17. C LANGUAGE

Introduction

This chapter contains a summary of the grammar and syntax rules of the C Programming Language. A consistent attempt is made to point out where other implementations may differ.

Lexical Conventions

There are six classes of tokens: identifiers, keywords, constants, string literals, 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 that could possibly constitute a token.

Comments

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

Identifiers (Names)

An identifier is a sequence of letters and digits. The first character must be a letter. The underscore (_) counts as a letter. Uppercase and lowercase letters are different. There is no limit on the length of a name. Other implementations may collapse case distinctions for external names, and may reduce the number of significant characters for both external and non-external names.

Keywords

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

asm	**double**	**if**	**struct**
auto	**else**	**int**	**switch**
break	**enum**	**long**	**typedef**
case	**external**	**register**	**union**
char	**float**	**return**	**unsigned**
continue	**for**	**short**	**void**
default	**fortran**	**sizeof**	**while**
do	**goto**	**static**	

Constants

There are several kinds of constants. Each has a type; an introduction to types is given in "Storage Class and Type."

Integer Constants

An integer constant consisting of a sequence of digits is taken to be octal if it begins with **0** (digit zero). An octal constant consists of the digits **0** through **7** only. 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. Otherwise, the integer constant is taken to be decimal. A decimal constant whose value exceeds the largest signed machine integer is taken to be **long**; an octal or hex constant that exceeds the largest unsigned machine integer is likewise taken to be **long**. Otherwise, integer constants are **int**.

Explicit Long Constants

A decimal, octal, or hexadecimal integer constant immediately followed by **l** (letter ell) or **L** is a long constant. As discussed below, integer and long values may be considered identical on some computers.

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. Certain nongraphic characters, the single quote (') and the backslash (\), may be represented according to the table of escape sequences shown in Figure 17-1.

```
new-lineNL (LF)\n
horizontal tabHT\t
vertical tabVT\v
backspaceBS\b
carriage returnCR\r
form feedFF\f
backslash\\\
single quote'\'
bit patternddd\ddd
```

Figure 17-1. Escape Sequences for Nongraphic Characters

The escape *ddd* consists of the backslash followed by 1, 2, or 3 octal digits that 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 ASCII character **NUL**. If the character following a backslash is not one of those specified, the behavior is undefined. An explicit new-line character is illegal in a character constant. The type of a character constant is **int**.

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** and the exponent (not both) may be missing. Every floating constant has type **double**.

Enumeration Constants

Names declared as enumerators (see "Structure, Union, and Enumeration Declarations" under "Declarations") have type **int**.

String Literals

A string literal is a sequence of characters surrounded by double quotes, as in "**...**". A string literal has type "array of **char**" and storage class **static** (see "Storage Class and Type") and is initialized with the given characters. The compiler places a null byte (\0) at the end of each string literal so that programs that scan the string literal can find its end. In a string literal, the double quote character (") must be preceded by a \\; in addition, the same escapes as described for character constants may be used.

A \\ and the immediately following new-line are ignored. All string literals, even when written identically, are distinct.

Syntax Notation

Syntactic categories are indicated by *italic* type and literal words and characters by **bold** type. Alternative categories are listed on separate lines. An optional entry is indicated by the subscript "opt," so that:

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

indicates an optional expression enclosed in braces. The syntax is summarized in "Syntax Summary" at the end of the chapter.

Storage Class and Type

The C language bases the interpretation of an identifier on two attributes of the identifier: its storage class and its type. The storage class determines the location and lifetime of the storage associated with an identifier; the type determines the meaning of the values found in the identifier's storage.

Storage Class

There are four declarable storage classes:

- automatic
- static
- external
- register

Automatic variables are local to each invocation of a block (see "Compound Statement or Block" in "Statements") and are discarded on exit from the block. Static variables are local to a block but retain their values on reentry to a block even after control has left the block. External variables exist and retain their values throughout the execution of the entire program and may be used for communication between functions, even separately compiled functions. Register variables are (if possible) stored in the fast registers of the machine; like automatic variables, they are local to each block and disappear on exit from the block.

Type

The C language supports several fundamental types of objects. Objects declared as characters (**char**) are large enough to store any member of the implementation's character set. If a genuine character from that character set is stored in a **char** 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. In particular, **char** may be signed or unsigned by default. In this implementation the default is unsigned.

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. The sizes for computers based on the M68000 family of microprocessors are shown in Figure 17-2.

Motorola M68000 Family (ASCII)	
char	8 bits
int	32
short	16
long	32
float	32
double	64
float range	IEEE specification
double range	IEEE specification

Figure 17-2. M68000 Family-Based Computer Hardware Characteristics

The properties of **enum** types (see "Structure, Union, and Enumeration Declarations" under "Declarations") are identical to those of some integer types.

The implementation may use the range of values to determine how to allot storage.

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. **Char, int** of all sizes whether **unsigned** or not, and **enum** will collectively be called integral types. The **float** and **double** types will collectively be called floating types.

The **void** type specifies an empty set of values. It is used as the type returned by functions that generate no value.

Besides the fundamental arithmetic types, there is a conceptually infinite class of derived types constructed from the fundamental types in the following ways:

- arrays of objects of most types
- functions that return objects of a given type
- pointers to objects of a given type
- structures containing a sequence of objects of various types
- unions capable of containing any one of several objects of various types

In general these methods of constructing objects can be applied recursively.

Objects and lvalues

An object is a manipulatable region of storage. An lvalue is an expression referring to an object. An obvious example of an lvalue expression is an identifier. There are operators that yield lvalues: for example, if **E** is an expression of pointer type, then ***E** is an lvalue expression referring to the object to which **E** points. The name "lvalue" comes from the assignment expression **E1** = **E2** in which the left operand **E1** must be an lvalue expression. The discussion of each operator below indicates whether it expects lvalue operands and whether it yields an lvalue.

