Appendix A - Reference Manual

A.1 Introduction

This manual describes the C language specified by the draft submitted to ANSI on 31 October, 1988, for approval as ``American Standard for Information Systems - programming Language C, X3.159-1989." The manual is an interpretation of the proposed standard, not the standard itself, although care has been taken to make it a reliable guide to the language.

For the most part, this document follows the broad outline of the standard, which in turn follows that of the first edition of this book, although the organization differs in detail. Except for renaming a few productions, and not formalizing the definitions of the lexical tokens or the preprocessor, the grammar given here for the language proper is equivalent to that of the standard.

> Throughout this manual, commentary material is indented and written in smaller type, as this is. Most often these comments highlight ways in which ANSI Standard C differs from the language defined by the first edition of this book, or from refinements subsequently introduced in various compilers.

A.2 Lexical Conventions

A program consists of one or more *translation units* stored in files. It is translated in several phases, which are described in Par.A.12. The first phases do low-level lexical transformations, carry out directives introduced by the lines beginning with the # character, and perform macro definition and expansion. When the preprocessing of Par.A.12 is complete, the program has been reduced to a sequence of tokens.

A.2.1 Tokens

There are six classes of tokens: identifiers, keywords, constants, string literals, operators, and other separators. Blanks, horizontal and vertical tabs, newlines, formfeeds and comments as described below (collectively, ``white space") are ignored except as they separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

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

A.2.2 Comments

The characters `/*` introduce a comment, which terminates with the characters `*/`. Comments do not nest, and they do not occur within a string or character literals.

A.2.3 Identifiers

An identifier is a 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. Identifiers may have any length, and for internal identifiers, at least the first 31 characters are significant; some implementations may take more characters significant. Internal identifiers include preprocessor macro names and all other names that do not have external linkage (Par.A.11.2). Identifiers with external linkage are more restricted: implementations may make as few as the first six characters significant, and may ignore case distinctions.

A.2.4 Keywords

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

```
auto        double      int         struct
break       else        long        switch
```

```
case            enum        register     typedef
char            extern      return       union
const           float       short        unsigned
continue        for         signed       void
default         goto        sizeof       volatile
do              if          static       while
```

Some implementations also reserve the words `fortran` and `asm`.

> The keywords `const`, `signed`, and `volatile` are new with the ANSI standard; `enum` and `void` are new since the first edition, but in common use; `entry`, formerly reserved but never used, is no longer reserved.

A.2.5 Constants

There are several kinds of constants. Each has a data type; Par.A.4.2 discusses the basic types:

constant:
  integer-constant
  character-constant
  floating-constant
  enumeration-constant

A.2.5.1 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. Octal constants do not contain the digits `8` or `9`. 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`.

An integer constant may be suffixed by the letter `u` or `U`, to specify that it is unsigned. It may also be suffixed by the letter `l` or `L` to specify that it is long.

The type of an integer constant depends on its form, value and suffix. (See Par.A.4 for a discussion of types). If it is unsuffixed and decimal, it has the first of these types in which its value can be represented: `int`, `long int`, `unsigned long int`. If it is unsuffixed, octal or hexadecimal, it has the first possible of these types: `int`, `unsigned int`, `long int`, `unsigned long int`. If it is suffixed by `u` or `U`, then `unsigned int`, `unsigned long int`. If it is suffixed by `l` or `L`, then `long int`, `unsigned long int`. If an integer constant is suffixed by `UL`, it is `unsigned long`.

> The elaboration of the types of integer constants goes considerably beyond the first edition, which merely caused large integer constants to be `long`. The `U` suffixes are new.

A.2.5.2 Character Constants

A character constant is a sequence of one or more characters enclosed in single quotes as in `'x'`. The value of a character constant with only one character is the numeric value of the character in the machine's character set at execution time. The value of a multi-character constant is implementation-defined.

Character constants do not contain the `'` character or newlines; in order to represent them, and certain other characters, the following escape sequences may be used:

newline	NL (LF)	`\n`	backslash	`\`	`\\`
horizontal tab	HT	`\t`	question mark	`?`	`\?`
vertical tab	VT	`\v`	single quote	`'`	`\'`
backspace	BS	`\b`	double quote	`"`	`\"`
carriage return	CR	`\r`	octal number	`ooo`	`\ooo`
formfeed	FF	`\f`	hex number	`hh`	`\xhh`
audible alert	BEL	`\a`			

The escape `\ooo` consists of the backslash followed by 1, 2, or 3 octal digits, which are taken to specify the value of the desired character. A common example of this construction is `\0` (not followed by a digit), which specifies the character NUL. The escape `\xhh` consists of the backslash, followed by `x`, followed by hexadecimal digits, which are taken to specify the value of the desired character. There is no limit on the number of digits, but the behavior is undefined if the resulting character value exceeds that of the largest character. For either octal or hexadecimal escape characters, if the implementation treats the `char` type as signed, the value is sign-extended as if cast to `char` type. If the character following the `\` is not one of those specified, the behavior is undefined.

In some implementations, there is an extended set of characters that cannot be represented in the `char` type. A constant in this extended set is written with a preceding `L`, for example `L'x'`, and is called a wide character constant. Such a constant has type `wchar_t`, an integral type defined in the standard header `<stddef.h>`. As with ordinary character constants, hexadecimal escapes may be used; the effect is undefined if the specified value exceeds that representable with `wchar_t`.

> Some of these escape sequences are new, in particular the hexadecimal character representation. Extended characters are also new. The character sets commonly used in the Americas and western Europe can be encoded to fit in the `char` type; the main intent in adding `wchar_t` was to accommodate Asian languages.

A.2.5.3 Floating Constants

A floating constant consists of an integer part, a decimal part, a fraction part, an `e` or `E`, an optionally signed integer exponent and an optional type suffix, one of `f`, `F`, `l`, or `L`. 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. The type is determined by the suffix; `F` or `f` makes it `float`, `L` or `l` makes it `long double`, otherwise it is `double`.

A2.5.4 Enumeration Constants

Identifiers declared as enumerators (see Par.A.8.4) are constants of type `int`.

A.2.6 String Literals

A string literal, also called a string constant, is a sequence of characters surrounded by double quotes as in `"..."`. A string has type ``array of characters" and storage class `static` (see Par.A.3 below) and is initialized with the given characters. Whether identical string literals are distinct is implementation-defined, and the behavior of a program that attempts to alter a string literal is undefined.

Adjacent string literals are concatenated into a single string. After any concatenation, a null byte `\0` is appended to the string so that programs that scan the string can find its end. String literals do not contain newline or double-quote characters; in order to represent them, the same escape sequences as for character constants are available.

As with character constants, string literals in an extended character set are written with a preceding `L`, as in `L"..."`. Wide-character string literals have type ``array of `wchar_t`." Concatenation of ordinary and wide string literals is undefined.

> The specification that string literals need not be distinct, and the prohibition against modifying them, are new in the ANSI standard, as is the concatenation of adjacent string literals. Wide-character string literals are new.

A.3 Syntax Notation

In the syntax notation used in this manual, syntactic categories are indicated by *italic* type, and literal words and characters in `typewriter` style. Alternative categories are usually listed on separate lines; in a few cases, a long set of narrow alternatives is presented on one line, marked

by the phrase ``one of." An optional terminal or nonterminal symbol carries the subscript ``*opt*," so that, for example,

$$\{\ expression_{opt}\ \}$$

means an optional expression, enclosed in braces. The syntax is summarized in Par.A.13.

> Unlike the grammar given in the first edition of this book, the one given here makes precedence and associativity of expression operators explicit.

A.4 Meaning of Identifiers

Identifiers, or names, refer to a variety of things: functions; tags of structures, unions, and enumerations; members of structures or unions; enumeration constants; typedef names; and objects. An object, sometimes called a variable, is a location in storage, and its interpretation depends on two main attributes: its *storage class* and its *type*. The storage class determines the lifetime of the storage associated with the identified object; the type determines the meaning of the values found in the identified object. A name also has a scope, which is the region of the program in which it is known, and a linkage, which determines whether the same name in another scope refers to the same object or function. Scope and linkage are discussed in Par.A.11.

A.4.1 Storage Class

There are two storage classes: automatic and static. Several keywords, together with the context of an object's declaration, specify its storage class. Automatic objects are local to a block (Par.9.3), and are discarded on exit from the block. Declarations within a block create automatic objects if no storage class specification is mentioned, or if the `auto` specifier is used. Objects declared `register` are automatic, and are (if possible) stored in fast registers of the machine.

Static objects may be local to a block or external to all blocks, but in either case retain their values across exit from and reentry to functions and blocks. Within a block, including a block that provides the code for a function, static objects are declared with the keyword `static`. The objects declared outside all blocks, at the same level as function definitions, are always static. They may be made local to a particular translation unit by use of the `static` keyword; this gives them *internal linkage*. They become global to an entire program by omitting an explicit storage class, or by using the keyword `extern`; this gives them *external linkage*.

A.4.2 Basic Types

There are several fundamental types. The standard header `<limits.h>` described in Appendix B defines the largest and smallest values of each type in the local implementation. The numbers given in Appendix B show the smallest acceptable magnitudes.

Objects declared as characters (`char`) are large enough to store any member of the execution character set. If a genuine character from that set is stored in a `char` object, its value is equivalent to the integer code for the character, and is non-negative. Other quantities may be stored into `char` variables, but the available range of values, and especially whether the value is signed, is implementation-dependent.

Unsigned characters declared `unsigned char` consume the same amount of space as plain characters, but always appear non-negative; explicitly signed characters declared `signed char` likewise take the same space as plain characters.

> unsigned `char` type does not appear in the first edition of this book, but is in common use. `signed char` is new.

Besides the `char` types, up to three sizes of integer, declared `short int`, `int`, and `long int`, are available. Plain `int` objects have the natural size suggested by the host machine

architecture; the other sizes are provided to meet special needs. Longer integers provide at least as much storage as shorter ones, but the implementation may make plain integers equivalent to either short integers, or long integers. The `int` types all represent signed values unless specified otherwise.

Unsigned integers, declared using the keyword `unsigned`, obey the laws of arithmetic modulo 2^n where *n* is the number of bits in the representation, and thus arithmetic on unsigned quantities can never overflow. The set of non-negative values that can be stored in a signed object is a subset of the values that can be stored in the corresponding unsigned object, and the representation for the overlapping values is the same.

Any of single precision floating point (`float`), double precision floating point (`double`), and extra precision floating point (`long double`) may be synonymous, but the ones later in the list are at least as precise as those before.

> `long double` is new. The first edition made `long float` equivalent to `double`; the locution has been withdrawn.

Enumerations are unique types that have integral values; associated with each enumeration is a set of named constants (Par.A.8.4). Enumerations behave like integers, but it is common for a compiler to issue a warning when an object of a particular enumeration is assigned something other than one of its constants, or an expression of its type.

Because objects of these types can be interpreted as numbers, they will be referred to as *arithmetic* types. Types `char`, and `int` of all sizes, each with or without sign, and also enumeration types, will collectively be called *integral* types. The types `float`, `double`, and `long double` will 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.

A.4.3 Derived types

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

arrays of objects of a given type;
functions returning 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 of one of several objects of various types.

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

A.4.4 Type Qualifiers

An object's type may have additional qualifiers. Declaring an object `const` announces that its value will not be changed; declaring it `volatile` announces that it has special properties relevant to optimization. Neither qualifier affects the range of values or arithmetic properties of the object. Qualifiers are discussed in Par.A.8.2.

A.5 Objects and Lvalues

An *Object* is a named region of storage; an *lvalue* is an expression referring to an object. An obvious example of an lvalue expression is an identifier with suitable type and storage class. There are operators that yield lvalues, 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 specifies whether it expects lvalue operands and whether it yields an lvalue.

A.6 Conversions

Some 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. Par.6.5 summarizes the conversions demanded by most ordinary operators; it will be supplemented as required by the discussion of each operator.

