier}
\]
\[
\text{identifier = constant-expression}
\]

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 continue the progression from the assigned value.

The names of enumerators must be distinct from those of ordinary variables. The names of enumerators with different constants must also be distinct. The values of the enumerators need not be distinct.

The role of the identifier in the enum-specifier is entirely analogous to that of the class name; it names a particular enumeration. For example,

\[
\text{enum color } \{ \text{chartreuse, burgundy, claret=20, winedark } \};
\]
\[
\ldots
\]
\[
\text{color *cp, col;}
\]
\[
\ldots
\]
\[
\text{col = claret;}
\]
\[
\text{cp = &col;}
\]
\[
\ldots
\]
\[
\text{if } (*cp == \text{burgundy}) \ldots
\]

makes color the name of a type describing various colors, and then declares cp as a pointer to an object of that type, and col as an object of that type. The possible values are drawn from the set \{0, 1, 20, 21\}.

8.11 Asm declaration

An asm declaration has the form

\[
\text{asm ( string ) ;}
\]

The meaning of an asm declaration is not defined. Typically it is used to pass information through the compiler to an assembler.

9. STATEMENTS

Except as indicated, statements are executed in sequence.

9.1 Expression statement

Most statements are expression statements, which have the form

\[
\text{expression ;}
\]

Usually expression statements are assignments or function calls.
9.2 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:
    statement
    statement statement-list
```

Note that a declaration is an example of a statement (§9.15).

9.3 Conditional statement

The two forms of the conditional statement are

```
if ( expression ) statement
if ( expression ) statement else statement
```

The expression must be of integral or pointer type or of a class type for which a conversion to integral or pointer type is defined (see §8.5.6). 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 0. As usual the "else" ambiguity is resolved by connecting an else with the last encountered else-less if.

9.4 While statement

The while statement has the form

```
while ( expression ) statement
```

The substatement is executed repeatedly so long as the value of the expression remains non-zero. The test takes place before each execution of the statement. The expression is handled as in a conditional statement (§9.3).

9.5 Do statement

The do statement has the form

```
do statement while ( expression ) ;
```

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 is handled as in a conditional statement (§9.3).

9.6 For statement

The for statement has the form