Operator Conversions

A number of operators may, depending on their operands, cause conversion of the value of an operand from one type to another. This part explains the result to be expected from such conversions. The conversions demanded by most ordinary operators are summarized under "Arithmetic Conversions." The summary will be supplemented as required by the discussion of each operator.

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 preserves sign. On the computer sign extension of **char** variables does not occur. It is guaranteed that a member of the standard character set is non-negative.

On machines that treat characters as signed, the characters of the ASCII set are all non-negative. 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 integer or to a **char,** it is truncated on the left. Excess bits are simply discarded.

Float and Double

All floating arithmetic in C is carried out in double precision. Whenever a **float** appears in an expression it is lengthened to **double** by zero padding its fraction. When a **double** must be converted to **float**, for example by an assignment, the **double** is rounded before truncation to **float** length. This result is undefined if it cannot be represented as a float.

Floating and Integral

Conversions of floating values to integral type are rather machine dependent. In particular, the direction of truncation of negative numbers varies. The result is undefined if it will not fit in the space provided.

Conversions of integral values to floating type behave well. Some loss of accuracy occurs if the destination lacks sufficient bits.

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 integer as specified in the discussion of the subtraction operator.

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 2^{wordsize}). In a 2's complement representation, this conversion is conceptual; there is no actual change in the bit pattern.

When an unsigned **short** 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.

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."

1. First, any operands of type **char** or **short** are converted to **int**, and any operands of type **unsigned char** or **unsigned short** are converted to **unsigned int**.
2. Then, if either operand is **double**, the other is converted to **double** and that is the type of the result.
3. Otherwise, if either operand is **unsigned long**, the other is converted to **unsigned long** and that is the type of the result.
4. Otherwise, if either operand is **long**, the other is converted to **long** and that is the type of the result.
5. Otherwise, if one operand is **long**, and the other is **unsigned int**, they are both converted to **unsigned long** and that is the type of the result.
6. Otherwise, if either operand is **unsigned**, the other is converted to **unsigned** and that is the type of the result.
7. Otherwise, both operands must be **int**, and that is the type of the result.

Void

The (nonexistent) value of a **void** object may not be used in any way, and neither explicit nor implicit conversion may be applied. Because a void expression denotes a nonexistent value, such an expression may be used only as an expression statement (see "Expression Statement" under "Statements") or as the left operand of a comma expression (see "Comma Operator" under "Expressions").

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.

Expressions and Operators

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 + (see "Additive Operators") are those expressions defined under "Primary Expressions", "Unary Operators", and "Multiplicative Operators". Within each subpart, 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 are summarized in the grammar of "Syntax Summary".

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. 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 undefined. 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.

Primary Expressions

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

> *primary-expression:*
> *identifier*
> *constant*
> *string literal*
> *(expression)*
> *primary-expression [expression]*
> *primary-expression (expression-list*$_{opt}$ *)*
> *primary-expression . identifier*
> *primary-expression -> identifier*
>
> *expression-list:*
> *expression*
> *expression-list , expression*

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 . . .", 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 that is declared "function returning . . .", when used except in the function-name position of a call, is converted to "pointer to function returning . . .".

A constant is a primary expression. Its type may be **int**, **long**, or **double** depending on its form. Character constants have type **int** and floating constants have type **double**.

A string literal 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 literal. (There is an exception in certain initializers; see "Initialization" under "Declarations.")

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 ***((E1)+(E2))**. All the clues needed to understand this notation are contained in this subpart together with the discussions in "Unary Operators" and "Additive Operators" on identifiers, * and +, respectively. The implications are summarized under "Arrays, Pointers, and Subscripting" under "Types Revisited."

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

Any actual arguments of type **float** are converted to **double** before the call. Any of type **char** or **short** are converted to **int**. Array names are converted to pointers. No other conversions are performed automatically; in particular, the compiler does not compare the types of actual arguments with those of formal arguments. If conversion is needed, use a cast; see "Unary Operators" and "Type Names" under "Declarations."

In preparing for the call to a function, a copy is made of each actual parameter. Thus, all argument passing in C is strictly by value. A function may change the values of its formal parameters, but these changes cannot affect the values of the actual parameters. It is possible to pass a pointer on the understanding that the function may change the value of the object to which the pointer points. An array name is a pointer expression. 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 is an expression. The first expression must be a structure or a union, and the identifier must name a member of the structure or union. The value is the named member of the structure or union, and it is an lvalue if the first expression is an lvalue.

A primary expression followed by an arrow (built from – and >) followed by an identifier is an expression. The first expression must be a pointer to a structure or a union and the identifier must name a member of that structure or union. The result is an lvalue referring to the named member of the structure or union to which the pointer expression points. Thus the expression **E1–>MOS** is the same as **(*E1).MOS**. Structures and unions are discussed in "Structure, Union, and Enumeration Declarations" under "Declarations."

Unary Operators

Expressions with unary operators group from right to left:

unary-expression:
- * *expression*
- *& lvalue*
- – *expression*
- *! expression*
- ~ *expression*
- ++ *lvalue*
- —*lvalue*
- *lvalue* ++
- *lvalue* —
- *(type-name) expression*
- sizeof *expression*
- sizeof *(type-name)*

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 lvalue. If the type of the lvalue is ". . . ", the type of the result is "pointer to . . .".

The result of the unary – operator is the negative of its operand. 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 the corresponding signed type.

There is no unary + operator.