A.6.1 Integral Promotion

A character, a short integer, or an integer bit-field, all either signed or not, or an object of enumeration type, may be used in an expression wherever an integer may be used. If an `int` can represent all the values of the original type, then the value is converted to `int`; otherwise the value is converted to `unsigned int`. This process is called *integral promotion*.

A.6.2 Integral Conversions

Any integer is converted to a given unsigned type by finding the smallest non-negative value that is congruent to that integer, modulo one more than the largest value that can be represented in the unsigned type. In a two's complement representation, this is equivalent to left-truncation if the bit pattern of the unsigned type is narrower, and to zero-filling unsigned values and sign-extending signed values if the unsigned type is wider.

When any integer is converted to a signed type, the value is unchanged if it can be represented in the new type and is implementation-defined otherwise.

A.6.3 Integer and Floating

When a value of floating type is converted to integral type, the fractional part is discarded; if the resulting value cannot be represented in the integral type, the behavior is undefined. In particular, the result of converting negative floating values to unsigned integral types is not specified.

When a value of integral type is converted to floating, and the value is in the representable range but is not exactly representable, then the result may be either the next higher or next lower representable value. If the result is out of range, the behavior is undefined.

A.6.4 Floating Types

When a less precise floating value is converted to an equally or more precise floating type, the value is unchanged. When a more precise floating value is converted to a less precise floating type, and the value is within representable range, the result may be either the next higher or the next lower representable value. If the result is out of range, the behavior is undefined.

A.6.5 Arithmetic Conversions

Many operators cause conversions and yield result types in a similar way. The effect is to bring operands into a common type, which is also the type of the result. This pattern is called the *usual arithmetic conversions.*

- First, if either operand is `long double`, the other is converted to `long double`.
- Otherwise, if either operand is `double`, the other is converted to `double`.
- Otherwise, if either operand is `float`, the other is converted to `float`.
- Otherwise, the integral promotions are performed on both operands; then, if either operand is `unsigned long int`, the other is converted to `unsigned long int`.

- Otherwise, if one operand is `long int` and the other is `unsigned int`, the effect depends on whether a `long int` can represent all values of an `unsigned int`; if so, the `unsigned int` operand is converted to `long int`; if not, both are converted to `unsigned long int`.

- Otherwise, if one operand is `long int`, the other is converted to `long int`.

- Otherwise, if either operand is `unsigned int`, the other is converted to `unsigned int`.

- Otherwise, both operands have type `int`.

  There are two changes here. First, arithmetic on `float` operands may be done in single precision, rather than double; the first edition specified that all floating arithmetic was double precision. Second, shorter unsigned types, when combined with a larger signed type, do not propagate the unsigned property to the result type; in the first edition, the unsigned always dominated. The new rules are slightly more complicated, but reduce somewhat the surprises that may occur when an unsigned quantity meets signed. Unexpected results may still occur when an unsigned expression is compared to a signed expression of the same size.

A.6.6 Pointers and Integers

An expression of integral type may be added to or subtracted from a pointer; in such a case the integral expression is converted as specified in the discussion of the addition operator (Par.A.7.7).

Two pointers to objects of the same type, in the same array, may be subtracted; the result is converted to an integer as specified in the discussion of the subtraction operator (Par.A.7.7).

An integral constant expression with value 0, or such an expression cast to type `void *`, may be converted, by a cast, by assignment, or by comparison, to a pointer of any type. This produces a null pointer that is equal to another null pointer of the same type, but unequal to any pointer to a function or object.

Certain other conversions involving pointers are permitted, but have implementation-defined aspects. They must be specified by an explicit type-conversion operator, or cast (Pars.A.7.5 and A.8.8).

A pointer may be converted to an integral type large enough to hold it; the required size is implementation-dependent. The mapping function is also implementation-dependent.

A pointer to one type may be converted to a pointer to another type. The resulting pointer may cause addressing exceptions if the subject pointer does not refer to an object suitably aligned in storage. It is guaranteed that a pointer to an object may be converted to a pointer to an object whose type requires less or equally strict storage alignment and back again without change; the notion of ``alignment'' is implementation-dependent, but objects of the `char` types have least strict alignment requirements. As described in Par.A.6.8, a pointer may also be converted to type `void *` and back again without change.

A pointer may be converted to another pointer whose type is the same except for the addition or removal of qualifiers (Pars.A.4.4, A.8.2) of the object type to which the pointer refers. If qualifiers are added, the new pointer is equivalent to the old except for restrictions implied by the new qualifiers. If qualifiers are removed, operations on the underlying object remain subject to the qualifiers in its actual declaration.

Finally, a pointer to a function may be converted to a pointer to another function type. Calling the function specified by the converted pointer is implementation-dependent; however, if the converted pointer is reconverted to its original type, the result is identical to the original pointer.

A.6.7 Void

The (nonexistent) value of a `void` object may not be used in any way, and neither explicit nor implicit conversion to any non-void type may be applied. Because a void expression denotes a nonexistent value, such an expression may be used only where the value is not required, for example as an expression statement (Par.A.9.2) or as the left operand of a comma operator (Par.A.7.18).

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

> `void` did not appear in the first edition of this book, but has become common since.

A.6.8 Pointers to Void

Any pointer to an object may be converted to type `void *` without loss of information. If the result is converted back to the original pointer type, the original pointer is recovered. Unlike the pointer-to-pointer conversions discussed in Par.A.6.6, which generally require an explicit cast, pointers may be assigned to and from pointers of type `void *`, and may be compared with them.

> This interpretation of `void *` pointers is new; previously, `char *` pointers played the role of generic pointer. The ANSI standard specifically blesses the meeting of `void *` pointers with object pointers in assignments and relationals, while requiring explicit casts for other pointer mixtures.

A.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 `+` (Par.A.7.7) are those expressions defined in Pars.A.7.1-A.7.6. Within each subsection, the operators have the same precedence. Left- or right-associativity is specified in each subsection for the operators discussed therein. The grammar given in Par.13 incorporates the precedence and associativity of the operators.

The precedence and associativity of operators is fully specified, but the order of evaluation of expressions is, with certain exceptions, undefined, even if the subexpressions involve side effects. That is, unless the definition of the operator guarantees that its operands are evaluated in a particular order, the implementation is free to evaluate operands in any order, or even to interleave their evaluation. However, each operator combines the values produced by its operands in a way compatible with the parsing of the expression in which it appears.

> This rule revokes the previous freedom to reorder expressions with operators that are mathematically commutative and associative, but can fail to be computationally associative. The change affects only floating-point computations near the limits of their accuracy, and situations where overflow is possible.

The handling of overflow, divide check, and other exceptions in expression evaluation is not defined by the language. Most existing implementations of C ignore overflow in evaluation of signed integral expressions and assignments, but this behavior is not guaranteed. Treatment of division by 0, and all floating-point exceptions, varies among implementations; sometimes it is adjustable by a non-standard library function.

A.7.1 Pointer Conversion

If the type of an expression or subexpression is ``array of *T*,'' for some type *T*, then the value of the expression is a pointer to the first object in the array, and the type of the expression is altered to ``pointer to *T*.'' This conversion does not take place if the expression is in the operand of the unary `&` operator, or of `++`, `--`, `sizeof`, or as the left operand of an assignment operator or the `.` operator. Similarly, an expression of type ``function returning *T*,'' except when used as the operand of the `&` operator, is converted to ``pointer to function returning *T*.''

A.7.2 Primary Expressions

Primary expressions are identifiers, constants, strings, or expressions in parentheses.

primary-expression
identifier
constant
string
(expression)

An identifier is a primary expression, provided it has been suitably declared as discussed below. Its type is specified by its declaration. An identifier is an lvalue if it refers to an object (Par.A.5) and if its type is arithmetic, structure, union, or pointer.

A constant is a primary expression. Its type depends on its form as discussed in Par.A.2.5.

A string literal is a primary expression. Its type is originally ``array of `char`" (for wide-char strings, ``array of `wchar_t`"), but following the rule given in Par.A.7.1, this is usually modified to ``pointer to `char`" (`wchar_t`) and the result is a pointer to the first character in the string. The conversion also does not occur in certain initializers; see Par.A.8.7.

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression. The precedence of parentheses does not affect whether the expression is an lvalue.

A.7.3 Postfix Expressions

The operators in postfix expressions group left to right.

postfix-expression:
primary-expression
postfix-expression[expression]
postfix-expression(argument-expression-list$_{opt}$*)*
postfix-expression.identifier
postfix-expression`->`*identifier*
postfix-expression`++`
postfix-expression`--`

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

A.7.3.1 Array References

A postfix expression followed by an expression in square brackets is a postfix expression denoting a subscripted array reference. One of the two expressions must have type ``pointer to *T*", where *T* is some type, and the other must have integral type; the type of the subscript expression is *T*. The expression `E1[E2]` is identical (by definition) to `*((E1)+(E2))`. See Par.A.8.6.2 for further discussion.

A.7.3.2 Function Calls

A function call is a postfix expression, called the function designator, followed by parentheses containing a possibly empty, comma-separated list of assignment expressions (Par.A7.17), which constitute the arguments to the function. If the postfix expression consists of an identifier for which no declaration exists in the current scope, the identifier is implicitly declared as if the declaration

`extern int` *identifier*`();`

had been given in the innermost block containing the function call. The postfix expression (after possible explicit declaration and pointer generation, Par.A7.1) must be of type ``pointer to function returning *T*,'' for some type *T*, and the value of the function call has type *T*.

> In the first edition, the type was restricted to ``function,'' and an explicit `*` operator was required to call through pointers to functions. The ANSI standard blesses the practice of some existing compilers by permitting the same syntax for calls to functions and to functions specified by pointers. The older syntax is still usable.

The term *argument* is used for an expression passed by a function call; the term *parameter* is used for an input object (or its identifier) received by a function definition, or described in a function declaration. The terms ``actual argument (parameter)'' and ``formal argument (parameter)'' respectively are sometimes used for the same distinction.

In preparing for the call to a function, a copy is made of each argument; all argument-passing is strictly by value. A function may change the values of its parameter objects, which are copies of the argument expressions, but these changes cannot affect the values of the arguments. However, 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.

There are two styles in which functions may be declared. In the new style, the types of parameters are explicit and are part of the type of the function; such a declaration os also called a function prototype. In the old style, parameter types are not specified. Function declaration is issued in Pars.A.8.6.3 and A.10.1.

If the function declaration in scope for a call is old-style, then default argument promotion is applied to each argument as follows: integral promotion (Par.A.6.1) is performed on each argument of integral type, and each `float` argument is converted to `double`. The effect of the call is undefined if the number of arguments disagrees with the number of parameters in the definition of the function, or if the type of an argument after promotion disagrees with that of the corresponding parameter. Type agreement depends on whether the function's definition is new-style or old-style. If it is old-style, then the comparison is between the promoted type of the arguments of the call, and the promoted type of the parameter, if the definition is new-style, the promoted type of the argument must be that of the parameter itself, without promotion.

If the function declaration in scope for a call is new-style, then the arguments are converted, as if by assignment, to the types of the corresponding parameters of the function's prototype. The number of arguments must be the same as the number of explicitly described parameters, unless the declaration's parameter list ends with the ellipsis notation `(, ...)`. In that case, the number of arguments must equal or exceed the number of parameters; trailing arguments beyond the explicitly typed parameters suffer default argument promotion as described in the preceding paragraph. If the definition of the function is old-style, then the type of each parameter in the definition, after the definition parameter's type has undergone argument promotion.

> These rules are especially complicated because they must cater to a mixture of old- and new-style functions. Mixtures are to be avoided if possible.

The order of evaluation of arguments is unspecified; take note that various compilers differ. However, the arguments and the function designator are completely evaluated, including all side effects, before the function is entered. Recursive calls to any function are permitted.