```
for ( statement-1 expression-1_opt ; expression-2_opt ) statement-2
```

This statement is equivalent to

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

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 0; the second expression often specifies an incrementing that is performed after each iteration.
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 statement-1 is a declaration, the scope of the name declared extends to the end of the block enclosing the for-statement.

9.7 Switch statement

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

\[ \text{switch ( expression ) statement} \]

The type of the expression must be of integral or pointer type. Any statement within the statement may be labeled with one or more case prefixes as follows:

\[ \text{case constant-expression :} \]

where the constant expression must be of the same type as the switch expression; the usual arithmetic conversions are performed. No two of the case constants in the same switch may have the same value. Constant expressions are defined in §15.

There may also be at most one statement prefix of the form

\[ \text{default :} \]

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 prefix. If no case constant matches the expression, and if there is a default prefix, control passes to the prefixed statement. If no case matches and if there is no default then none of the statements in the switch is executed.

case and default prefixes in themselves do not alter the flow of control, which continues unimpeded across such prefixes. To exit from a switch, see break, §9.8.

Usually the statement that is the subject of a switch is compound. Declarations may appear at the head of this statement, but initializations of automatic or register variables are ineffective.

9.8 Break statement

The statement

\[ \text{break ;} \]

causes termination of the smallest enclosing while, do, for, or switch statement; control passes to the statement following the terminated statement.

9.9 Continue statement

The statement

\[ \text{continue ;} \]

causes control to pass to the loop-continuation portion of the smallest enclosing while, do, or for statement; that is to the end of the loop. More precisely, in each of the statements

\[ \text{while (...) \{ \quad do \{ \quad for (...) \{ \quad } \]

\[ \ldots \quad \ldots \quad \ldots \]

\[ \text{contin: ; \quad contin: ; \quad contin: ; \quad} \]

\[ \text{\} \quad \} \quad \} \quad \} \text{ while (...) ;} \]

a continue is equivalent to goto contin. (Following the contin: is a null statement, §9.13.)

9.10 Return statement

A function returns to its caller by means of the return statement, which has one of the forms
return;
return expression;

The first form can be used only in functions which does not return a value, that is, a function with the return value type void. The second form 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 type of the function in which it appears. Flowing off the end of a function is equivalent to a return with no returned value.

9.11 Goto statement

Control may be transferred unconditionally by means of the statement

    goto identifier;

The identifier must be a label (§9.12) located in the current function. It is not possible to transfer control past a declaration with an (implicit or explicit) initializer.

9.12 Labeled statement

Any statement may be preceded by label prefixes of the form

    identifier:

which serve to declare the identifier as a label. The only use of a label is as a target of a goto. The scope of a label is the current function, excluding any sub-blocks in which the same identifier has been redeclared. See §4.1.

9.13 Null statement

The null statement has the form

    ;

A null statement is useful to carry a label just before the } of a compound statement or to supply a null body to a looping statement such as while.

9.14 Delete statement

The delete statement has the form

    delete expression;

The result of the expression must be a pointer. The object pointed to is deleted. That is, after the delete statement the object cannot be assured to have a well defined value; see §17. The effect of applying delete to a pointer not obtained from the new operator (§7.1) is undefined. However, deleting a pointer with the value zero is harmless.

9.15 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 were previously declared, the outer declaration is pushed down for the duration of the block, after which it resumes its force.

Any initializations of auto or register variables are performed 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 performed; see §9.11. Initializations of variables with storage class static (§4.2) are performed only once when the program begins execution.
10. EXTERNAL DEFINITIONS

A program consists of a sequence of external definitions. The scope of external definitions persists to the end of the file in which they are declared. The syntax of external definitions is the same as that of declarations, except that only at this level and within class declarations may the code for functions be given.

10.1 Function definitions

Function definitions have the form

\[
\text{function-definition:} \\
\text{decl-specifiers opt, function-declarator base-class-initializer opt, function-body}
\]

The decl-specifiers register, auto, typedef may not be used, and friend, and virtual may only be used within a class declaration (§8.5). A function declarator is similar to a declarator for a “function returning ...” except that it includes the names of the formal parameters of the function being defined. Function declarators have the form

\[
\text{function-declarator:} \\
\text{declarator ( argument-declaration-list )}
\]

The form of an argument-declaration-list is specified in §8.4. If an argument is specified register, the corresponding actual parameter will be copied, if possible, into a register at the outset of the function. If a constant expression is specified as initializer for an argument this value is used as a default argument value.

The function-body has the form

\[
\text{function-body:} \\
\text{compound-statement}
\]

A simple example of a complete function definition is

\[
\text{int max(int a, int b, int c) } \\
\{ \\
\text{int m = (a > b) ? a : b; } \\
\text{return (m > c) ? m : c ; } \\
\}
\]

Here int is the type-specifier; max(int a, int b, int c) is the function-declarator; \{ ... \} is the block giving the code for the statement.

Since in expression context an array name (in particular as an actual parameter) is taken to mean a pointer to the first element of the array, declarations of formal parameters declared “array of ...” are adjusted to read “pointer to ...”.

A base class initializer has the form

\[
\text{base-class-initializer:} \\
\text{: ( argument-list opt )}
\]

It is used to specify arguments for a base class constructor in a constructor for a derived class. For example:

\[
\text{struct base \{ base(int); ... \}; } \\
\text{struct derived : base \{ derived(int); ... \}; } \\
\text{derived.derived(int a) : (a+1) \{ ... \}} \\
\text{derived d(10);}
\]

The base class’s constructor is called for the object d with the argument 11.
10.2 External data definitions
An external data definition has the form

\[ \text{data-definition:} \]
\[ \text{declaration} \]

The storage class of such data is static.

If there is more than one external data definition of the same name, the definitions must agree exactly in type and storage class and all initializers (if any) must have the same value.

11. SCOPE RULES
See §4.1.

12. COMPILER CONTROL LINES
The compiler contains a preprocessor capable of macro substitution, conditional compilation, and inclusion of named files. 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 most uses of \#define.

12.1 Token replacement
A compiler-control line of the form

\[ \text{\#define identifier token-string} \]

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

\[ \text{\#define identifier( identifier , \ldots , identifier ) token-string} \]

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 parameter 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 parameters must be the same. Strings and character constants in the token-string are scanned for formal parameters, 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 \( \text{\backslash } \) at the end of the line to be continued. A control line of the form

\[ \text{\#undef identifier} \]

causes the identifier's preprocessor definition to be forgotten.

12.2 File inclusion
A compiler control line of the form

\[ \text{\#include "filename"} \]

causes the replacement of that line by the entire contents of the file \text{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.

12.3 Conditional compilation
A compiler control line of the form

#define expression

checks whether the expression evaluates to non-zero. The expression must be a constant expression as discussed in §15; the following additional restriction applies here: the constant expression may not contain sizeof or an enumeration constant.) 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

#undef 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 an #else or, lacking an #else, the #endif, are ignored.

These constructions may be nested.

12.4 Line control
For the benefit of other preprocessors which generate C++ programs, a line of the form

#line constant "filename"

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.

13. IMPLICIT DECLARATIONS
See §8.1.

14. TYPES REVISITED
This section summarizes the operations which can be performed on objects of certain types.

14.1 Classes
Class objects may be assigned, passed as arguments to functions, and returned by functions (except objects of some derived classes; see §8.5.3). Other plausible operators, such as equality comparison, can be defined by the user; see §8.5.10.
14.2 Functions

There are only two things that can be done with a function: call it, or take its address. If the name of a function appears in an expression not in the function-name position of a call, a pointer to the function is generated. Thus, to pass one function to another, one might say

```c
typedef int (*PF)();
extern g(PF);
extern f();
...
g(f);
```

Then the definition of g might read

```c
g(PF funcp)
{
...
    (*funcp)();
...  
}
```

Notice that f must be declared explicitly in the calling routine since its appearance in g(f) was not followed by (.

14.3 Arrays, pointers, and subscripting

Every time an identifier of array type appears in an expression, it is converted into a pointer to the first member of the array. Because of this conversion, arrays are not lvalues. By definition, the subscript operator [] is interpreted in such a way that E1[E2] is identical to *(E1)*(E2)). Because of the conversion rules which 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×j×⋯×k, then E appearing in an expression is converted to a pointer to an (n-1)-dimensional array with rank j×⋯×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

```c
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 i is converted to the type of x, which involves multiplying i by the length 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.