The result of the logical negation operator **!** is one if the value of its operand is zero, zero if the value of its operand is nonzero. 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 object referred to by the lvalue operand of prefix ++ is incremented. The value is the new value of the operand but is not an lvalue. The expression ++**x** is equivalent to **x** += **1**. See the discussions "Additive Operators" and "Assignment Operators" for information on conversions.

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

When postfix ++ is applied to an lvalue, the result is the value of the object referred to by the lvalue. After the result is noted, the object is incremented in the same way as for the prefix ++ operator. The type of the result is the same as the type of the lvalue expression.

When postfix — is applied to an lvalue, the result is the value of the object referred to by the 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 lvalue expression.

An expression preceded by the parenthesized name of a data type causes conversion of the value of the expression to the named type. This construction is called a cast. Type names are described in "Type Names" under "Declarations."

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 construction **sizeof(*type*)** is taken to be a unit, so the expression **sizeof(*type*)–2** is the same as **(sizeof(*type*))–2.**

Multiplicative Operators

The multiplicative operators *****, **/**, and **%** group from 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. Expressions with several multiplications at the same level may be rearranged by the compiler. The binary **/** operator indicates division.

The binary **%** operator yields the remainder from the division of the first expression by the second. The operands must be integral.

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).

Additive Operators

The additive operators + and − group from 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 always 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 that 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. 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 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 **int** representing the number of objects separating the pointed-to objects. 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.

Shift Operators

The shift operators << and >> group from 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.

Relational Operators

The relational operators group from left to right:

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.

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.

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.

Bitwise Exclusive OR Operator

exclusive-or-expression:
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.

Bitwise Inclusive OR Operator

inclusive-or-expression:
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.

Logical AND Operator

logical-and-expression:
expression && expression

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

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

Logical OR Operator

logical-or-expression:
    expression || *expression*

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

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**.

Conditional Operator

conditional-expression:
    expression ? expression : expression

Conditional expressions group from right to left. The first expression is evaluated; if it is nonzero, 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. If both are structures or unions of the same type, the result has the type of the structure or union. If both pointers are 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.

Assignment Operators

There are a number of assignment operators, all of which group from 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. The two parts of a compound assignment operator are separate tokens.

assignment-expression:
  lvalue = expression
  lvalue += expression
  lvalue −= expression
  *lvalue *= expression*
  lvalue /= expression
  lvalue %= expression
  lvalue >>= expression
  lvalue <<= expression
  lvalue &= expression
  lvalue ^= expression
  lvalue |= expression

In the simple assignment with =, the value of the expression replaces that of the object referred to by the lvalue. If both operands have arithmetic type, the right operand is converted to the type of the left preparatory to the assignment. Second, both operands may be structures or unions of the same type. Finally, if the left operand is a pointer, the right operand must in general be a pointer of the same type. However, the constant 0 may be assigned to a pointer; it is guaranteed that this value will produce a null pointer distinguishable from a pointer to any object.

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 "Additive Operators." All right operands and all nonpointer left operands must have arithmetic type.

Comma Operator

comma-expression:
  expression , expression

A pair of expressions separated by a comma is evaluated from 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 from left to right. In contexts where comma is given a special meaning, e.g., in lists of actual arguments to functions (see "Primary Expressions") and lists of initializers (see "Initialization" under "Declarations"), the comma operator as described in this subpart can only appear in parentheses. For example, the expression:

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

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

Declarations

Declarations are used to specify the interpretation that C gives to each identifier; they do not necessarily reserve storage associated with the identifier. Declarations have the form:

> *declaration:*
> > *decl-specifiers declarator-list*$_{opt}$ *;*

The declarators in the declarator-list contain the identifiers being declared. The decl-specifiers consist of a sequence of type and storage class specifiers.

> *decl-specifiers:*
> > *type-specifier decl-specifiers*$_{opt}$
> > *sc-specifier decl-specifiers*$_{opt}$

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

Storage Class Specifiers

The sc-specifiers are:

> *sc-specifier:*
> > **auto**
> > **static**
> > **extern**
> > **register**
> > **typedef**

The **typedef** specifier does not reserve storage. It is called a "storage class specifier" only for syntactic convenience. See "**typedef**" for more information. The meanings of the various storage classes were discussed in "Names."

The **auto, static,** and **register** declarations also serve as definitions in that they cause an appropriate amount of storage to be reserved. In the **extern** case, there must be an external definition (see "External Definitions") for the given identifiers somewhere outside the function in which they are declared.

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. Only the first few such declarations in each function are effective. Moreover, only variables of certain types will be stored in registers. One other restriction applies to variables declared using register storage class: the address-of operator, **&**, cannot be applied to them. Smaller, faster programs can be expected if register declarations are used appropriately.

At most, one sc-specifier may be given in a declaration. If the sc-specifier is missing from a declaration, it is taken to be **auto** inside a function, **extern** outside. Exception: functions are never automatic.

Type Specifiers

The type specifiers are:

> *type-specifier:*
>   *struct-or-union-specifier*
>   *typedef-name*
>   *enum-specifier*
> *basic-type-specifier:*
>   *basic-type*
>   *basic-type basic-type-specifiers*
> *basic-type:*
>   **char**
>   **short**
>   **int**
>   **long**
>   **unsigned**
>   **float**
>   **double**
>   **void**

At most, one of the words **long** or **short** may be specified with **int**; the meaning is the same as if **int** were not mentioned. The word **long** may be specified with **float**; the meaning is the same as **double**. The word **unsigned** may be specified alone, or with **int** or any of its short or long varieties, or with **char**.

Otherwise, at most one type-specifier may be given in a declaration. In particular, adjectival use of **long**, **short**, or **unsigned** is not permitted with **typedef** names. If the type-specifier is missing from a declaration, it is taken to be **int**.