A.7.3.3 Structure References

A postfix expression followed by a dot followed by an identifier is a postfix expression. The first operand 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 its

type is the type of the member. The expression is an lvalue if the first expression is an lvalue, and if the type of the second expression is not an array type.

A postfix expression followed by an arrow (built from `-` and `>`) followed by an identifier is a postfix expression. The first operand expression must be a pointer to a structure or union, and the identifier must name a member of the structure or union. The result refers to the named member of the structure or union to which the pointer expression points, and the type is the type of the member; the result is an lvalue if the type is not an array type.

Thus the expression `E1->MOS` is the same as `(*E1).MOS`. Structures and unions are discussed in Par.A.8.3.

> In the first edition of this book, it was already the rule that a member name in such an expression had to belong to the structure or union mentioned in the postfix expression; however, a note admitted that this rule was not firmly enforced. Recent compilers, and ANSI, do enforce it.

A.7.3.4 Postfix Incrementation

A postfix expression followed by a `++` or `--` operator is a postfix expression. The value of the expression is the value of the operand. After the value is noted, the operand is incremented `++` or decremented `--` by 1. The operand must be an lvalue; see the discussion of additive operators (Par.A.7.7) and assignment (Par.A.7.17) for further constraints on the operand and details of the operation. The result is not an lvalue.

A.7.4 Unary Operators

Expressions with unary operators group right-to-left.

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

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

A.7.4.1 Prefix Incrementation Operators

A unary expression followed by a `++` or `--` operator is a unary expression. The operand is incremented `++` or decremented `--` by 1. The value of the expression is the value after the incrementation (decrementation). The operand must be an lvalue; see the discussion of additive operators (Par.A.7.7) and assignment (Par.A.7.17) for further constraints on the operands and details of the operation. The result is not an lvalue.

A.7.4.2 Address Operator

The unary operator `&` takes the address of its operand. The operand must be an lvalue referring neither to a bit-field nor to an object declared as `register`, or must be of function type. The result is a pointer to the object or function referred to by the lvalue. If the type of the operand is *T*, the type of the result is ``pointer to *T*."

A.7.4.3 Indirection Operator

The unary `*` operator denotes indirection, and returns the object or function to which its operand points. It is an lvalue if the operand is a pointer to an object of arithmetic, structure, union, or pointer type. If the type of the expression is ``pointer to *T*," the type of the result is *T*.

A.7.4.4 Unary Plus Operator

The operand of the unary + operator must have arithmetic type, and the result is the value of the operand. An integral operand undergoes integral promotion. The type of the result is the type of the promoted operand.

The unary + is new with the ANSI standard. It was added for symmetry with the unary -.

A.7.4.5 Unary Minus Operator

The operand of the unary - operator must have arithmetic type, and the result is the negative of its operand. An integral operand undergoes integral promotion. The negative of an unsigned quantity is computed by subtracting the promoted value from the largest value of the promoted type and adding one; but negative zero is zero. The type of the result is the type of the promoted operand.

A.7.4.6 One's Complement Operator

The operand of the ~ operator must have integral type, and the result is the one's complement of its operand. The integral promotions are performed. If the operand is unsigned, the result is computed by subtracting the value from the largest value of the promoted type. If the operand is signed, the result is computed by converting the promoted operand to the corresponding unsigned type, applying ~, and converting back to the signed type. The type of the result is the type of the promoted operand.

A.7.4.7 Logical Negation Operator

The operand of the `!` operator must have arithmetic type or be a pointer, and the result is 1 if the value of its operand compares equal to 0, and 0 otherwise. The type of the result is `int`.

A.7.4.8 Sizeof Operator

The `sizeof` operator yields the number of bytes required to store an object of the type of its operand. The operand is either an expression, which is not evaluated, or a parenthesized type name. When `sizeof` is applied to a `char`, the result is 1; when applied to an array, the result is the total number of bytes in the array. When applied to a structure or union, the result is the number of bytes in the object, including any padding required to make the object tile an array: the size of an array of *n* elements is *n* times the size of one element. The operator may not be applied to an operand of function type, or of incomplete type, or to a bit-field. The result is an unsigned integral constant; the particular type is implementation-defined. The standard header `<stddef.h>` (See appendix B) defines this type as `size_t`.

A.7.5 Casts

A unary expression preceded by the parenthesized name of a type causes conversion of the value of the expression to the named type.

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

This construction is called a *cast*. The names are described in Par.A.8.8. The effects of conversions are described in Par.A.6. An expression with a cast is not an lvalue.

A.7.6 Multiplicative Operators

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

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

The operands of * and / must have arithmetic type; the operands of % must have integral type. The usual arithmetic conversions are performed on the operands, and predict the type of the result.

The binary * operator denotes multiplication.

The binary / operator yields the quotient, and the % operator the remainder, of the division of the first operand by the second; if the second operand is 0, the result is undefined. Otherwise, it is always true that (a/b)*b + a%b is equal to a. If both operands are non-negative, then the remainder is non-negative and smaller than the divisor, if not, it is guaranteed only that the absolute value of the remainder is smaller than the absolute value of the divisor.

A.7.7 Additive Operators

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

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

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

> The provision for pointers just beyond the end of an array is new. It legitimizes a common idiom for looping over the elements of an array.

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

If two pointers to objects of the same type are subtracted, the result is a signed integral value representing the displacement between the pointed-to objects; pointers to successive objects differ by 1. The type of the result is defined as ptrdiff_t in the standard header <stddef.h>. The value is undefined unless the pointers point to objects within the same array; however, if P points to the last member of an array, then (P+1)-P has value 1.

A.7.8 Shift Operators

The shift operators << and >> group left-to-right. For both operators, each operand must be integral, and is subject to integral the promotions. The type of the result is that of the promoted left operand. The result is undefined if the right operand is negative, or greater than or equal to the number of bits in the left expression's type.

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

The value of E1<<E2 is E1 (interpreted as a bit pattern) left-shifted E2 bits; in the absence of overflow, this is equivalent to multiplication by 2^{E2}. The value of E1>>E2 is E1 right-shifted E2

bit positions. The right shift is equivalent to division by 2^{E2} if `E1` is unsigned or it has a non-negative value; otherwise the result is implementation-defined.

A.7.9 Relational Operators

The relational operators group left-to-right, but this fact is not useful; `a<b<c` is parsed as `(a<b)<c`, and evaluates to either 0 or 1.

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

The operators < (less), > (greater), <= (less or equal) and >= (greater or equal) all yield 0 if the specified relation is false and 1 if it is true. The type of the result is `int`. The usual arithmetic conversions are performed on arithmetic operands. Pointers to objects of the same type (ignoring any qualifiers) may be compared; the result depends on the relative locations in the address space of the pointed-to objects. Pointer comparison is defined only for parts of the same object; if two pointers point to the same simple object, they compare equal; if the pointers are to members of the same structure, pointers to objects declared later in the structure compare higher; if the pointers refer to members of an array, the comparison is equivalent to comparison of the the corresponding subscripts. If `P` points to the last member of an array, then `P+1` compares higher than `P`, even though `P+1` points outside the array. Otherwise, pointer comparison is undefined.

> These rules slightly liberalize the restrictions stated in the first edition, by permitting comparison of pointers to different members of a structure or union. They also legalize comparison with a pointer just off the end of an array.

A.7.10 Equality Operators

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

The == (equal to) and the != (not equal to) operators are 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.)

The equality operators follow the same rules as the relational operators, but permit additional possibilities: a pointer may be compared to a constant integral expression with value 0, or to a pointer to `void`. See Par.A.6.6.

A.7.11 Bitwise AND Operator

AND-expression:
  equality-expression
  AND-expression & *equality-expression*

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

A.7.12 Bitwise Exclusive OR Operator

exclusive-OR-expression:
  AND-expression
  exclusive-OR-expression ^ *AND-expression*

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

A.7.13 Bitwise Inclusive OR Operator

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

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

A.7.14 Logical AND Operator

logical-AND-expression:
  inclusive-OR-expression
  logical-AND-expression `&&` *inclusive-OR-expression*

The `&&` operator groups left-to-right. It returns 1 if both its operands compare unequal to zero, 0 otherwise. Unlike `&`, `&&` guarantees left-to-right evaluation: the first operand is evaluated, including all side effects; if it is equal to 0, the value of the expression is 0. Otherwise, the right operand is evaluated, and if it is equal to 0, the expression's value is 0, otherwise 1.

The operands need not have the same type, but each must have arithmetic type or be a pointer. The result is `int`.

A.7.15 Logical OR Operator

logical-OR-expression:
  logical-AND-expression
  logical-OR-expression `||` *logical-AND-expression*

The `||` operator groups left-to-right. It returns 1 if either of its operands compare unequal to zero, and 0 otherwise. Unlike `|`, `||` guarantees left-to-right evaluation: the first operand is evaluated, including all side effects; if it is unequal to 0, the value of the expression is 1. Otherwise, the right operand is evaluated, and if it is unequal to 0, the expression's value is 1, otherwise 0.

The operands need not have the same type, but each must have arithmetic type or be a pointer. The result is `int`.

A.7.16 Conditional Operator

conditional-expression:
  logical-OR-expression
  logical-OR-expression `?` *expression* `:` *conditional-expression*

The first expression is evaluated, including all side effects; if it compares unequal to 0, the result is the value of the second expression, otherwise that of the third expression. Only one of the second and third operands is evaluated. If the second and third operands are arithmetic, the usual arithmetic conversions are performed to bring them to a common type, and that type is the type of the result. If both are `void`, or structures or unions of the same type, or pointers to objects of the same type, the result has the common type. If one is a pointer and the other the constant 0, the 0 is converted to the pointer type, and the result has that type. If one is a pointer to `void` and the other is another pointer, the other pointer is converted to a pointer to `void`, and that is the type of the result.

In the type comparison for pointers, any type qualifiers (Par.A.8.2) in the type to which the pointer points are insignificant, but the result type inherits qualifiers from both arms of the conditional.

A.7.17 Assignment Expressions

There are several assignment operators; all group right-to-left.

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

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

All require an lvalue as left operand, and the lvalue must be modifiable: it must not be an array, and must not have an incomplete type, or be a function. Also, its type must not be qualified with `const`; if it is a structure or union, it must not have any member or, recursively, submember qualified with `const`. The type of an assignment expression is that of its left operand, and the value is the value stored in the left operand after the assignment has taken place.

In the simple assignment with `=`, the value of the expression replaces that of the object referred to by the lvalue. One of the following must be true: both operands have arithmetic type, in which case the right operand is converted to the type of the left by the assignment; or both operands are structures or unions of the same type; or one operand is a pointer and the other is a pointer to `void`, or the left operand is a pointer and the right operand is a constant expression with value 0; or both operands are pointers to functions or objects whose types are the same except for the possible absence of `const` or `volatile` in the right operand.

An expression of the form `E1` *op*`=` `E2` is equivalent to `E1 = E1` *op* `(E2)` except that `E1` is evaluated only once.

A.7.18 Comma Operator

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

A pair of expressions separated by a comma is evaluated left-to-right, and the value of the left expression is discarded. The type and value of the result are the type and value of the right operand. All side effects from the evaluation of the left-operand are completed before beginning the evaluation of the right operand. In contexts where comma is given a special meaning, for example in lists of function arguments (Par.A.7.3.2) and lists of initializers (Par.A.8.7), the required syntactic unit is an assignment expression, so the comma operator appears only in a parenthetical grouping, for example,

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

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

A.7.19 Constant Expressions

Syntactically, a constant expression is an expression restricted to a subset of operators:

constant-expression:
  conditional-expression

Expressions that evaluate to a constant are required in several contexts: after `case`, as array bounds and bit-field lengths, as the value of an enumeration constant, in initializers, and in certain preprocessor expressions.