14.4 Explicit pointer conversions

Certain conversions involving pointers are permitted but have implementation-dependent aspects. They are all specified by means of an explicit type-conversion operator, §§7.2 and 8.7.
A pointer may be converted to any of the integral types large enough to hold it. Whether an int or long is required is machine dependent. The mapping function is also machine dependent, but is intended to be unsurprising to those who know the addressing structure of the machine. Details for some particular machines were given in §2.6.

An object of integral type may be explicitly converted to a pointer. The mapping always carries an integer converted from a pointer back to the same pointer, but is otherwise machine dependent.

A pointer to one type may be converted to a pointer to another type. 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 a 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.

15. CONSTANT EXPRESSIONS

In several places C++ requires expressions which evaluate to a constant: as array bounds (§8.3), as case expressions (§9.7), as default function arguments (§8.3), and in initializers (§8.6). In the first case, the expression can involve only integer constants, character constants, enumeration constants, names declared const, and sizeof expressions, possibly connected by the binary operators

\[
+ - * / \% & | ^ << >> == != < > <= >= && !
\]

or by the unary operators

- - !

or by the ternary operator

?:

Parentheses can be used for grouping, but not for function calls.

More latitude is permitted for the other three uses; besides constant expressions as discussed above, float constants are permitted, and one can also apply the unary & operator to external or static objects, or to external or static arrays subscripted with a constant expression. The unary & can also be applied implicitly by appearance of unsubscripted arrays and functions. The basic rule is that initializers must evaluate either to a constant or to the address of a previously declared external or static object plus or minus a constant.

Less latitude is allowed for constant expressions after #if; names declared const, sizeof expressions, and enumeration constants are not permitted.

16. PORTABILITY CONSIDERATIONS

Certain parts of C++ are inherently machine dependent. The following list of potential trouble spots is not meant to be all-inclusive, but to point out the main ones.

Purely hardware issues like word size and the properties of floating point arithmetic and integer division have proven in practice to be not much of a problem. Other facets of the hardware are reflected in differing implementations. Some of these, particularly sign extension (converting a negative character into a negative integer) and the order in which bytes are placed in a word, are a nuisance that must be carefully watched. Most of the others are only minor problems.

The number of register variables that can actually be placed in registers varies from machine to machine, as does the set of valid types. Nonetheless, the compilers all do things properly for their own machine; excess or invalid register declarations are ignored.
Some difficulties arise only when dubious coding practices are used. It is exceedingly unwise to write programs that depend on any of these properties.

The order of evaluation of function arguments is not specified by the language. It is right-to-left on some machines left-to-right on others. The order in which side effects take place is also unspecified.

Since character constants are really objects of type int, multi-character character constants may be permitted. The specific implementation is very machine dependent, however, because the order in which characters are assigned to a word varies from one machine to another. On some machines fields are assigned left-to-right in a word, in others right-to-left.

These differences are invisible to isolated programs which do not indulge in type punning (for example, by converting an int pointer to a char pointer and inspecting the pointed-to storage), but must be accounted for when conforming to externally-imposed storage layouts.

17. FREE STORE

The new operator (§7.2) will call the function

    extern void* operator new(long);

To obtain storage. The argument specifies the number of bytes required. The store will be uninitialized. If operator new() cannot find the amount of store required it will return zero. The delete operator will call the function

    extern void operator delete(void*);

To free the store pointed to for re-use. The effect of calling operator delete() for a pointer not obtained from operator new() is undefined, and so is the effect of calling operator delete() twice for the same pointer. However, deleting a pointer with the value zero is harmless.

Default versions of operator new() and operator delete() are provided, but a user may supply others more suitable for particular applications.

When a class object is created using the new operator the constructor will (implicitly) use operator new() to obtain the store needed. 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 deallocation operation for objects of its class. For example:

    class cl {
      int v[10];
      cl() { this = my_own_allocator( sizeof(cl) ); }
      ~cl() { my_own_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 objects) and zero otherwise.

If a derived class assigns to this, the call to its base class's constructor (if any) will take place after the assignment so that the base class constructor will refer to the object allocated by the derived class's constructor. If a base class's constructor assigns to this, the new value will also be used by the derived class's constructor (if any).