Specifiers for structures, unions, and enumerations are discussed in "Structure, Union, and Enumeration Declarations." Declarations with **typedef** names are discussed in "**typedef**."

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 "Initialization." The specifiers in the declaration indicate the type and storage class of the objects to which the declarators refer. Declarators have the syntax:

> *declarator:*
> > *identifier*
> > *(declarator)*
> > ** declarator*
> > *declarator ()*
> > *declarator [constant-expression*$_{opt}$ *]*

The grouping is the same as in expressions.

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 identifier; it is this identifier that is declared. 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 of the form:

T D1

where **T** is a type-specifier (like **int**, etc.) and **D1** is a declarator. Suppose this declaration makes the identifier have type "... **T** ," where the "... " is empty if **D1** is just a plain identifier (so that the type of **x** in "**int x**" is just **int**).

Then if **D1** has the form:

***D**

the type of the contained identifier is "... pointer to **T** ."

If **D1** has the form:

D()

then the contained identifier has the type "... function returning **T**."

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, whose type is **int**, and whose value is positive. (Constant expressions are defined precisely in "Constant Expressions.") When several "array of" specifications are adjacent, a multi-dimensional array is created; the constant expressions that specify the bounds of the arrays may be missing only for the first member of the sequence. This elision is useful 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 actually permitted. The restrictions are as follows: functions may not return arrays or functions although they may return pointers; there are no arrays of functions although there may be arrays of pointers to functions. Likewise, a structure or union may not contain a function; but it may contain a pointer to a function.

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())**. The declaration suggests (and the same construction in an expression requires) the calling of a function **fip**, and then the use of indirection through the (pointer) result to yield an integer. In the declarator

(*pfi)(), the extra parentheses are necessary (as they are also in an expression) to indicate that indirection through a pointer to a function yields a function, which is then called. This function returns an integer. As another example, the function:

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

declares an array of **float** numbers and an array of pointers to **float** numbers. Finally, the expression:

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

declares a static 3-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" and the last has type **int**.

Structure and Union Declarations

A structure is an object consisting of a sequence of named members. Each member may have any type. A union is an object that may, at a given time, contain any one of several members. Structure and union specifiers have the same form.

struct-or-union-specifier:
  struct-or-union { *struct-decl-list* }
  struct-or-union identifier { *struct-decl-list* }
  struct-or-union identifier

struct-or-union:
  struct
  union

The struct-decl-list is a sequence of declarations for the members of the structure or union:

struct-decl-list:
  struct-declaration
  struct-declaration struct-decl-list

struct-declaration:
  type-specifier struct-declarator-list ;

struct-declarator-list:
  struct-declarator
  struct-declarator , struct-declarator-list

In the usual case, a struct-declarator is just a declarator for a member of a structure or union. A structure member may also consist of a specified number of bits. Such a member is also called a field; its length, a non-negative constant expression, is set off from the field name by a colon.

> *struct-declarator:*
>   *declarator*
>   *declarator : constant-expression*
>   *: constant-expression*

Within a structure, the objects declared have addresses that increase as the declarations are read from left to right. Each non-field member of a structure begins on an addressing boundary appropriate to its type; therefore, there may be unnamed holes in a structure. Field members are packed into machine integers; they do not straddle words. A field that does not fit into the space remaining in a word is put into the next word. No field may be wider than a word. (See Figure 17-2 for sizes of basic types.)

A struct-declarator with no declarator, only a colon and a width, indicates an unnamed field useful for padding to conform to externally-imposed layouts. As a special case, a field with a width of 0 specifies alignment of the next field at an implementation dependent boundary.

The language does not restrict the types of things that are declared as fields. Moreover, even **int** fields may be considered to be unsigned. For these reasons, it is strongly recommended that fields be declared as **unsigned** where that is the intent. There are no arrays of fields, and the address-of operator, **&,** may not be applied to them, so that there are no pointers to fields.

A union may be thought of as a structure all of whose members begin at offset 0 and whose size is sufficient to contain any of its members. At most, one of the members can be stored in a union at any time.

A structure or union specifier of the second form, that is, one of:

> **struct** *identifier* { *struct-decl-list* }
> **union** *identifier* { *struct-decl-list* }

declares the identifier to be the *structure tag* (or union tag) of the structure specified by the list. A subsequent declaration may then use the third form of specifier, one of:

> **struct** *identifier*
> **union** *identifier*

Structure tags allow definition of self-referential structures. Structure tags also permit the long part of the declaration to be given once and used several times. It is illegal to declare a structure or union that contains an instance of itself, but a structure or union may contain a pointer to an instance of itself.

The third form of a structure or union specifier may be used prior to a declaration that gives the complete specification of the structure or union in situations in which the size of the structure or union is unnecessary. The size is unnecessary in two situations: when a pointer to a structure or union is being declared and when a **typedef** name is declared to be a synonym for a structure or union. This, for example, allows the declaration of a pair of structures that contain pointers to each other.

The names of members and tags do not conflict with each other or with ordinary variables. A particular name may not be used twice in the same structure, but the same name may be used in several different structures in the same scope.

A simple but important example of a structure declaration is the following binary tree structure:

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

struct tnode s, *sp;

declares **s** to be a structure of the given sort and **sp** to be a pointer to a structure of the given sort. With these declarations, the expression:

sp->count

refers to the **count** field of the structure to which **sp** points; the expression:

s.left

refers to the left subtree pointer of the structure **s**; and the expression:

s.right->tword[0]

refers to the first character of the **tword** member of the right subtree of **s**.