Constant expressions may not contain assignments, increment or decrement operators, function calls, or comma operators; except in an operand of `sizeof`. If the constant expression is required to be integral, its operands must consist of integer, enumeration, character, and floating constants; casts must specify an integral type, and any floating constants must be cast to integer. This necessarily rules out arrays, indirection, address-of, and structure member operations. (However, any operand is permitted for `sizeof`.)

More latitude is permitted for the constant expressions of initializers; the operands may be any type of constant, and the unary `&` operator may be applied to external or static objects, and to external and static arrays subscripted with a constant expression. The unary `&` operator can also be applied implicitly by appearance of unsubscripted arrays and functions. 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 the integral constant expressions after `#if`; `sizeof` expressions, enumeration constants, and casts are not permitted. See Par.A.12.5.

A.8 Declarations

Declarations specify the interpretation given to each identifier; they do not necessarily reserve storage associated with the identifier. Declarations that reserve storage are called *definitions*. Declarations have the form

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

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

declaration-specifiers:
  storage-class-specifier declaration-specifiers$_{opt}$
  type-specifier declaration-specifiers$_{opt}$
  type-qualifier declaration-specifiers$_{opt}$

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

init-declarator:
  declarator
  declarator `=` *initializer*

Declarators will be discussed later (Par.A.8.5); they contain the names being declared. A declaration must have at least one declarator, or its type specifier must declare a structure tag, a union tag, or the members of an enumeration; empty declarations are not permitted.

A.8.1 Storage Class Specifiers

The storage class specifiers are:

storage-class specifier:
  `auto`
  `register`

```
static
extern
typedef
```

The meaning of the storage classes were discussed in Par.A.4.4.

The `auto` and `register` specifiers give the declared objects automatic storage class, and may be used only within functions. Such declarations also serve as definitions and cause storage to be reserved. A `register` declaration is equivalent to an `auto` declaration, but hints that the declared objects will be accessed frequently. Only a few objects are actually placed into registers, and only certain types are eligible; the restrictions are implementation-dependent. However, if an object is declared `register`, the unary `&` operator may not be applied to it, explicitly or implicitly.

> The rule that it is illegal to calculate the address of an object declared `register`, but actually taken to be `auto`, is new.

The `static` specifier gives the declared objects static storage class, and may be used either inside or outside functions. Inside a function, this specifier causes storage to be allocated, and serves as a definition; for its effect outside a function, see Par.A.11.2.

A declaration with `extern`, used inside a function, specifies that the storage for the declared objects is defined elsewhere; for its effects outside a function, see Par.A.11.2.

The `typedef` specifier does not reserve storage and is called a storage class specifier only for syntactic convenience; it is discussed in Par.A.8.9.

At most one storage class specifier may be given in a declaration. If none is given, these rules are used: objects declared inside a function are taken to be `auto`; functions declared within a function are taken to be `extern`; objects and functions declared outside a function are taken to be `static`, with external linkage. See Pars. A.10-A.11.

A.8.2 Type Specifiers

The type-specifiers are

type specifier:
`void`
`char`
`short`
`int`
`long`
`float`
`double`
`signed`
`unsigned`
struct-or-union-specifier
enum-specifier
typedef-name

At most one of the words `long` or `short` may be specified together with `int`; the meaning is the same if `int` is not mentioned. The word `long` may be specified together with `double`. At most one of `signed` or `unsigned` may be specified together with `int` or any of its `short` or `long` varieties, or with `char`. Either may appear alone in which case `int` is understood. The `signed` specifier is useful for forcing `char` objects to carry a sign; it is permissible but redundant with other integral types.

Otherwise, at most one type-specifier may be given in a declaration. If the type-specifier is missing from a declaration, it is taken to be `int`.

Types may also be qualified, to indicate special properties of the objects being declared.

type-qualifier:
`const`
`volatile`

Type qualifiers may appear with any type specifier. A `const` object may be initialized, but not thereafter assigned to. There are no implementation-dependent semantics for `volatile` objects.

> The `const` and `volatile` properties are new with the ANSI standard. The purpose of `const` is to announce objects that may be placed in read-only memory, and perhaps to increase opportunities for optimization. The purpose of `volatile` is to force an implementation to suppress optimization that could otherwise occur. For example, for a machine with memory-mapped input/output, a pointer to a device register might be declared as a pointer to `volatile`, in order to prevent the compiler from removing apparently redundant references through the pointer. Except that it should diagnose explicit attempts to change `const` objects, a compiler may ignore these qualifiers.

A.8.3 Structure and Union Declarations

A structure is an object consisting of a sequence of named members of various types. A union is an object that contains, at different times, any of several members of various types. Structure and union specifiers have the same form.

struct-or-union-specifier:
struct-or-union identifier$_{opt}$ `{` *struct-declaration-list* `}`
struct-or-union identifier

struct-or-union:
`struct`
`union`

A struct-declaration-list is a sequence of declarations for the members of the structure or union:

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

struct-declaration: *specifier-qualifier-list struct-declarator-list* `;`

specifier-qualifier-list:
type-specifier specifier-qualifier-list$_{opt}$
type-qualifier specifier-qualifier-list$_{opt}$

struct-declarator-list:
struct-declarator
struct-declarator-list `,` *struct-declarator*

Usually, 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 *bit-field*; its length is set off from the declarator for the field name by a colon.

struct-declarator:
declarator *declarator*$_{opt}$ `:` *constant-expression*

A type specifier of the form

struct-or-union identifier `{` *struct-declaration-list* `}`

declares the identifier to be the *tag* of the structure or union specified by the list. A subsequent declaration in the same or an inner scope may refer to the same type by using the tag in a specifier without the list:

struct-or-union identifier

If a specifier with a tag but without a list appears when the tag is not declared, an *incomplete type* is specified. Objects with an incomplete structure or union type may be mentioned in contexts where their size is not needed, for example in declarations (not definitions), for specifying a pointer, or for creating a `typedef`, but not otherwise. The type becomes complete on occurrence of a subsequent specifier with that tag, and containing a declaration list. Even in specifiers with a list, the structure or union type being declared is incomplete within the list, and becomes complete only at the `}` terminating the specifier.

A structure may not contain a member of incomplete type. Therefore, it is impossible to declare a structure or union containing an instance of itself. However, besides giving a name to the structure or union type, tags allow definition of self-referential structures; a structure or union may contain a pointer to an instance of itself, because pointers to incomplete types may be declared.

A very special rule applies to declarations of the form

struct-or-union identifier `;`

that declare a structure or union, but have no declaration list and no declarators. Even if the identifier is a structure or union tag already declared in an outer scope (Par.A.11.1), this declaration makes the identifier the tag of a new, incompletely-typed structure or union in the current scope.

> This recondite is new with ANSI. It is intended to deal with mutually-recursive structures declared in an inner scope, but whose tags might already be declared in the outer scope.

A structure or union specifier with a list but no tag creates a unique type; it can be referred to directly only in the declaration of which it is a part.

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

> In the first edition of this book, the names of structure and union members were not associated with their parent. However, this association became common in compilers well before the ANSI standard.

A non-field member of a structure or union may have any object type. A field member (which need not have a declarator and thus may be unnamed) has type `int`, `unsigned int`, or `signed int`, and is interpreted as an object of integral type of the specified length in bits; whether an `int` field is treated as signed is implementation-dependent. Adjacent field members of structures are packed into implementation-dependent storage units in an implementation-dependent direction. When a field following another field will not fit into a partially-filled storage unit, it may be split between units, or the unit may be padded. An unnamed field with width 0 forces this padding, so that the next field will begin at the edge of the next allocation unit.

> The ANSI standard makes fields even more implementation-dependent than did the first edition. It is advisable to read the language rules for storing bit-fields as ``implementation-dependent" without qualification. Structures with bit-fields may be used as a portable way of attempting to reduce the storage required for a structure (with the probable cost of increasing the instruction space, and time,

> needed to access the fields), or as a non-portable way to describe a storage layout known at the bit-level. In the second case, it is necessary to understand the rules of the local implementation.

The members of a structure have addresses increasing in the order of their declarations. A non-field member of a structure is aligned at an addressing boundary depending on its type; therefore, there may be unnamed holes in a structure. If a pointer to a structure is cast to the type of a pointer to its first member, the result refers to the first member.

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. If a pointr to a union is cast to the type of a pointer to a member, the result refers to that member.

A simple example of a structure declaration is

```
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 bene 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;

```
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`.

In general, a member of a union may not be inspected unless the value of the union has been assigned using the same member. However, one special guarantee simplifies the use of unions: if a union contains several structures that share a common initial sequence, and the union currently contains one of these structures, it is permitted to refer to 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) ...
```

A.8.4 Enumerations

Enumerations are unique types with values ranging over a set of named constants called enumerators. The form of an enumeration specifier borrows from that of structures and unions.

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

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

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

The identifiers in an enumerator list are declared as constants of type `int`, and may appear wherever constants are required. If no enumerations 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 specified; subsequent identifiers continue the progression from the assigned value.

Enumerator names in the same scope must all be distinct from each other and from ordinary variable names, but the values need not be distinct.

The role of the identifier in the enum-specifier is analogous to that of the structure tag in a struct-specifier; it names a particular enumeration. The rules for enum-specifiers with and without tags and lists are the same as those for structure or union specifiers, except that incomplete enumeration types do not exist; the tag of an enum-specifier without an enumerator list must refer to an in-scope specifier with a list.

> Enumerations are new since the first edition of this book, but have been part of the language for some years.

A.8.5 Declarators

Declarators have the syntax:

declarator:
  pointer$_{opt}$ *direct-declarator*

direct-declarator:
  identifier
  `(`*declarator*`)`
  direct-declarator `[` *constant-expression*$_{opt}$ `]`
  direct-declarator `(` *parameter-type-list* `)`
  direct-declarator `(` *identifier-list*$_{opt}$ `)`

pointer:
  `*` *type-qualifier-list*$_{opt}$
  `*` *type-qualifier-list*$_{opt}$ *pointer*

type-qualifier-list:
  type-qualifier
  type-qualifier-list type-qualifier

The structure of declarators resembles that of indirection, function, and array expressions; the grouping is the same.

A.8.6 Meaning of Declarators

A list of declarators appears after a sequence of type and storage class specifiers. Each declarator declares a unique main identifier, the one that appears as the first alternative of the production for *direct-declarator*. The storage class specifiers apply directly to this identifier, but its type depends on the form of its declarator. A declarator is read as an assertion that when its identifier appears in an expression of the same form as the declarator, it yields an object of the specified type.

Considering only the type parts of the declaration specifiers (Par. A.8.2) and a particular declarator, a declaration has the form ``T D," where T is a type and D is a declarator. The type attributed to the identifier in the various forms of declarator is described inductively using this notation.

In a declaration T D where D is an unadored identifier, the type of the identifier is T.

In a declaration T D where D has the form

```
( D1 )
```

then the type of the identifier in D1 is the same as that of D. The parentheses do not alter the type, but may change the binding of complex declarators.

A.8.6.1 Pointer Declarators

In a declaration T D where D has the form

* *type-qualifier-list*$_{opt}$ D1

and the type of the identifier in the declaration T D1 is ``*type-modifier* T," the type of the identifier of D is ``*type-modifier type-qualifier-list* pointer to T." Qualifiers following * apply to pointer itself, rather than to the object to which the pointer points.

For example, consider the declaration

```
int *ap[];
```

Here, ap[] plays the role of D1; a declaration ``int ap[]" (below) would give ap the type ``array of int," the type-qualifier list is empty, and the type-modifier is ``array of." Hence the actual declaration gives ap the type ``array to pointers to int."

As other examples, the declarations