18. SYNTAX SUMMARY

This summary of C++ syntax is intended to be an aid to comprehension. It is not an exact statement of the language.
18.1 Expressions

expression:
    term
    expression binary-operator expression
    expression ? expression : expression
    expression-list

term:
    primary
    * term
    & term
    - term
    ! term
    ~ term
    ++ term
    -- term
    term ++
    term --
    ( type-name ) expression
    simple-type-name ( expression-list )
    sizeof expression
    sizeof ( type-name )
    new type-name
    new ( type-name )

primary:
    id
    :: identifier
    constant
    string
    this
    ( expression )
    primary[ expression ]
    primary ( expression-list opt )
    primary . id
    primary -> id

id:
    identifier
    operator-function-name
    typedef-name :: identifier
    typedef-name :: operator-function-name

expression-list:
    expression
    expression-list , expression

operator:
    unary-operator
    binary-operator
    special-operator
    free-store-operator

Binary operators have precedence decreasing as indicated:
binary-operator: one of
*  /  %
+  -
<<  >>
<  >
==  !=
&
^  
| 
&&
||
+=  -=  *= /= %= ^= &= |= >>= <<=

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

special-operator: one of
()  []

free-store-operator: one of
new  delete

type-name:
   decl-specifiers abstract-declarator

abstract-declarator:
   empty
   * abstract-declarator
   abstract-declarator ( argument-declaration-list )
   abstract-declarator [ constant-expression opt ]

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

typedef-name:
   identifier

18.2 Declarations

declaration:
   decl-specifiers opt  declarator-list opt ;
   name-declaration
   asm-declaration
name-declaration:
    aggr identifier;
    enum identifier;

aggr:
    class
    struct
    union

asm-declaration:
    asm(string);

decl-specifiers:
    decl-specifier decl-specifiers_opt

decl-specifier:
    ss-specifier
    type-specifier
    fct-specifier
    friend
    typedef

type-specifier
    simple-type-name
    class-specifier
    enum-specifier
    const

ss-specifier:
    auto
    extern
    register
    static

fct-specifier:
    inline
    overload
    virtual

declarator-list:
    init-declarator
    init-declarator, declarator-list

init-declarator:
    declarator initializer_opt

declarator:
    dname
    (declarator)
    * const_opt declarator
    & const_opt declarator
    declarator_opt (argument-declaration-list)
    declarator [ constant-expression_opt]
dname:
  simple-dname
typedef-name . simple-dname

simple-dname:
  identifier
typedef-name
  - typedef-name
  operator-function-name
  conversion-function-name

operator-function-name:
  operator operator

conversion-function-name:
  operator type

argument-declaration-list:
  arg-declaration-list_opt . . . opt

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

argument-declaration:
  decl-specifiers declarator
dcl-specifiers declarator = constant-expression

class-specifier:
  class-head { member-list_opt }
  class-head { member-list_opt public : member-list_opt }

class-head:
  agrgr identifier_opt
  agrgr identifier_opt : public_opt typedef-name

member-list:
  member-declaration member-list_opt

member-declaration:
  decl-specifiers_opt member-declarator initializer_opt ;
  function-definition ; opt

member-declarator:
  declarator
  identifier_opt : constant-expression

initializer:
  = expression
  = { initializer-list }
  = { initializer-list , }
  ( expression-list )
initializer-list:
   expression
   initializer-list , initializer-list
   { initializer-list }

enum-specifier:
   enum identifier_opt { enum-list }

enum-list:
   enumerator
   enum-list , enumerator

default:
   enumerator
   identifier = constant-expression

18.3 Statements

compound-statement:
   { statement-list_opt }

statement-list:
   statement
   statement statement-list

statement:
   declaration
   expression ;
   if ( expression ) statement
   if ( expression ) statement else statement
   while ( expression ) statement
   do statement while ( expression ) ;
   for ( statement expression_opt ; expression_opt ) statement
   switch ( expression ) statement
   case constant-expression : statement
   default : statement
   break ;
   continue ;
   return expression_opt ;
   goto identifier ;
   identifier : statement
   delete expression ;

18.4 External definitions

program:
   external-definition
   external-definition program

external-definition:
   function-definition
   declaration
function-definition:
    decl-specifiers\textsubscript{opt} function-declarator base-class-initializer\textsubscript{opt} function-body

function-declarator:
    declator ( argument-declaration-list )

function-body:
    compound-statement

base-class-initializer:
    : ( argument-list\textsubscript{opt} )

18.5 Preprocessor

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

19. DIFFERENCES FROM C

19.1 Extensions

The types of function arguments can be specified (§8.4) and will be checked (§7.1). Type conversions will be performed (§7.1).

Single-precision floating arithmetic may be used for\texttt{float} expressions; §6.2.

Function names can be overloaded; §8.6.

Operators can be overloaded; §7.16, §8.5.10.

Functions can be inline substituted; §8.1.

Data objects can be\texttt{const}; §8.3.

Objects of reference type can be declared; §8.3, §8.6.3.

A\texttt{free} store is provided by the\texttt{new} and\texttt{delete} operators; §17.

Classes can provide data hiding (§8.5.8), guaranteed initialization (§8.6.2), user-defined conversions (§8.5.6), and dynamic typing through use of virtual functions (§8.5.4).

The name of a class is a type name; §8.5.

Any pointer can be assigned to a\texttt{void*} without use of a cast; §7.14.

A declaration within a block is a statement; §9.15.

Anonymous unions can be declared; §8.5.12.

19.2 Summary of incompatibilities

Most constructs in C are legal in C++ with their meaning unchanged. The exceptions are as follows
Programs using one of the new keywords

\begin{verbatim}
class  const  delete  friend  inline
new    operator  overload  public  this
virtual
\end{verbatim}

as identifiers are not legal.