Enumeration Declarations

Enumeration variables and constants have integral type.

enum-specifier:
  enum { *enum-list* }
  enum *identifier* { *enum-list* }
  enum *identifier*

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

enumerator:
  identifier
  identifier = *constant-expression*

The identifiers in an enum-list are declared as constants and may appear wherever constants are required. If no enumerators with = appear, then the values of the corresponding constants begin at 0 and increase by 1 as the declaration is read from left to right. An enumerator with = gives the associated identifier the value indicated; subsequent identifiers continue the progression from the assigned value.

The names of enumerators in the same scope must all be distinct from each other and from those of ordinary variables.

The role of the identifier in the enum-specifier is entirely analogous to that of the structure tag in a struct-specifier; it names a particular enumeration. For example, the segment:

```
enum color { chartreuse, burgundy, claret=20, winedark };
...
enum color *cp, col;
...
col = claret;
cp = &col;
...
if (*cp == burgundy) ...
```

makes **color** the enumeration-tag 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}.

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* , }
>
> *initializer-list:*
> *expression*
> *initializer-list* , *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 "Constant Expressions," or expressions that 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 and previously declared variables and functions.

Static and external variables that are not initialized are guaranteed to start off as zero. Automatic and register variables that 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.

When the declared variable is an aggregate (a structure or array), the initializer consists of a brace-enclosed, comma-separated list of initializers for the members of the aggregate written in increasing subscript or member order. If the aggregate contains subaggregates, this rule applies recursively to the members of the aggregate. If there are fewer initializers in the list than there are members of the aggregate, then the aggregate is padded with zeros. It is not permitted to initialize unions or automatic aggregates.

Braces may in some cases be omitted. 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.

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

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

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

```
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:

```
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, the expression:

```
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.

Finally, the expression:

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

shows a character array whose members are initialized with a string literal. The length of the string (or size of the array) includes the terminating NUL character, \0.

Type Names

In two contexts (to specify type conversions explicitly via a cast and as an argument of **sizeof**), 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 that omits the name of the object.

type-name:
  type-specifier abstract-declarator

abstract-declarator:
  empty
  (abstract-declarator)
  * *abstract-declarator*
  abstract-declarator ()
  abstract-declarator [*constant-expression*$_{opt}$]

To avoid ambiguity, in the construction:

(abstract-declarator)

the abstract-declarator is required to be nonempty. Under this restriction, 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, the expressions:

```
int
int *
int *[3]
int (*)[3]
int *()
int (*)()
int (*[3])()
```

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

Implicit Declarations

It is not always necessary to specify both the storage class and the type of identifiers in a declaration. The storage class is supplied by the context in external definitions and in declarations of formal parameters and structure members. In a declaration inside a function, if a storage class but no type is given, the identifier is assumed to be **int**; if a type but no storage class is indicated, the identifier is assumed to be **auto**. An exception to the latter rule is made for functions because **auto** functions do not exist. If the type of an identifier is "function returning . . . ," it is implicitly declared to be **extern**.

In an expression, an identifier followed by **(** and not already declared is contextually declared to be "function returning **int.**"

typedef

Declarations whose "storage class" is **typedef** do not define storage, but instead define identifiers that 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 "Meaning of Declarators." For example, after:

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

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**," and that of **z** is the specified structure. The **zp** is a pointer to such a structure.

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

Statements

Except as indicated, statements are executed in sequence.

Expression Statement

Most statements are expression statements, which have the form:

expression ;

Expression statements are usually assignments or function calls.

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

declaration-list:
declaration
declaration declaration-list

statement-list:
statement
statement statement-list

If any of the identifiers in the declaration-list 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 the block is entered at the top. It is currently possible (but a bad practice) to transfer into a block; in that case the initializations are not performed. Initializations of **static** variables are performed only once when the program begins execution. Inside a block, **extern** declarations do not reserve storage, so initialization is not permitted.

Conditional Statement

The two forms of the conditional statement are:

> **if** (*expression*) *statement*
> **if** (*expression*) *statement* **else** *statement*

In both cases, the expression is evaluated; if it is nonzero, the first substatement is executed. In the second case, the second substatement is executed if the expression is 0. The **else** ambiguity is resolved by connecting an **else** with the last encountered **else**-less **if**.

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 nonzero. The test takes place before each execution of the statement.

do Statement

The **do** statement has the form:

> **do** *statement* **while** (*expression*) ;

The substatement is executed repeatedly until the value of the expression becomes 0. The test takes place after each execution of the statement.

for Statement

The **for** statement has the form:

> **for** ($exp\text{-}1_{opt}$; $exp\text{-}2_{opt}$; $exp\text{-}3_{opt}$) *statement*

Except for the behavior of **continue**, this statement is equivalent to:

```
exp-1 ;
while ( exp-2 )
{
        statement
        exp-3 ;
}
```

Thus the first expression specifies initialization for the loop; the second specifies a test, made before each iteration, such that the loop is exited when the expression becomes 0. The third expression often specifies an incrementing that is performed after each iteration.

Any or all the expressions may be dropped. A missing *exp-2* makes the implied **while** clause equivalent to **while(1)**; other missing expressions are simply dropped from the expansion above.

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:

switch (*expression*) *statement*

The usual arithmetic conversion is performed on the expression, but the result must be **int**. The statement is typically compound. Any statement within the statement may be labeled with one or more case prefixes as follows:

case *constant-expression* :

where the constant expression must be **int**. No two of the case constants in the same switch may have the same value. Constant expressions are precisely defined in "Constant Expressions."

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

default :

which properly goes at the end of the case constants.

When the **switch** statement is executed, its expression is evaluated and compared in turn 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 statement prefixed by **default**. If no case matches and if there is no **default**, then none of the statements in the switch is executed.

The prefixes **case** and **default** do not alter the flow of control, which continues unimpeded across such prefixes. That is, once a case constant is matched, all **case** statements (and the **default**) from there to the end of the **switch** are executed. To exit from a switch, see "**break** Statement."

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. A simple example of a complete **switch** statement is:

```
switch (c) {
            case 'o':
                          oflag = TRUE;
                          break;
            case 'p':
                          pflag = TRUE;
                          break;
            case 'r':
                          rflag = TRUE;
                          break;
            default :
                          (void) fprintf(stderr, "Unknown option\n");
                          exit(2);
            }
```

break Statement

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

continue Statement

The statement **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 below, a **continue** is equivalent to **goto contin:**

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

(Following the **contin:** is a null statement; see "Null Statement.")

return Statement

A function returns to its caller by means of the **return** statement, which has one of the forms below:

return ;
return *expression* ;

In the first case, the returned value is undefined. In the second case, the value of the expression is returned to the caller of the function. If required, the expression is converted, as if by assignment, to the type of function in which it appears. Flowing off the end of a function is equivalent to a return with no returned value.

goto Statement

Control may be transferred unconditionally by means of the statement:

goto *identifier* ;

The identifier must be a label (see "Labeled Statement") located in the current function.

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 subblocks in which the same identifier has been redeclared. See "Scope Rules."

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**.

External Definitions

A C program consists of a sequence of external definitions. An external definition declares an identifier to have storage class **extern** (by default) or perhaps **static**, and a specified type. The type-specifier (see "Type Specifiers" in "Declarations") may also be empty, in which case the type is taken to be **int**. The scope of external definitions persists to the end of the file in which they are declared just as the effect of declarations persists to the end of a block. The syntax of external definitions is the same as that of all declarations except that only at this level may the code for functions be given.

External Function Definitions

Function definitions have the form:

> *function-definition:*
> *decl-specifiers$_{opt}$ function-declarator function-body*

The only sc-specifiers allowed among the decl-specifiers are **extern** or **static**; see "Scope of Externals" in "Scope Rules" for the distinction between them. A function declarator is similar to a declarator for a "function returning . . . " except that it lists the formal parameters of the function being defined.

> *function-declarator:*
> *declarator (parameter-list$_{opt}$)*
>
> *parameter-list:*
> *identifier*
> *identifier , parameter-list*

The function-body has the form:

> *function-body:*
> *declaration-list$_{opt}$ compound-statement*

The identifiers in the parameter list, and only those identifiers, may be declared in the declaration list. Any identifiers whose type is not given are taken to be **int**. The only storage class that may be specified is **register**; if it is specified, the corresponding actual parameter will be copied, if possible, into a register at the outset of the function.

A simple example of a complete function definition is:

```
int max(a, b, c)
        int a, b, c;
{
        int m;

        m = (a > b) ? a : b;
        return((m > c) ? m : c);
}
```

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

The C program converts all **float** actual parameters to **double**, so formal parameters declared **float** have their declaration adjusted to read **double**. All **char** and **short** formal parameter declarations are similarly adjusted to read **int**.

Also, since a reference to an array in any context (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 . . . ."

External Data Definitions

An external data definition has the form:

data-definition:
    declaration

The storage class of such data may be **extern** (which is the default) or **static**, but not **auto** or **register**.

Scope Rules

A C program need not all be compiled at the same time. The source text of the program may be kept in several files, and precompiled routines may be loaded from libraries. Communication among the functions of a program may be carried out both through explicit calls and through manipulation of external data.

Therefore, there are two kinds of scopes to consider: first, what may be called the lexical scope of an identifier, which is essentially the region of a program during which it may be used without drawing "undefined identifier" diagnostics; and second, the scope associated with external identifiers, which is characterized by the rule that references to the same external identifier are references to the same object.

Lexical Scope

The lexical scope of identifiers declared in external definitions persists from the definition through the end of the source file in which they appear. The lexical scope of identifiers that are formal parameters persists through the function with which they are associated. The lexical scope of identifiers declared at the head of a block persists until the end of the block. The lexical scope of labels is the whole of the function in which they appear.

In all cases, however, if an identifier is explicitly declared at the head of a block, including the block constituting a function, any declaration of that identifier outside the block is suspended until the end of the block.

Remember also (see the "Structure and Union" and "Enumeration" sections under "Declarations") that tags, identifiers associated with ordinary variables, and identities associated with structure and union members form three distinct classes which do not conflict. Members and tags follow the same scope rules as other identifiers. The **enum** constants are in the same class as ordinary variables and follow the same scope rules. The **typedef** names are in the same class as ordinary identifiers. They may be redeclared in inner blocks, but an explicit type must be given in the inner declaration:

```
typedef float distance;
...
{
     int distance;
     ...
```

The second declaration must contain the **int**, or it would be taken as a declaration with no declarators and type **distance**.

Scope of Externals

If a function refers to an identifier declared to be **extern**, then somewhere among the files or libraries constituting the complete program there must be at least one external definition for the identifier. All functions in a given program that refer to the same external identifier refer to the same object, so care must be taken that the type and size specified in the definition are compatible with those specified by each function that references the data.

It is illegal to explicitly initialize any external identifier more than once in the set of files and libraries comprising a multi-file program. It is legal to have more than one data definition for any external non-function identifier; explicit use of **extern** does not change the meaning of an external declaration.

In restricted environments, the use of the **extern** storage class takes on an additional meaning. In these environments, the explicit appearance of the **extern** keyword in external data declarations of identities without initialization indicates that the storage for the identifiers is allocated elsewhere, either in this file or another file. It is required that there be exactly one definition of each external identifier (without **extern**) in the set of files and libraries comprising a mult-file program.

Identifiers declared **static** at the top level in external definitions are not visible in other files. Functions may be declared **static**.

Compiler Control Lines

The C compilation system contains a preprocessor capable of macro substitution, conditional compilation, and inclusion of named files. Lines beginning with # communicate with this preprocessor. There may be any number of blanks and horizontal tabs between the # and the directive, but no additional material (such as comments) is permitted. These lines have syntax independent of the rest of the language; they may appear anywhere and have effect that lasts (independent of scope) until the end of the source program file.

Token Replacement

A control line of the form:

#define *identifier token-string*$_{opt}$

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

#define *identifier(identifier, ...) token-string*$_{opt}$

where there is no space between the first identifier and the **(**, is a macro definition with arguments. There may be zero or more formal parameters. 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 to replace.

In both forms the replacement string is rescanned for more defined identifiers. In both forms a long definition may be continued on another line by writing \ at the end of the line to be continued. This facility is most valuable for definition of "manifest constants," as in:

```
#define TABSIZE 100

int table[TABSIZE];
```

A control line of the form:

> **#undef** *identifier*

causes the identifier's preprocessor definition (if any) to be forgotten.

If a **#define**d identifier is the subject of a subsequent **#define** with no intervening **#undef**, then the two token-strings are compared textually. If the two token-strings are not identical (all white space is considered as equivalent), then the identifier is considered to be redefined.

File Inclusion

A control line of the form:

> **#include** "*filename*"

causes the replacement of that line by the entire contents of the file *filename*. The named file is searched for first in the directory of the file containing the **#include**, 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 **#include**. (How the places are specified is not part of the language. See **cpp**(1) for a description of how to specify additional libraries.)

The **#include**s may be nested.

Conditional Compilation

A compiler control line of the form:

> **#if** *restricted-constant-expression*

checks whether the restricted-constant expression evaluates to nonzero. (Constant expressions are discussed in "Constant Expressions"; the following additional restrictions apply here: the constant expression may not contain **sizeof**, casts, or an enumeration constant.)

A restricted-constant expression may also contain the additional unary expression:

defined *identifier*

or:

defined (*identifier***)**

which evaluates to 1 if the identifier is currently defined in the preprocessor and to 0 if it is not.

All currently defined identifiers in restricted-constant-expressions are replaced by their token-strings (except those identifiers modified by **defined**) just as in normal text. The restricted-constant expression will be evaluated only after all expressions have finished. During this evaluation, all undefined (to the procedure) identifiers evaluate to zero.

A control line of the form:

#ifdef *identifier*

checks whether the identifier is currently defined in the preprocessor; i.e., whether it has been the subject of a **#define** control line. It is equivalent to **#if defined (***identifier***)**.

A control line of the form:

#ifndef *identifier*

checks whether the identifier is currently undefined in the preprocessor. It is equivalent to **#if !defined (***identifier***)**.

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

#else

and then by a control line:

#endif

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

Another control directive is:

#elif *restricted-constant-expression*

An arbitrary number of **#elif** directives can be included between **#if**, **#ifdef**, or **#ifndef** and **#else**, or **#endif** directives. These constructions may be nested.

Line Control

For the benefit of other preprocessors that 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 *"filename"*. If *"filename"* is absent, the remembered file name does not change.

Version Control

This capability, known as *S-lists,* helps administer version control information. A line of the form:

> **#ident** *"version"*

puts any arbitrary string in the **.comment** section of the **a.out** file. It is usually used for version control. It is worth remembering that **.comment** sections are not loaded into memory when the **a.out** file is executed.

Types Revisited

This part summarizes the operations that can be performed on objects of certain types.

Structures and Unions

Structures and unions may be assigned, passed as arguments to functions, and returned by functions. Other plausible operators, such as equality comparison and structure casts, are not implemented.

In a reference to a structure or union member, the name on the right of the -> or the . must specify a member of the aggregate named or pointed to by the expression on the left. In general, a member of a union may not be inspected unless the value of the union has been assigned using that same member. However, one special guarantee is made by the language in order to simplify the use of unions: if a union contains several structures that share a common initial sequence and if the union currently contains one of these structures, it is permitted to inspect the common initial part of any of the contained structures.

For example, the following is a legal fragment:

```
union
{
        struct
        {
                int         type;
        } n;
        struct
        {
                int         type;
                int         intnode;
        } ni;
        struct
        {
                int         type;
                float       floatnode;
        } nf;
} u;
...
u.nf.type = FLOAT;
u.nf.floatnode = 3.14;
...
if (u.n.type == FLOAT)
        ... sin(u.nf.floatnode) ...
```

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:

```
int f();
...
g(f);
```

Then the definition of **g** might read:

```
g(funcp)
        int (*funcp)();
{
        ...
        (*funcp)();
        ...
}
```

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

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 that apply to +, if **E1** is an array and **E2** an integer, then **E1[E2]** refers to the **E2 -th** member of **E1**. Therefore, despite its asymmetric appearance, subscripting is a commutative operation.

A consistent rule is followed in the case of multidimensional 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 **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 of the object to which the pointer points, namely five-integer objects. The results are added and indirection applied to yield an array (of five integers) which in turn is converted to a pointer to the first of the integers. If there is another subscript, the same argument applies again; this time the result is an integer.

Arrays in C are stored row-wise (last subscript varies fastest) and the first subscript in the declaration helps determine the amount of storage consumed by an array. Arrays play no other part in subscript calculations.

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 (see "Unary Operators" under "Expressions" and "Type Names" under "Declarations").

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.

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 when used 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.

For example, a storage-allocation routine might accept a size (in bytes) of an object to allocate, and return a **char** pointer; it might be used in this way:

```
extern char *alloc();
double *dp;

dp = (double *) alloc(sizeof(double));
*dp = 22.0 / 7.0;
```

The **alloc** must ensure (in a machine-dependent way) that its return value is suitable for conversion to a pointer to **double**; then the use of the function is portable.

Constant Expressions

In several places, C requires expressions that evaluate to a constant: after **case**, as array bounds, and in initializers. In the first two cases, the expression can involve only integer constants, character constants, casts to integral types, enumeration constants, 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 initializers. Besides constant expressions as discussed above, one can also use floating constants and arbitrary casts and can also apply the unary **&** operator to external or static objects and 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.

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 or integer division have proven 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 nuisances 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.

The order of evaluation of function arguments is not specified by the language. The order in which side effects take place is also unspecified.

Since character constants are really objects of type **int**, multicharacter character constants may be permitted. The specific implementation is very machine-dependent because the order in which characters are assigned to a word varies from one machine to another.

Fields are assigned to words and characters to integers right to left on some machines and left to right on other machines. These differences are invisible to isolated programs that do not indulge in type punning (e.g., 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.

Syntax Summary

This summary of C syntax is intended more for aiding comprehension than as an exact statement of the language.

Expressions

The basic expressions are:

expression:
- *primary*
- * *expression*
- & *lvalue*
- – *expression*
- ! *expression*
- ~ *expression*
- ++ *lvalue*
- — *lvalue*
- *lvalue* ++
- *lvalue* —
- **sizeof** *expression*
- **sizeof** (*type-name*)
- (*type-name*) *expression*
- *expression binop expression*
- *expression ? expression : expression*
- *lvalue asgnop expression*
- *expression , expression*

```
primary:
        identifier
        constant
        string literal
        ( expression )
        primary ( expression-list_opt )
        primary [ expression ]
        primary . identifier
        primary -> identifier

lvalue:
        identifier
        primary [ expression ]
        lvalue . identifier
        primary -> identifier
        * expression
        ( lvalue )
```

The primary-expression operators:

```
() [] . ->
```

have highest priority and group from left to right. The unary operators:

```
* & - ! ~ ++ -- sizeof ( type-name )
```

have priority below the primary operators but higher than any binary operator. They group from right to left. Binary operators group from left to right; they have priority decreasing as indicated below.

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

The conditional operator groups from right to left.

Assignment operators all have the same priority and all group from right to left.

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

The comma operator has the lowest priority and groups from left to right.

Declarations

declaration:
    decl-specifiers init-declarator-list$_{opt}$;

decl-specifiers:
    type-specifier decl-specifiers$_{opt}$
    sc-specifier decl-specifiers$_{opt}$

sc-specifier:
    auto
    static
    extern
    register
    typedef

type-specifier:
    struct-or-union-specifier
    typedef-name
    enum-specifier

basic-type-specifier:
    basic-type
    basic-type basic-type-specifiers

basic-type:
    char
    short
    int
    long
    unsigned
    float
    double
    void

enum-specifier:
    enum { *enum-list* }
    enum *identifier* { *enum-list* }
    enum *identifier*

enum-list:
  enumerator
  enum-list , enumerator

enumerator:
  identifier
  identifier = constant-expression

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

init-declarator:
  declarator $initializer_{opt}$

declarator:
  identifier
  (declarator)
  ** declarator*
  declarator ()
  declarator [$constant\text{-}expression_{opt}$ *]*

struct-or-union-specifier:
  struct { *struct-decl-list* }
  struct *identifier* { *struct-decl-list* }
  struct *identifier*
  union { *struct-decl-list* }
  union *identifier* { *struct-decl-list* }
  union *identifier*

struct-decl-list:
  struct-declaration
  struct-declaration struct-decl-list

struct-declaration:
  type-specifier struct-declarator-list ;

struct-declarator-list:
  struct-declarator
  struct-declarator , struct-declarator-list

struct-declarator:
  declarator
  declarator : constant-expression
  : constant-expression

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

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

type-name:
  type-specifier abstract-declarator

abstract-declarator:
  empty
  (abstract-declarator)
  ** abstract-declarator*
  abstract-declarator ()
  abstract-declarator [constant-expression$_{opt}$ *]*

typedef-name:
  identifier

Statements

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

declaration-list:
  declaration
  declaration declaration-list

statement-list:
  statement
  statement statement-list

statement:
- *compound-statement*
- *expression ;*
- **if** *(expression) statement*
- **if** *(expression) statement* **else** *statement*
- **while** *(expression) statement*
- **do** *statement* **while** *(expression) ;*
- **for** *(*exp_{opt}*;*exp_{opt}*;*exp_{opt}*) statement*
- **switch** *(expression) statement*
- **case** *constant-expression : statement*
- **default** *: statement*
- **break ;**
- **continue ;**
- **return ;**
- **return** *expression ;*
- **goto** *identifier ;*
- *identifier : statement*
- *;*

External Definitions

program:
- *external-definition*
- *external-definition program*

external-definition:
- *function-definition*
- *data-definition*

function-definition:
- *decl-specifier*$_{opt}$ *function-declarator function-body*

function-declarator:
- *declarator (parameter-list*$_{opt}$ *)*

parameter-list:
- *identifier*
- *identifier , parameter-list*

function-body:
- *declaration-list*$_{opt}$ *compound-statement*

data-definition:
  extern *declaration* **;**
  static *declaration* **;**

Preprocessor

#define *identifier token-string*$_{opt}$
#define *identifier(identifier,...)token-string*$_{opt}$
#undef *identifier*
#include "*filename*"
#include <*filename*>
#if *restricted-constant-expression*
#ifdef *identifier*
#ifndef *identifier*
#elif *restricted-constant-expression*
#else
#endif
#line *constant* "*filename*"
#ident "*version*"

17