```
int i, *pi, *const cpi = &i;
const int ci = 3, *pci;
```

declare an integer i and a pointer to an integer pi. The value of the constant pointer cpi may not be changed; it will always point to the same location, although the value to which it refers may be altered. The integer ci is constant, and may not be changed (though it may be initialized, as here.) The type of pci is ``pointer to const int," and pci itself may be changed to point to another place, but the value to which it points may not be altered by assigning through pci.

A.8.6.2 Array Declarators

In a declaration T D where D has the form

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

and the type of the identifier in the declaration `T D1` is ``*type-modifier* `T`," the type of the identifier of `D` is ``*type-modifier* array of `T`." If the constant-expression is present, it must have integral type, and value greater than 0. If the constant expression specifying the bound is missing, the array has an incomplete type.

An array may be constructed from an arithmetic type, from a pointer, from a structure or union, or from another array (to generate a multi-dimensional array). Any type from which an array is constructed must be complete; it must not be an array of structure of incomplete type. This implies that for a multi-dimensional array, only the first dimension may be missing. The type of an object of incomplete aray type is completed by another, complete, declaration for the object (Par.A.10.2), or by initializing it (Par.A.8.7). For example,

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

declares an array of `float` numbers and an array of pointers to `float` numbers. Also,

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

declares a static three-dimensional array of integers, with rank 3 `X` 5 `X` 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`. More specifically, `x3d[i][j]` is an array of 7 integers, and `x3d[i]` is an array of 5 arrays of 7 integers.

The array subscripting operation is defined so that `E1[E2]` is identical to `*(E1+E2)`. Therefore, despite its asymmetric appearance, subscripting is a commutative operation. Because of the conversion rules that apply to `+` and to arrays (Pars.A6.6, A.7.1, A.7.7), if `E1` is an array and `E2` an integer, then `E1[E2]` refers to the `E2`-th member of `E1`.

In the example, `x3d[i][j][k]` is equivalent to `*(x3d[i][j] + k)`. The first subexpression `x3d[i][j]` is converted by Par.A.7.1 to type ``pointer to array of integers," by Par.A.7.7, the addition involves multiplication by the size of an integer. It follows from the rules that arrays are stored by rows (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.

A.8.6.3 Function Declarators

In a new-style function declaration `T D` where `D` has the form

`D1` (*parameter-type-list*)

and the type of the identifier in the declaration `T D1` is ``*type-modifier* `T`," the type of the identifier of `D` is ``*type-modifier* function with arguments *parameter-type-list* returning `T`."

The syntax of the parameters is

parameter-type-list:
  parameter-list
  parameter-list `,` `...`

parameter-list:
  parameter-declaration
  parameter-list `,` *parameter-declaration*

parameter-declaration:
  declaration-specifiers declarator
  declaration-specifiers abstract-declarator$_{opt}$

In the new-style declaration, the parameter list specifies the types of the parameters. As a special case, the declarator for a new-style function with no parameters has a parameter list consisting soley of the keyword `void`. If the parameter list ends with an ellipsis "`, ...`", then the function may accept more arguments than the number of parameters explicitly described, see Par.A.7.3.2.

The types of parameters that are arrays or functions are altered to pointers, in accordance with the rules for parameter conversions; see Par.A.10.1. The only storage class specifier permitted in a parameter's declaration is `register`, and this specifier is ignored unless the function declarator heads a function definition. Similarly, if the declarators in the parameter declarations contain identifiers and the function declarator does not head a function definition, the identifiers go out of scope immediately. Abstract declarators, which do not mention the identifiers, are discussed in Par.A.8.8.

In an old-style function declaration `T D` where `D` has the form

`D1`(*identifier-list*$_{opt}$)

and the type of the identifier in the declaration `T D1` is "*type-modifier* `T`," the type of the identifier of `D` is "*type-modifier* function of unspecified arguments returning `T`." The parameters (if present) have the form

identifier-list:
identifier
identifier-list `,` *identifier*

In the old-style declarator, the identifier list must be absent unless the declarator is used in the head of a function definition (Par.A.10.1). No information about the types of the parameters is supplied by the declaration.

For example, the declaration

```
int f(), *fpi(), (*pfi)();
```

declares a function `f` returning an integer, a function `fpi` returning a pointer to an integer, and a pointer `pfi` to a function returning an integer. In none of these are the parameter types specified; they are old-style.

In the new-style declaration

```
int strcpy(char *dest, const char *source), rand(void);
```

`strcpy` is a function returning `int`, with two arguments, the first a character pointer, and the second a pointer to constant characters. The parameter names are effectively comments. The second function `rand` takes no arguments and returns `int`.

Function declarators with parameter prototypes are, by far, the most important language change introduced by the ANSI standard. They offer an advantage over the "old-style" declarators of the first edition by providing error-detection and coercion of arguments across function calls, but at a cost: turmoil and confusion during their introduction, and the necessity of accomodating both forms. Some syntactic ugliness was required for the sake of compatibility, namely `void` as an explicit marker of new-style functions without parameters.

The ellipsis notation "`, ...`" for variadic functions is also new, and, together with the macros in the standard header `<stdarg.h>`, formalizes a mechanism that was officially forbidden but unofficially condoned in the first edition.

These notations were adapted from the C++ language.

A.8.7 Initialization

When an object is declared, its init-declarator may specify an initial value for the identifier being declared. The initializer is preceded by =, and is either an expression, or a list of initializers nested in braces. A list may end with a comma, a nicety for neat formatting.

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

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

All the expressions in the initializer for a static object or array must be constant expressions as described in Par.A.7.19. The expressions in the initializer for an `auto` or `register` object or array must likewise be constant expressions if the initializer is a brace-enclosed list. However, if the initializer for an automatic object is a single expression, it need not be a constant expression, but must merely have appropriate type for assignment to the object.

> The first edition did not countenance initialization of automatic structures, unions, or arrays. The ANSI standard allows it, but only by constant constructions unless the initializer can be expressed by a simple expression.

A static object not explicitly initialized is initialized as if it (or its members) were assigned the constant 0. The initial value of an automatic object not explicitly intialized is undefined.

The initializer for a pointer or an object of arithmetic type is a single expression, perhaps in braces. The expression is assigned to the object.

The initializer for a structure is either an expression of the same type, or a brace-enclosed list of initializers for its members in order. Unnamed bit-field members are ignored, and are not initialized. If there are fewer initializers in the list than members of the structure, the trailing members are initialized with 0. There may not be more initializers than members. Unnamed bit-field members are ignored,and are not initialized.

The initializer for an array is a brace-enclosed list of initializers for its members. If the array has unknown size, the number of initializers determines the size of the array, and its type becomes complete. If the array has fixed size, the number of initializers may not exceed the number of members of the array; if there are fewer, the trailing members are initialized with 0.

As a special case, a character array may be initialized by a string literal; successive characters of the string initialize successive members of the array. Similarly, a wide character literal (Par.A.2.6) may initialize an array of type `wchar_t`. If the array has unknown size, the number of characters in the string, including the terminating null character, determines its size; if its size is fixed, the number of characters in the string, not counting the terminating null character, must not exceed the size of the array.

The initializer for a union is either a single expression of the same type, or a brace-enclosed initializer for the first member of the union.

> The first edition did not allow initialization of unions. The ``first-member'' rule is clumsy, but is hard to generalize without new syntax. Besides allowing unions to be explicitly initialized in at least a primitive way, this ANSI rule makes definite the semantics of static unions not explicitly initialized.

An *aggregate* is a structure or array. If an aggregate contains members of aggregate type, the initialization rules apply recursively. Braces may be elided in the initialization as follows: if the initializer for an aggregate's member that itself is an aggregate begins with a left brace, then the succeding comma-separated list of initializers initializes the members of the subaggregate; it is erroneous for there to be more initializers than members. If, however, the initializer for a

subaggregate does not begin with a left brace, then only enough elements from the list are taken into account for the members of the subaggregate; any remaining members are left to initialize the next member of the aggregate of which the subaggregate is a part.

For example,

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

declares and initializes `x` as a 1-dimensional array with three members, since no size was specified and there are three initializers.

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

is a completely-bracketed initialization: 1, 3 and 5 initialize the first row of the array `y[0]`, namely `y[0][0]`, `y[0][1]`, and `y[0][2]`. Likewise the next two lines initialize `y[1]` and `y[2]`. The initializer ends early, and therefore the elements of `y[3]` are 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 for `y[2]`. Also,

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

initializes the first column of `y` (regarded as a two-dimensional array) and leaves the rest 0.

Finally,

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

shows a character array whose members are initialized with a string; its size includes the terminating null character.

A.8.8 Type names

In several contexts (to specify type conversions explicitly with a cast, to declare parameter types in function declarators, and as argument of `sizeof`) it is necessary to supply the name of a data type. This is accomplished using a *type name*, which is syntactically a declaration for an object of that type omitting the name of the object.

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

abstract-declarator:
  pointer
  pointer$_{opt}$ *direct-abstract-declarator*

direct-abstract-declarator:
  (*abstract-declarator*)
  direct-abstract-declarator$_{opt}$ [*constant-expression*$_{opt}$]
  direct-abstract-declarator$_{opt}$ (*parameter-type-list*$_{opt}$)

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 *()
int (*[])(void)
```

name respectively the types ``integer,'' ``pointer to integer,'' ``array of 3 pointers to integers,'' ``pointer to an unspecified number of integers,'' ``function of unspecified parameters returning pointer to integer,'' and ``array, of unspecified size, of pointers to functions with no parameters each returning an integer.''

A.8.9 Typedef

Declarations whose storage class specifier is `typedef` do not declare objects; instead they define identifiers that name types. These identifiers are called typedef names.

typedef-name:
identifier

A `typedef` declaration attributes a type to each name among its declarators in the usual way (see Par.A.8.6). Thereafter, each such typedef name is syntactically equivalent to a type specifier keyword for the associated type.

For example, after

```
typedef long Blockno, *Blockptr;
typedef struct { double r, theta; } Complex;
```

the constructions

```
Blockno b;
extern Blockptr bp;
Complex z, *zp;
```

are legal declarations. The type of `b` is `long`, that of `bp` is ``pointer to `long`,'' and that of `z` is the specified structure; `zp` is a pointer to such a structure.

`typedef` does not introduce new types, only synonyms for types that could be specified in another way. In the example, `b` has the same type as any `long` object.

Typedef names may be redeclared in an inner scope, but a non-empty set of type specifiers must be given. For example,

```
extern Blockno;
```

does not redeclare `Blockno`, but

```
extern int Blockno;
```

does.

A.8.10 Type Equivalence

Two type specifier lists are equivalent if they contain the same set of type specifiers, taking into account that some specifiers can be implied by others (for example, `long` alone implies `long int`). Structures, unions, and enumerations with different tags are distinct, and a tagless union, structure, or enumeration specifies a unique type.

Two types are the same if their abstract declarators (Par.A.8.8), after expanding any `typedef` types, and deleting any function parameter specifiers, are the same up to the equivalence of type specifier lists. Array sizes and function parameter types are significant.

A.9 Statements

Except as described, statements are executed in sequence. Statements are executed for their effect, and do not have values. They fall into several groups.

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

A.9.1 Labeled Statements

Statements may carry label prefixes.

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

A label consisting of an identifier declares the identifier. The only use of an identifier label is as a target of `goto`. The scope of the identifier is the current function. Because labels have their own name space, they do not interfere with other identifiers and cannot be redeclared. See Par.A.11.1.

Case labels and default labels are used with the `switch` statement (Par.A.9.4). The constant expression of `case` must have integral type.

Labels themselves do not alter the flow of control.

A.9.2 Expression Statement

Most statements are expression statements, which have the form

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

Most expression statements are assignments or function calls. All side effects from the expression are completed before the next statement is executed. If the expression is missing, the construction is called a null statement; it is often used to supply an empty body to an iteration statement to place a label.

A.9.3 Compound Statement

So that several statements can be used where one is expected, the compound statement (also called ``block'') is provided. The body of a function definition is a compound statement.