The function declaration

\begin{verbatim}
f();
\end{verbatim}

means that \( f \) takes no arguments, in C it means that \( f \) could take arguments of any type at all.

The default scope of a name declared outside any block or class is \emph{file}; in C it was \emph{program}. This implies that to make a name visible to functions in other files a name must explicitly be declared \emph{extern}; thus

\begin{verbatim}
int a; f() {}
\end{verbatim}
in C++ is

\begin{verbatim}
static int a; static f() {}
\end{verbatim}
in C, and

\begin{verbatim}
extern int a; int a; extern f() {}
\end{verbatim}
in C++ is

\begin{verbatim}
int a; f() {}
\end{verbatim}
in C. However, note that

\begin{verbatim}
int f();
\end{verbatim}

means

\begin{verbatim}
extern int f();
\end{verbatim}
in both C++ and C.

Since class names in C++ are in the same name space as other names, constructs like

\begin{verbatim}
struct s { int a; } s;
struct stat stat();
\end{verbatim}
cannot be used.

19.3 Anachronisms

A class name can be prefixed with the one of the keywords \texttt{class}, \texttt{struct}, or union in the declaration of class objects, pointers, etc. For example

\begin{verbatim}
struct s a, *p;
struct s f();
\end{verbatim}

Programs using the old function definition syntax
old-function-definition:
    decl-specifiers_opt old-function-declarator declaration-list function-body

old-function-declarator:
    declarator ( parameter-list )

parameter-list:
    identifier
    identifier , identifier

for example

    max(a,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.
1. INTRODUCTION ................................................. 1

2. LEXICAL CONVENTIONS ........................................ 1
   2.1 Comments .............................................. 1
   2.2 Identifiers (Names) .................................... 1
   2.3 Keywords ............................................. 1
   2.4 Constants ............................................ 1
   2.5 Strings ............................................... 2
   2.6 Hardware characteristics ............................. 3

3. SYNTAX NOTATION ............................................. 3

4. WHAT'S IN A NAME? ............................................ 3
   4.1 Scopes ............................................... 3
   4.2 Storage classes ....................................... 4
   4.3 Fundamental types .................................... 4
   4.4 Derived types ........................................ 4

5. OBJECTS AND LVALUES ....................................... 5

6. CONVERSIONS .................................................. 5
   6.1 Characters and integers ............................... 5
   6.2 Float and double ..................................... 5
   6.3 Floating and integral ................................ 5
   6.4 Pointers and integers ................................ 5
   6.5 Unsigned ............................................. 5
   6.6 Arithmetic conversions ............................... 6
   6.7 Void .................................................. 6

7. EXPRESSIONS ................................................. 6
   7.1 Primary expressions ................................... 7
   7.2 Unary operators ....................................... 8
   7.3 Multiplicative operators .............................. 10
   7.4 Additive operators .................................... 10
   7.5 Shift operators ....................................... 11
   7.6 Relational operators .................................. 11
   7.7 Equality operators .................................... 11
   7.8 Bitwise AND operator ................................ 11
   7.9 Bitwise exclusive OR operator ....................... 12
   7.10 Bitwise inclusive OR operator ....................... 12
   7.11 Logical AND operator ................................ 12
   7.12 Logical OR operator .................................. 12
   7.13 Conditional operator ................................ 12
   7.14 Assignment operators ................................ 12
   7.15 Comma operator ...................................... 13
   7.16 Overloaded operators ................................ 13