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

declaration-list:
  declaration
  declaration-list declaration

statement-list:
statement
statement-list statement

If an identifier in the declaration-list was in scope outside the block, the outer declaration is suspended within the block (see Par.A.11.1), after which it resumes its force. An identifier may be declared only once in the same block. These rules apply to identifiers in the same name space (Par.A.11); identifiers in different name spaces are treated as distinct.

Initialization of automatic objects is performed each time the block is entered at the top, and proceeds in the order of the declarators. If a jump into the block is executed, these initializations are not performed. Initialization of `static` objects are performed only once, before the program begins execution.

A.9.4 Selection Statements

Selection statements choose one of several flows of control.

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

In both forms of the `if` statement, the expression, which must have arithmetic or pointer type, is evaluated, including all side effects, and if it compares unequal to 0, the first substatement is executed. In the second form, 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` at the same block nesting level.

The `switch` statement causes control to be transferred to one of several statements depending on the value of an expression, which must have integral type. The substatement controlled by a `switch` is typically compound. Any statement within the substatement may be labeled with one or more `case` labels (Par.A.9.1). The controlling expression undergoes integral promotion (Par.A.6.1), and the case constants are converted to the promoted type. No two of these case constants associated with the same switch may have the same value after conversion. There may also be at most one `default` label associated with a switch. Switches may be nested; a `case` or `default` label is associated with the smallest switch that contains it.

When the `switch` statement is executed, its expression is evaluated, including all side effects, and compared with each case constant. If one of the case constants is equal to the value of the expression, control passes to the statement of the matched `case` label. If no case constant matches the expression, and if there is a `default` label, control passes to the labeled statement. If no case matches, and if there is no `default`, then none of the substatements of the swtich is executed.

> In the first edition of this book, the controlling expression of `switch`, and the case constants, were required to have `int` type.

A.9.5 Iteration Statements

Iteration statements specify looping.

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

In the `while` and `do` statements, the substatement is executed repeatedly so long as the value of the expression remains unequal to 0; the expression must have arithmetic or pointer type. With `while`, the test, including all side effects from the expression, occurs before each execution of the statement; with `do`, the test follows each iteration.

In the `for` statement, the first expression is evaluated once, and thus specifies initialization for the loop. There is no restriction on its type. The second expression must have arithmetic or pointer type; it is evaluated before each iteration, and if it becomes equal to 0, the `for` is terminated. The third expression is evaluated after each iteration, and thus specifies a re-initialization for the loop. There is no restriction on its type. Side-effects from each expression are completed immediately after its evaluation. If the substatement does not contain `continue`, a statement

`for` (*expression1*`;` *expression2*`;` *expression3*) *statement*

is equivalent to

```
expression1;
while (expression2) {
    statement
    expression3;
}
```

Any of the three expressions may be dropped. A missing second expression makes the implied test equivalent to testing a non-zero element.

A.9.6 Jump statements

Jump statements transfer control unconditionally.

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

In the `goto` statement, the identifier must be a label (Par.A.9.1) located in the current function. Control transfers to the labeled statement.

A `continue` statement may appear only within an iteration statement. It causes control to pass to the loop-continuation portion of the smallest enclosing such statement. More precisely, within each of the statements

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

a `continue` not contained in a smaller iteration statement is the same as `goto contin`.

A `break` statement may appear only in an iteration statement or a `switch` statement, and terminates execution of the smallest enclosing such statement; control passes to the statement following the terminated statement.

A function returns to its caller by the `return` statement. When `return` is followed by an expression, the value is returned to the caller of the function. The expression is converted, as by assignment, to the type returned by the function in which it appears.

Flowing off the end of a function is equivalent to a return with no expression. In either case, the returned value is undefined.

A.10 External Declarations

The unit of input provided to the C compiler is called a translation unit; it consists of a sequence of external declarations, which are either declarations or function definitions.

translation-unit:
  external-declaration
  translation-unit external-declaration

external-declaration:
  function-definition
  declaration

The scope of external declarations persists to the end of the translation unit in which they are declared, just as the effect of declarations within the blocks persists to the end of the block. The syntax of external declarations is the same as that of all declarations, except that only at this level may the code for functions be given.

A.10.1 Function Definitions

Function definitions have the form

function-definition:
  declaration-specifiers$_{opt}$ *declarator declaration-list*$_{opt}$ *compound-statement*

The only storage-class specifiers allowed among the declaration specifiers are `extern` or `static`; see Par.A.11.2 for the distinction between them.

A function may return an arithmetic type, a structure, a union, a pointer, or `void`, but not a function or an array. The declarator in a function declaration must specify explicitly that the declared identifier has function type; that is, it must contain one of the forms (see Par.A.8.6.3).

direct-declarator (*parameter-type-list*)
direct-declarator (*identifier-list*$_{opt}$)

where the direct-declarator is an identifier or a parenthesized identifier. In particular, it must not achieve function type by means of a `typedef`.

In the first form, the definition is a new-style function, and its parameters, together with their types, are declared in its parameter type list; the declaration-list following the function's declarator must be absent. Unless the parameter type list consists solely of `void`, showing that the function takes no parameters, each declarator in the parameter type list must contain an identifier. If the parameter type list ends with "`, ...`" then the function may be called with more arguments than parameters; the `va_arg` macro mechanism defined in the standard header `<stdarg.h>` and described in Appendix B must be used to refer to the extra arguments. Variadic functions must have at least one named parameter.

In the second form, the definition is old-style: the identifier list names the parameters, while the declaration list attributes types to them. If no declaration is given for a parameter, its type is taken to be `int`. The declaration list must declare only parameters named in the list, initialization is not permitted, and the only storage-class specifier possible is `register`.

In both styles of function definition, the parameters are understood to be declared just after the beginning of the compound statement constituting the function's body, and thus the same identifiers must not be redeclared there (although they may, like other identifiers, be redeclared in inner blocks). If a parameter is declared to have type "array of *type*," the declaration is adjusted to read "pointer to *type*;" similarly, if a parameter is declared to have type "function

returning *type*," the declaration is adjusted to read ``pointer to function returning *type*." During the call to a function, the arguments are converted as necessary and assigned to the parameters; see Par.A.7.3.2.

> New-style function definitions are new with the ANSI standard. There is also a small change in the details of promotion; the first edition specified that the declarations of `float` parameters were adjusted to read `double`. The difference becomes noticable when a pointer to a parameter is generated within a function.

A complete example of a new-style function definition is

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

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

Here `int` is the declaration specifier; `max(int a, int b, int c)` is the function's declarator, and `{ ... }` is the block giving the code for the function. The corresponding old-style definition would be

```
int max(a, b, c)
int a, b, c;
{
    /* ... */
}
```

where now `int max(a, b, c)` is the declarator, and `int a, b, c;` is the declaration list for the parameters.

A.10.2 External Declarations

External declarations specify the characteristics of objects, functions and other identifiers. The term ``external" refers to their location outside functions, and is not directly connected with the `extern` keyword; the storage class for an externally-declared object may be left empty, or it may be specified as `extern` or `static`.

Several external declarations for the same identifier may exist within the same translation unit if they agree in type and linkage, and if there is at most one definition for the identifier.

Two declarations for an object or function are deemed to agree in type under the rule discussed in Par.A.8.10. In addition, if the declarations differ because one type is an incomplete structure, union, or enumeration type (Par.A.8.3) and the other is the corresponding completed type with the same tag, the types are taken to agree. Moreover, if one type is an incomplete array type (Par.A.8.6.2) and the other is a completed array type, the types, if otherwise identical, are also taken to agree. Finally, if one type specifies an old-style function, and the other an otherwise identical new-style function, with parameter declarations, the types are taken to agree.

If the first external declarator for a function or object includes the `static` specifier, the identifier has *internal linkage*; otherwise it has *external linkage*. Linkage is discussed in Par.11.2.

An external declaration for an object is a definition if it has an initializer. An external object declaration that does not have an initializer, and does not contain the `extern` specifier, is a *tentative definition*. If a definition for an object appears in a translation unit, any tentative definitions are treated merely as redundant declarations. If no definition for the object appears in the translation unit, all its tentative definitions become a single definition with initializer 0.

Each object must have exactly one definition. For objects with internal linkage, this rule applies separately to each translation unit, because internally-linked objects are unique to a translation unit. For objects with external linkage, it applies to the entire program.

> Although the one-definition rule is formulated somewhat differently in the first edition of this book, it is in effect identical to the one stated here. Some implementations relax it by generalizing the notion of tentative definition. In the alternate formulation, which is usual in UNIX systems and recognized as a common extension by the Standard, all the tentative definitions for an externally linked object, throughout all the translation units of the program, are considered together instead of in each translation unit separately. If a definition occurs somewhere in the program, then the tentative definitions become merely declarations, but if no definition appears, then all its tentative definitions become a definition with initializer 0.

A.11 Scope and Linkage

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

Therefore, there are two kinds of scope to consider: first, the *lexical scope* of an identifier which is the region of the program text within which the identifier's characteristics are understood; and second, the scope associated with objects and functions with external linkage, which determines the connections between identifiers in separately compiled translation units.

A.11.1 Lexical Scope

Identifiers fall into several name spaces that do not interfere with one another; the same identifier may be used for different purposes, even in the same scope, if the uses are in different name spaces. These classes are: objects, functions, typedef names, and `enum` constants; labels; tags of structures or unions, and enumerations; and members of each structure or union individually.

> These rules differ in several ways from those described in the first edition of this manual. Labels did not previously have their own name space; tags of structures and unions each had a separate space, and in some implementations enumerations tags did as well; putting different kinds of tags into the same space is a new restriction. The most important departure from the first edition is that each structure or union creates a separate name space for its members, so that the same name may appear in several different structures. This rule has been common practice for several years.

The lexical scope of an object or function identifier in an external declaration begins at the end of its declarator and persists to the end of the translation unit in which it appears. The scope of a parameter of a function definition begins at the start of the block defining the function, and persists through the function; the scope of a parameter in a function declaration ends at the end of the declarator. The scope of an identifier declared at the head of a block begins at the end of its declarator, and persists to the end of the block. The scope of a label is the whole of the function in which it appears. The scope of a structure, union, or enumeration tag, or an enumeration constant, begins at its appearance in a type specifier, and persists to the end of a translation unit (for declarations at the external level) or to the end of the block (for declarations within a function).

If an identifier is explicitly declared at the head of a block, including the block constituting a function, any declaration of the identifier outside the block is suspended until the end of the block.

A.11.2 Linkage

Within a translation unit, all declarations of the same object or function identifier with internal linkage refer to the same thing, and the object or function is unique to that translation unit. All

declarations for the same object or function identifier with external linkage refer to the same thing, and the object or function is shared by the entire program.

As discussed in Par.A.10.2, the first external declaration for an identifier gives the identifier internal linkage if the `static` specifier is used, external linkage otherwise. If a declaration for an identifier within a block does not include the `extern` specifier, then the identifier has no linkage and is unique to the function. If it does include `extern`, and an external declaration for is active in the scope surrounding the block, then the identifier has the same linkage as the external declaration, and refers to the same object or function; but if no external declaration is visible, its linkage is external.

A.12 Preprocessing

A preprocessor performs macro substitution, conditional compilation, and inclusion of named files. Lines beginning with `#`, perhaps preceded by white space, communicate with this preprocessor. The syntax of these lines is independent of the rest of the language; they may appear anywhere and have effect that lasts (independent of scope) until the end of the translation unit. Line boundaries are significant; each line is analyzed individually (bus see Par.A.12.2 for how to adjoin lines). To the preprocessor, a token is any language token, or a character sequence giving a file name as in the `#include` directive (Par.A.12.4); in addition, any character not otherwise defined is taken as a token. However, the effect of white spaces other than space and horizontal tab is undefined within preprocessor lines.

Preprocessing itself takes place in several logically successive phases that may, in a particular implementation, be condensed.

1. First, trigraph sequences as described in Par.A.12.1 are replaced by their equivalents. Should the operating system environment require it, newline characters are introduced between the lines of the source file.

2. Each occurrence of a backslash character `\` followed by a newline is deleted, this splicing lines (Par.A.12.2).

3. The program is split into tokens separated by white-space characters; comments are replaced by a single space. Then preprocessing directives are obeyed, and macros (Pars.A.12.3-A.12.10) are expanded.

4. Escape sequences in character constants and string literals (Pars. A.2.5.2, A.2.6) are replaced by their equivalents; then adjacent string literals are concatenated.

5. The result is translated, then linked together with other programs and libraries, by collecting the necessary programs and data, and connecting external functions and object references to their definitions.

A.12.1 Trigraph Sequences

The character set of C source programs is contained within seven-bit ASCII, but is a superset of the ISO 646-1983 Invariant Code Set. In order to enable programs to be represented in the reduced set, all occurrences of the following trigraph sequences are replaced by the corresponding single character. This replacement occurs before any other processing.

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

No other such replacements occur.

Trigraph sequences are new with the ANSI standard.

A.12.2 Line Splicing

Lines that end with the backslash character \ are folded by deleting the backslash and the following newline character. This occurs before division into tokens.

A.12.3 Macro Definition and Expansion

A control line of the form

```
# define identifier token-sequence
```

causes the preprocessor to replace subsequent instances of the identifier with the given sequence of tokens; leading and trailing white space around the token sequence is discarded. A second `#define` for the same identifier is erroneous unless the second token sequence is identical to the first, where all white space separations are taken to be equivalent.

A line of the form

```
# define identifier (identifier-list) token-sequence
```

where there is no space between the first identifier and the (, is a macro definition with parameters given by the identifier list. As with the first form, leading and trailing white space arround the token sequence is discarded, and the macro may be redefined only with a definition in which the number and spelling of parameters, and the token sequence, is identical.

A control line of the form

```
# undef identifier
```

causes the identifier's preprocessor definition to be forgotten. It is not erroneous to apply `#undef` to an unknown identifier.

When a macro has been defined in the second form, subsequent textual instances of the macro identifier followed by optional white space, and then by (, a sequence of tokens separated by commas, and a) constitute a call of the macro. The arguments of the call are the comma-separated token sequences; commas that are quoted or protected by nested parentheses do not separate arguments. During collection, arguments are not macro-expanded. The number of arguments in the call must match the number of parameters in the definition. After the arguments are isolated, leading and trailing white space is removed from them. Then the token sequence resulting from each argument is substituted for each unquoted occurrence of the corresponding parameter's identifier in the replacement token sequence of the macro. Unless the parameter in the replacement sequence is preceded by #, or preceded or followed by ##, the argument tokens are examined for macro calls, and expanded as necessary, just before insertion.

Two special operators influence the replacement process. First, if an occurrence of a parameter in the replacement token sequence is immediately preceded by #, string quotes (") are placed around the corresponding parameter, and then both the # and the parameter identifier are replaced by the quoted argument. A \ character is inserted before each " or \ character that appears surrounding, or inside, a string literal or character constant in the argument.

Second, if the definition token sequence for either kind of macro contains a ## operator, then just after replacement of the parameters, each ## is deleted, together with any white space on either side, so as to concatenate the adjacent tokens and form a new token. The effect is undefined if invalid tokens are produced, or if the result depends on the order of processing of the ## operators. Also, ## may not appear at the beginning or end of a replacement token sequence.

In both kinds of macro, the replacement token sequence is repeatedly rescanned for more defined identifiers. However, once a given identifier has been replaced in a given expansion, it is not replaced if it turns up again during rescanning; instead it is left unchanged.

Even if the final value of a macro expansion begins with with `#`, it is not taken to be a preprocessing directive.

> The details of the macro-expansion process are described more precisely in the ANSI standard than in the first edition. The most important change is the addition of the # and ## operators, which make quotation and concatenation admissible. Some of the new rules, especially those involving concatenation, are bizarre. (See example below.)

For example, this facility may be used for ``manifest-constants," as in

```
#define TABSIZE 100
int table[TABSIZE];
```

The definition

```
#define ABSDIFF(a, b)  ((a)>(b) ? (a)-(b) : (b)-(a))
```

defines a macro to return the absolute value of the difference between its arguments. Unlike a function to do the same thing, the arguments and returned value may have any arithmetic type or even be pointers. Also, the arguments, which might have side effects, are evaluated twice, once for the test and once to produce the value.

Given the definition

```
#define tempfile(dir)    #dir "%s"
```

the macro call `tempfile(/usr/tmp)` yields

```
"/usr/tmp" "%s"
```

which will subsequently be catenated into a single string. After

```
#define cat(x, y)        x ## y
```

the call `cat(var, 123)` yields `var123`. However, the call `cat(cat(1,2),3)` is undefined: the presence of ## prevents the arguments of the outer call from being expanded. Thus it produces the token string

```
cat  (  1  ,  2  )3
```

and `)3` (the catenation of the last token of the first argument with the first token of the second) is not a legal token. If a second level of macro definition is introduced,

```
#define xcat(x, y)       cat(x,y)
```

things work more smoothly; `xcat(xcat(1, 2), 3)` does produce `123`, because the expansion of `xcat` itself does not involve the ## operator.

Likewise, `ABSDIFF(ABSDIFF(a,b),c)` produces the expected, fully-expanded result.

A.12.4 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 characters in the name *filename* must not include > or newline, and the effect is undefined if it contains any of ", ', \, or /*. The named file is searched for in a sequence of implementation-defined places.

Similarly, a control line of the form

`# include "`*filename*`"`

searches first in association with the original source file (a deliberately implementation-dependent phrase), and if that search fails, then as in the first form. The effect of using `'`, `\`, or `/*` in the filename remains undefined, but `>` is permitted.

Finally, a directive of the form

`#` `include` *token-sequence*

not matching one of the previous forms is interpreted by expanding the token sequence as for normal text; one of the two forms with `<...>` or `"..."` must result, and is then treated as previously described.

`#include` files may be nested.

A.12.5 Conditional Compilation

Parts of a program may be compiled conditionally, according to the following schematic syntax.

preprocessor-conditional:
if-line text elif-parts else-part$_{opt}$ `#endif`

if-line:
`#` `if` *constant-expression*
`#` `ifdef` *identifier*
`#` `ifndef` *identifier*

elif-parts:
elif-line text
elif-parts$_{opt}$

elif-line:
`#` `elif` *constant-expression*

else-part:
else-line text

else-line:
`#else`

Each of the directives (if-line, elif-line, else-line, and `#endif`) appears alone on a line. The constant expressions in `#if` and subsequent `#elif` lines are evaluated in order until an expression with a non-zero value is found; text following a line with a zero value is discarded. The text following the successful directive line is treated normally. ``Text'' here refers to any material, including preprocessor lines, that is not part of the conditional structure; it may be empty. Once a successful `#if` or `#elif` line has been found and its text processed, succeeding `#elif` and `#else` lines, together with their text, are discarded. If all the expressions are zero, and there is an `#else`, the text following the `#else` is treated normally. Text controlled by inactive arms of the conditional is ignored except for checking the nesting of conditionals.

The constant expression in `#if` and `#elif` is subject to ordinary macro replacement. Moreover, any expressions of the form

`defined` *identifier*

or

`defined` (*identifier*)

are replaced, before scanning for macros, by `1L` if the identifier is defined in the preprocessor, and by `0L` if not. Any identifiers remaining after macro expansion are replaced by `0L`. Finally, each integer constant is considered to be suffixed with `L`, so that all arithmetic is taken to be long or unsigned long.

The resulting constant expression (Par.A.7.19) is restricted: it must be integral, and may not contain `sizeof`, a cast, or an enumeration constant.

The control lines

`#ifdef` *identifier*
`#ifndef` *identifier*

are equivalent to

`# if defined` *identifier*
`# if ! defined` *identifier*

respectively.

> `#elif` is new since the first edition, although it has been available is some preprocessors. The `defined` preprocessor operator is also new.

A.12.6 Line Control

For the benefit of other preprocessors that generate C programs, a line in one of the forms

`# line` *constant* "*filename*"
`# line` *constant*

causes the compiler to believe, for purposes of error diagnostics, that the line number of the next source line is given by the decimal integer constant and the current input file is named by the identifier. If the quoted filename is absent, the remembered name does not change. Macros in the line are expanded before it is interpreted.

A.12.7 Error Generation

A preprocessor line of the form

`# error` $token\text{-}sequence_{opt}$

causes the preprocessor to write a diagnostic message that includes the token sequence.

A.12.8 Pragmas

A control line of the form

`# pragma` $token\text{-}sequence_{opt}$

causes the preprocessor to perform an implementation-dependent action. An unrecognized pragma is ignored.

A.12.9 Null directive

A control line of the form

`#`

has no effect.

A.12.10 Predefined names

Several identifiers are predefined, and expand to produce special information. They, and also the preprocessor expansion operator `defined`, may not be undefined or redefined.

`__LINE__` A decimal constant containing the current source line number.

`__FILE__` A string literal containing the name of the file being compiled.

`__DATE__` A string literal containing the date of compilation, in the form `"Mmmm dd yyyy"`

`__TIME__` A string literal containing the time of compilation, in the form `"hh:mm:ss"`

`__STDC__` The constant `1`. It is intended that this identifier be defined to be `1` only in standard-conforming implementations.

`#error` and `#pragma` are new with the ANSI standard; the predefined preprocessor macros are new, but some of them have been available in some implementations.

A.13 Grammar

Below is a recapitulation of the grammar that was given throughout the earlier part of this appendix. It has exactly the same content, but is in different order.

The grammar has undefined terminal symbols *integer-constant*, *character-constant*, *floating-constant*, *identifier*, *string*, and *enumeration-constant*; the `typewriter` style words and symbols are terminals given literally. This grammar can be transformed mechanically into input acceptable for an automatic parser-generator. Besides adding whatever syntactic marking is used to indicate alternatives in productions, it is necessary to expand the ``one of'' constructions, and (depending on the rules of the parser-generator) to duplicate each production with an *opt* symbol, once with the symbol and once without. With one further change, namely deleting the production *typedef-name*: *identifier* and making *typedef-name* a terminal symbol, this grammar is acceptable to the YACC parser-generator. It has only one conflict, generated by the `if-else` ambiguity.

translation-unit:
  external-declaration
  translation-unit external-declaration

external-declaration:
  function-definition
  declaration

function-definition:
  declaration-specifiers$_{opt}$ *declarator declaration-list*$_{opt}$ *compound-statement*

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

declaration-list:
  declaration
  declaration-list declaration

declaration-specifiers:
  storage-class-specifier declaration-specifiers$_{opt}$
  type-specifier declaration-specifiers$_{opt}$
  type-qualifier declaration-specifiers$_{opt}$

storage-class specifier: one of
  `auto register static extern typedef`

type specifier: one of
  `void char short int long float double signed`
  `unsigned` *struct-or-union-specifier enum-specifier typedef-name*

type-qualifier: one of
`const volatile`

struct-or-union-specifier:
struct-or-union identifier$_{opt}$ `{` *struct-declaration-list* `}`
struct-or-union identifier

struct-or-union: one of
`struct union`

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

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

init-declarator:
declarator
declarator `=` *initializer*

struct-declaration:
specifier-qualifier-list struct-declarator-list`;`

specifier-qualifier-list:
type-specifier specifier-qualifier-list$_{opt}$
type-qualifier specifier-qualifier-list$_{opt}$

struct-declarator-list:
struct-declarator
struct-declarator-list `,` *struct-declarator*

struct-declarator:
declarator
declarator$_{opt}$ `:` *constant-expression*

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

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

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

declarator:
pointer$_{opt}$ *direct-declarator*

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

direct-declarator (*parameter-type-list*)
direct-declarator (*identifier-list*$_{opt}$)

pointer:
* *type-qualifier-list*$_{opt}$
* *type-qualifier-list*$_{opt}$ *pointer*

type-qualifier-list:
type-qualifier
type-qualifier-list type-qualifier

parameter-type-list:
parameter-list
parameter-list , . . .