8. DECLARATIONS ............................................... 14
   8.1 Scope and storage class specifiers ................... 14
   8.2 Type specifiers ....................................... 15
   8.3 Declarators .......................................... 16
   8.4 Meaning of declarators ................................ 17
   8.5 Class declarations .................................... 19
   8.6 Initialization ........................................ 28
   8.7 Type names ........................................... 30
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.8</td>
<td>Typedef</td>
<td>31</td>
</tr>
<tr>
<td>8.9</td>
<td>Overloaded function names</td>
<td>32</td>
</tr>
<tr>
<td>8.10</td>
<td>Enumeration declarations</td>
<td>33</td>
</tr>
<tr>
<td>8.11</td>
<td>Asm declaration</td>
<td>33</td>
</tr>
<tr>
<td>9.</td>
<td>STATEMENTS</td>
<td>33</td>
</tr>
<tr>
<td>9.1</td>
<td>Expression statement</td>
<td>33</td>
</tr>
<tr>
<td>9.2</td>
<td>Compound statement, or block</td>
<td>34</td>
</tr>
<tr>
<td>9.3</td>
<td>Conditional statement</td>
<td>34</td>
</tr>
<tr>
<td>9.4</td>
<td>While statement</td>
<td>34</td>
</tr>
<tr>
<td>9.5</td>
<td>Do statement</td>
<td>34</td>
</tr>
<tr>
<td>9.6</td>
<td>For statement</td>
<td>34</td>
</tr>
<tr>
<td>9.7</td>
<td>Switch statement</td>
<td>35</td>
</tr>
<tr>
<td>9.8</td>
<td>Break statement</td>
<td>35</td>
</tr>
<tr>
<td>9.9</td>
<td>Continue statement</td>
<td>35</td>
</tr>
<tr>
<td>9.10</td>
<td>Return statement</td>
<td>35</td>
</tr>
<tr>
<td>9.11</td>
<td>Goto statement</td>
<td>36</td>
</tr>
<tr>
<td>9.12</td>
<td>Labeled statement</td>
<td>36</td>
</tr>
<tr>
<td>9.13</td>
<td>Null statement</td>
<td>36</td>
</tr>
<tr>
<td>9.14</td>
<td>Delete statement</td>
<td>36</td>
</tr>
<tr>
<td>9.15</td>
<td>Declaration statement</td>
<td>36</td>
</tr>
<tr>
<td>10.</td>
<td>EXTERNAL DEFINITIONS</td>
<td>37</td>
</tr>
<tr>
<td>10.1</td>
<td>Function definitions</td>
<td>37</td>
</tr>
<tr>
<td>10.2</td>
<td>External data definitions</td>
<td>38</td>
</tr>
<tr>
<td>11.</td>
<td>SCOPE RULES</td>
<td>38</td>
</tr>
<tr>
<td>12.</td>
<td>COMPILER CONTROL LINES</td>
<td>38</td>
</tr>
<tr>
<td>12.1</td>
<td>Token replacement</td>
<td>38</td>
</tr>
<tr>
<td>12.2</td>
<td>File inclusion</td>
<td>38</td>
</tr>
<tr>
<td>12.3</td>
<td>Conditional compilation</td>
<td>39</td>
</tr>
<tr>
<td>12.4</td>
<td>Line control</td>
<td>39</td>
</tr>
<tr>
<td>13.</td>
<td>IMPLICIT DECLARATIONS</td>
<td>39</td>
</tr>
<tr>
<td>14.</td>
<td>TYPES REVISITED</td>
<td>39</td>
</tr>
<tr>
<td>14.1</td>
<td>Classes</td>
<td>39</td>
</tr>
<tr>
<td>14.2</td>
<td>Functions</td>
<td>40</td>
</tr>
<tr>
<td>14.3</td>
<td>Arrays, pointers, and subscripting</td>
<td>40</td>
</tr>
<tr>
<td>14.4</td>
<td>Explicit pointer conversions</td>
<td>40</td>
</tr>
<tr>
<td>15.</td>
<td>CONSTANT EXPRESSIONS</td>
<td>41</td>
</tr>
<tr>
<td>16.</td>
<td>PORTABILITY CONSIDERATIONS</td>
<td>41</td>
</tr>
<tr>
<td>17.</td>
<td>FREE STORE</td>
<td>42</td>
</tr>
<tr>
<td>18.</td>
<td>SYNTAX SUMMARY</td>
<td>42</td>
</tr>
<tr>
<td>18.1</td>
<td>Expressions</td>
<td>43</td>
</tr>
<tr>
<td>18.2</td>
<td>Declarations</td>
<td>44</td>
</tr>
<tr>
<td>18.3</td>
<td>Statements</td>
<td>47</td>
</tr>
<tr>
<td>18.4</td>
<td>External definitions</td>
<td>47</td>
</tr>
<tr>
<td>18.5</td>
<td>Preprocessor</td>
<td>48</td>
</tr>
<tr>
<td>19.</td>
<td>DIFFERENCES FROM C</td>
<td>48</td>
</tr>
<tr>
<td>19.1</td>
<td>Extensions</td>
<td>48</td>
</tr>
<tr>
<td>19.2</td>
<td>Summary of incompatibilities</td>
<td>48</td>
</tr>
<tr>
<td>19.3</td>
<td>Anachronisms</td>
<td>49</td>
</tr>
</tbody>
</table>
Index

- addition operator 10
- address operator 9
- backslash character 2
- & bitwise AND operator 11
- ^ bitwise exclusive OR operator 12
- | bitwise inclusive OR operator 12
- . class member operator 7
- => class pointer operator 7
- ?: conditional expression 12
- -- decrement operator 9
- / division operator 10
- == equality operator 11
- escape character 2
- (), function call operator 8
- > greater than operator 11
- >= greater than or equal to operator 11
- ++ increment operator 9
- * indirect operator 9
- != inequality operator 11
- <<= left shift operator 11
- < less than operator 11
- <= less than or equal to operator 11
- & logical AND operator 12
- || logical OR operator 12
- % modulus operator 10
- * multiplication operator 10
- one's complement operator 9
- :: operator 7
- >> right shift operator 11
- ::, scope operator 7
- [], subscript operator 7
- - subtraction operator 10
- - unary minus operator 9
- abstract declarator 30
- addition operator, + 10
- additive operators 10
- address-of field 27
- address-of member operator 22
- address of overloaded name 32
- address of register 15
- address operator, & 9
- aggregate 28
- alignment of fields 27
- alignment restriction 41
- allocation of storage 42
- ambiguity, if-else 34
- anachronism C function definition 49
- anachronism class names 49
- anachronism VCWstructSP 49
- anachronism VCWunionSP 49
- anonymous union 26
- argument evaluation, order of 8
- argument, function 8
- arithmetic conversions 6
- arithmetic operators 10
- arithmetic, pointer 10
- arithmetic types 4
- array declaration 18
- array explanation of subscripting 40
- array initialization 28
- array, multi-dimensional 40
- array, storage order of 40
- array name, conversion of 7
- array name argument 37
- array size, default 18
- asm declaration 33
- assignment, conversion by 13
- assignment expression 12
- assignment operators 22, 26
- assignment of class objects 22
- assignment to pointer to class object 13
- associativity of operators 6
- auto specifier 14
- automatic functions 15
- automatic storage class 4
- automatics, initialization of 28
- backslash character, 2
- base class 22
- base class members, visibility of 25
- binary operator, interpretation of 14
- binary operators, overloaded 14
- bit fields 27
- bitwise operators 11
- bitwise AND operator, & 11
- bitwise exclusive OR operator, ^ 12
- bitwise inclusive OR operator, || 12
- block 34
- blocks, initialization in 36
- block structure 36
- break statement 35
- byte 9
- C, C++ 1, 48
- C++ C compatibility 48
- C++ C default scope 49
- C++ C function declaration 49
- C++ C name space 49
- C compatibility, C++ 48
- C default scope, C++ 49
- C++ extensions 48
- C function declaration, C++ 49
- C function definition, anachronism 49
- C++ keywords 49
- C name space, C++ 49
- call of function 8
- cast of overloaded function 32
- case prefix 35
- cast operator 8, 9, 30
- char type 4, 16
- character constant 2
- character, signed 4
- character array initialization 30
- character-integer conversion 5
- character string 2
- class, base 22
- class declaration 19
conversion, unsigned-integer 5
conversion by assignment 13
conversion by return 36
conversion from user-defined type 34
conversion of array name 7
conversion of function 7
conversion operator, explicit 8, 40
conversions, arithmetic 6
conversions for overloaded functions 32
conversion to fundamental types 24
conversion to integral type 34
class member function 20
class member name 20
class, member of 20
class, name of 19
class scope 3
class struct union 20
class declaration, example of 20
class declarations, nesting of 27
classes 4
classes, operations permitted on 39
classes, overloading 26
class member semantics 8
class member, static 20
class member syntax 8
class member operator, -> 7
class name new type 31
class names, anachronism 49
class object initialization 28, 29
class objects, assignment of 22
class pointer operator, -> 7
comma operator 13
commutative operators 6
comparison, pointer 11
compatibility, C++ C 48
compiler control lines 38
compound statement 34
condition 34
conditional compilation 39
conditional expression, ?: 12
const, example of 18
const type 16
constant, character 2
constant, double 2
constant expression 41
constant, floating 2
constant, hexadecimal 1
constant, integer 1
constant, long 2
constant, octal 1
constant type 16
constant pointer, declaration of 17
constant pointer, example of 18
constants 1, 4
constants, enumeration 33
constructor type-conversion 23
constructors 23
constructors, free store 42
constructors, initialization 29
constructors, overloading and 32
constructors, structures 26
constructors, this 42
constructors, union 26
constructors and initialization 29
constructors for initialization, example of 29
continue statement 35
conversion, character-integer 5
conversion, double-float 5
conversion, float-double 5
conversion, floating-integer 5
conversion, function argument 8
conversion, integer-character 5
conversion, integer-float 5
conversion, integer-long 5
class scope 3
conversion, integer-pointer 10, 41
conversion, integer-unsigned 5
conversion, long-integer 5
conversion, overloading and 32
conversion, pointer 10, 40
conversion, pointer-integer 5, 10, 41
conversion, un signed-integer 5
conversion by assignment 13
conversion by return 36
conversion from user-defined type 34
conversion of array name 7
conversion of function 7
conversion operator, explicit 8, 40
conversions, arithmetic 6
conversions for overloaded functions 32
conversion to fundamental types 24
conversion to integral type 34
data definitions 38
declaration 44
declaration, array 18
declaration, asm 33
declaration, class 19
declaration, enum 33
declaration, example of 18
declaration, extern 15
declaration, field 27
declaration, name 31
declaration, register 15
declaration, storage class 14
declaration, type 17
declaration, typedef 31
declaration of constant pointer 17
declaration of default function arguments 18
declaration of function 17
declaration of function, implicit 8
declaration of function arguments 17
declaration of pointer 17
declaration of reference 17
declarations 14
declaration statement 36
declarator 16
declarator, abstract 30
decrement operator, -- 9
default array size 18
default initialization 28
default scope, C++ C 49
default arguments, example of 19
default function arguments, declaration of 18
default prefix 35
#define 38
defined operator 39
definition, external 37
definition, function 37, 47
definition, member function 21
definition of virtual function 23
definitions, external 47
delete operator 42
delete statement 36
derreferencing 8
dereferencing 8
derived class 22
derived class, example of 22
destructors 24
destructors, free store 42
destructors, this 42
division, integer 41
division operator, / 10
do statement 34
double constant 2
double type 16
double-float conversion 5
eile 39
eempty statement 36
endif 39
eenum declaration 33
enumeration 33
enumeration constants 33
integral types 4
integral type, conversion to 34
interpretation of binary operator 14
interpretation of unary operator 13
keyword, public 24
keywords, C++ 49
keywords, list of 1
label 36
labeled statement 36
left shift operator,  << 11
length of names 1
less than operator, < 11
less than or equal to operator, <= 11
lexical conventions 1
#Line 39
list of keywords 1
list of operators 26
local scope 3
logical AND operator, && 12
logical AND operator,  9
logical OR operator, || 12
long constant 2
long type 4, 16
long-integer conversion 5
lvalue 5, 16
machine dependency 41
macro 48
macro preprocessor 38
member function, class 20
member function, inline 21
member name, class 20
member function, address-of 22
member function definition 21
member function, example of 20
member function, friend of 26
member functions, example of 25
member functions, overloaded 22
member functions, structures 26
member functions, union 26
member name visibility, example of 25
member names, visibility of 24
member of class 20
members, private 24
members, public 24
missing storage class specifier 15
modulus operator, % 10
multi-dimensional array 40
multiplication operator, * 10
multiplicative operators 10
name 1, 3
name, class member 20
name declaration 31
name, overloaded function 32
name of class 19
names, length of 1
name space, C++ C 49
nested class declarations, example of 27
nesting of class declarations 27
new operator 42
new operator, 8, 10
new type, class name 31
new type, typedef 31
NULL pointer 13
null statement 36
numbers, size of 3
object 3, 5
octal constant 1
one's complement operator,  9
operations permitted on classes 39
operations permitted on functions 40
operator, :: 7
operator, defined 39
operator, delete 42
operator, function call 26
operator, new 42
operator, subscripting 26
:: operator, use of 20
operators, additive 10
operators, arithmetic 10
operators, assignment 12, 26
operators, associativity of 6
operators, bitwise 11
operators, commutative 6
operators, equality 11
operators, list of 26
operators, multiplicative 10
operators, overloading of 26
operators, precedence of 6
operators, relational 11
operators, shift 11
operators, unary 8
order of argument evaluation 8
order of evaluation 6, 42
order of fields 3
order of evaluation of expressions 6
overflow 6
overload specifier 15
overloaded binary operators 14
overloaded function, call of 32
overloaded function name 32
overloaded function call operator 14
overloaded member functions 22
overloaded subscribing operator 14
overloaded unary operators 13
overloaded functions, conversions for 32
overloaded name, address of 32
overloaded operators 13
overloading classes 26
overloading, restriction on 26
overloading and constructors 32
overloading and conversion 32
overloading of operators 26
parameter, formal 37
parenthesized expression 7
permitted form of initializer 41
permitted on classes, operations 39
permitted on functions, operations 40
pointer arithmetic 10
pointer comparison 11
pointer conversion 10, 40
pointer, declaration of 17
pointer, NULL 13
pointer, this 7, 21
pointer-integer conversion 5, 10, 41
pointers 4
pointers, size of 3
pointer to function 40
pointer to class object, assignment to 13
pointer type, conversion to 34
pointer type, void* 13, 48
portability 41
postfix ++ and -- 9
precedence of operators 6
prefix ++ and -- 9
preprocessor 48
preprocessor, macro 38
primary expression 7
private members 24
program 47
program scope 3
program format 1
public keyword 24
public members 24
recursion 8, 4
reference, assignment to 13
reference, declaration of 17
subscripting, array explanation of 40
subscripting, example of 19
subscripting operator 26
subscripting operator, overloaded 14
subscript operator [ ] 7
subtraction operator, - 10
switch statement 35
syntax 42
syntax, class member 8
syntax notation 3
this constructors 42
this destructors 42
this pointer 7, 21
token replacement 38
type 3
type, char 4, 16
type, const 16
type, constant 16
type, conversion from user-defined 34
type declaration 17
type, double 16
type equivalence 31
type, float 16
type, int 16
type, long 4, 16
type, short 4, 16
type specifier 15
type, unsigned 4, 16
type, void 4, 6, 16
type-conversion, constructor 23
type-conversion, function argument 23
type conversion, function argument 18
type conversion rules 6
type conversion uniqueness 32
type-conversion uniqueness, user-defined 24
type-conversion user-defined 23
type conversion by return 36
typedef declaration 31
typedef, example of 31
typedef new type 31
type names 30
type of fields 3
type of string 7
type of virtual function 23
types, arithmetic 4
types, derived 4
types, floating 4
types, fundamental 4
types, integral 4
types, restrictions on 18
unary expression 8
unary operators 8
unary minus operator, - 9
unary operator, interpretation of 13
unary operators, overloaded 13
#undef 38
underscore character 1
union, anonymous 26
union, class struct 20
union constructors 26
union, field 27
union member functions 26
unions 4, 26
uniqueness, type conversion 32
uniqueness, user-defined type-conversion 24
unknown function argument types 17
unnamed fields 27
unsigned type 4, 16
unsigned-integer conversion 5
use of :: operator 20
use of ellipsis, example of 19
user-defined type, conversion from 34
user-defined, type-conversion 23
user-defined type-conversion uniqueness 24
virtual functions 22
virtual specifier 15
virtual and friend 23
virtual function, definition of 23
virtual function, example of 22
virtual function, type of 23
visibility, example of member name 25
visibility of base class members 25
visibility of member names 24
void& 17
void* pointer type 13, 48
void type 4, 6, 16
while statement 34
zero-sized fields 27