parameter-list:
parameter-declaration
parameter-list , *parameter-declaration*

parameter-declaration:
declaration-specifiers declarator
declaration-specifiers abstract-declarator$_{opt}$

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

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

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

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

abstract-declarator:
pointer
pointer$_{opt}$ *direct-abstract-declarator*

direct-abstract-declarator:
(*abstract-declarator*)
direct-abstract-declarator$_{opt}$ [*constant-expression*$_{opt}$]
direct-abstract-declarator$_{opt}$ (*parameter-type-list*$_{opt}$)

typedef-name:
identifier

statement:
labeled-statement
expression-statement
compound-statement
selection-statement

iteration-statement
jump-statement

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

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

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

statement-list:
statement
statement-list statement

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

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

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

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

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

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

conditional-expression:
logical-OR-expression
logical-OR-expression `?` *expression* `:` *conditional-expression*

constant-expression:
conditional-expression

logical-OR-expression:
logical-AND-expression
logical-OR-expression `||` *logical-AND-expression*

logical-AND-expression:
  inclusive-OR-expression
  logical-AND-expression `&&` *inclusive-OR-expression*

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

exclusive-OR-expression:
  AND-expression
  exclusive-OR-expression `^` *AND-expression*

AND-expression:
  equality-expression
  AND-expression `&` *equality-expression*

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

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

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

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

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

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

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

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

postfix-expression:
primary-expression
postfix-expression[*expression*]
postfix-expression(*argument-expression-list*$_{opt}$)
postfix-expression.*identifier*
postfix-expression`->`+*identifier*
postfix-expression++
postfix-expression`--`

primary-expression:
identifier
constant
string
(*expression*)

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

constant:
integer-constant
character-constant
floating-constant
enumeration-constant

The following grammar for the preprocessor summarizes the structure of control lines, but is not suitable for mechanized parsing. It includes the symbol *text*, which means ordinary program text, non-conditional preprocessor control lines, or complete preprocessor conditional instructions.

control-line:
`#` `define` *identifier token-sequence*
`#` `define` *identifier*(*identifier, ... , identifier*) *token-sequence*
`#` `undef` *identifier*
`#` `include` <*filename*>
`#` `include` "*filename*"
`#` `line` *constant* "*filename*"
`#` `line` *constant*
`#` `error` *token-sequence*$_{opt}$
`#` `pragma` *token-sequence*$_{opt}$
`#`
preprocessor-conditional

preprocessor-conditional:
if-line text elif-parts else-part$_{opt}$ `#endif`

if-line:
`#` `if` *constant-expression*
`#` `ifdef` *identifier*
`#` `ifndef` *identifier*

elif-parts:
  elif-line text
  elif-parts$_{opt}$

elif-line:
  `#` `elif` *constant-expression*

else-part:
  else-line text

else-line:
  `#else`

Appendix B - Standard Library

This appendix is a summary of the library defined by the ANSI standard. The standard library is not part of the C language proper, but an environment that supports standard C will provide the function declarations and type and macro definitions of this library. We have omitted a few functions that are of limited utility or easily synthesized from others; we have omitted multi-byte characters; and we have omitted discussion of locale issues; that is, properties that depend on local language, nationality, or culture.

The functions, types and macros of the standard library are declared in standard *headers*:

```
<assert.h>  <float.h>   <math.h>    <stdarg.h>  <stdlib.h>
<ctype.h>   <limits.h>  <setjmp.h>  <stddef.h>  <string.h>
<errno.h>   <locale.h>  <signal.h>  <stdio.h>   <time.h>
```

A header can be accessed by

`#include` <*header*>

Headers may be included in any order and any number of times. A header must be included outside of any external declaration or definition and before any use of anything it declares. A header need not be a source file.

External identifiers that begin with an underscore are reserved for use by the library, as are all other identifiers that begin with an underscore and an upper-case letter or another underscore.

B.1 Input and Output: <stdio.h>

The input and output functions, types, and macros defined in `<stdio.h>` represent nearly one third of the library.

A *stream* is a source or destination of data that may be associated with a disk or other peripheral. The library supports text streams and binary streams, although on some systems, notably UNIX, these are identical. A text stream is a sequence of lines; each line has zero or more characters and is terminated by `'\n'`. An environment may need to convert a text stream to or from some other representation (such as mapping `'\n'` to carriage return and linefeed). A binary stream is a sequence of unprocessed bytes that record internal data, with the property that if it is written, then read back on the same system, it will compare equal.

A stream is connected to a file or device by *opening* it; the connection is broken by *closing* the stream. Opening a file returns a pointer to an object of type `FILE`, which records whatever information is necessary to control the stream. We will use ``file pointer'' and ``stream'' interchangeably when there is no ambiguity.

When a program begins execution, the three streams `stdin`, `stdout`, and `stderr` are already open.

B.1.1 File Operations

The following functions deal with operations on files. The type `size_t` is the unsigned integral type produced by the `sizeof` operator.

```
FILE *fopen(const char *filename, const char *mode)
```

`fopen` opens the named file, and returns a stream, or `NULL` if the attempt fails. Legal values for `mode` include:

- `"r"` open text file for reading
- `"w"` create text file for writing; discard previous contents if any
- `"a"` append; open or create text file for writing at end of file

"r+" open text file for update (i.e., reading and writing)
"w+" create text file for update, discard previous contents if any
"a+" append; open or create text file for update, writing at end

Update mode permits reading and writing the same file; `fflush` or a file-positioning function must be called between a read and a write or vice versa. If the mode includes `b` after the initial letter, as in `"rb"` or `"w+b"`, that indicates a binary file. Filenames are limited to `FILENAME_MAX` characters. At most `FOPEN_MAX` files may be open at once.

`FILE *freopen(const char *filename, const char *mode, FILE *stream)`
`freopen` opens the file with the specified mode and associates the stream with it. It returns `stream`, or `NULL` if an error occurs. `freopen` is normally used to change the files associated with `stdin`, `stdout`, or `stderr`.

`int fflush(FILE *stream)`
On an output stream, `fflush` causes any buffered but unwritten data to be written; on an input stream, the effect is undefined. It returns `EOF` for a write error, and zero otherwise. `fflush(NULL)` flushes all output streams.

`int fclose(FILE *stream)`
`fclose` flushes any unwritten data for `stream`, discards any unread buffered input, frees any automatically allocated buffer, then closes the stream. It returns `EOF` if any errors occurred, and zero otherwise.

`int remove(const char *filename)`
`remove` removes the named file, so that a subsequent attempt to open it will fail. It returns non-zero if the attempt fails.

`int rename(const char *oldname, const char *newname)`
`rename` changes the name of a file; it returns non-zero if the attempt fails.

`FILE *tmpfile(void)`
`tmpfile` creates a temporary file of mode `"wb+"` that will be automatically removed when closed or when the program terminates normally. `tmpfile` returns a stream, or `NULL` if it could not create the file.

`char *tmpnam(char s[L_tmpnam])`
`tmpnam(NULL)` creates a string that is not the name of an existing file, and returns a pointer to an internal static array. `tmpnam(s)` stores the string in `s` as well as returning it as the function value; `s` must have room for at least `L_tmpnam` characters. `tmpnam` generates a different name each time it is called; at most `TMP_MAX` different names are guaranteed during execution of the program. Note that `tmpnam` creates a name, not a file.

`int setvbuf(FILE *stream, char *buf, int mode, size_t size)`
`setvbuf` controls buffering for the stream; it must be called before reading, writing or any other operation. A `mode` of `_IOFBF` causes full buffering, `_IOLBF` line buffering of text files, and `_IONBF` no buffering. If `buf` is not `NULL`, it will be used as the buffer, otherwise a buffer will be allocated. `size` determines the buffer size. `setvbuf` returns non-zero for any error.

`void setbuf(FILE *stream, char *buf)`
If `buf` is `NULL`, buffering is turned off for the stream. Otherwise, `setbuf` is equivalent to `(void) setvbuf(stream, buf, _IOFBF, BUFSIZ)`.

B.1.2 Formatted Output

The `printf` functions provide formatted output conversion.

```
int fprintf(FILE *stream, const char *format, ...)
```

`fprintf` converts and writes output to `stream` under the control of `format`. The return value is the number of characters written, or negative if an error occurred.

The format string contains two types of objects: ordinary characters, which are copied to the output stream, and conversion specifications, each of which causes conversion and printing of

the next successive argument to `fprintf`. Each conversion specification begins with the character `%` and ends with a conversion character. Between the `%` and the conversion character there may be, in order:

- Flags (in any order), which modify the specification:
    - `-`, which specifies left adjustment of the converted argument in its field.
    - `+`, which specifies that the number will always be printed with a sign.
    - *space*: if the first character is not a sign, a space will be prefixed.
    - `0`: for numeric conversions, specifies padding to the field width with leading zeros.
    - `#`, which specifies an alternate output form. For `o`, the first digit will become zero. For `x` or `X`, `0x` or `0X` will be prefixed to a non-zero result. For `e`, `E`, `f`, `g`, and `G`, the output will always have a decimal point; for `g` and `G`, trailing zeros will not be removed.
- A number specifying a minimum field width. The converted argument will be printed in a field at least this wide, and wider if necessary. If the converted argument has fewer characters than the field width it will be padded on the left (or right, if left adjustment has been requested) to make up the field width. The padding character is normally space, but is `0` if the zero padding flag is present.
- A period, which separates the field width from the precision.
- A number, the precision, that specifies the maximum number of characters to be printed from a string, or the number of digits to be printed after the decimal point for `e`, `E`, or `f` conversions, or the number of significant digits for `g` or `G` conversion, or the number of digits to be printed for an integer (leading `0`s will be added to make up the necessary width).
- A length modifier `h`, `l` (letter ell), or `L`. ``h" indicates that the corresponding argument is to be printed as a `short` or `unsigned short`; ``l" indicates that the argument is a `long` or `unsigned long`, ``L" indicates that the argument is a `long double`.

Width or precision or both may be specified as `*`, in which case the value is computed by converting the next argument(s), which must be `int`.

The conversion characters and their meanings are shown in Table B.1. If the character after the `%` is not a conversion character, the behavior is undefined.

Table B.1 *Printf Conversions*

Character	Argument type; Printed As
`d,i`	`int`; signed decimal notation.
`o`	`int`; unsigned octal notation (without a leading zero).
`x,X`	`unsigned int`; unsigned hexadecimal notation (without a leading `0x` or `0X`), using `abcdef` for `0x` or `ABCDEF` for `0X`.
`u`	`int`; unsigned decimal notation.
`c`	`int`; single character, after conversion to `unsigned char`
`s`	`char *`; characters from the string are printed until a `'\0'` is reached or until the number of characters indicated by the precision have been printed.

`f`	`double`; decimal notation of the form `[-]`*mmm.ddd*, where the number of *d*'s is given by the precision. The default precision is 6; a precision of 0 suppresses the decimal point.
`e,E`	`double`; decimal notation of the form `[-]`*m.dddddd*`e+/-`*xx* or `[-]`*m.dddddd*`E+/-`*xx*, where the number of *d*'s is specified by the precision. The default precision is 6; a precision of 0 suppresses the decimal point.
`g,G`	`double`; `%e` or `%E` is used if the exponent is less than -4 or greater than or equal to the precision; otherwise `%f` is used. Trailing zeros and a trailing decimal point are not printed.
`p`	`void *`; print as a pointer (implementation-dependent representation).
`n`	`int *`; the number of characters written so far by this call to `printf` is *written into* the argument. No argument is converted.
`%`	no argument is converted; print a %

```
int printf(const char *format, ...)
```

`printf(...)` is equivalent to `fprintf(stdout, ...)`.

```
int sprintf(char *s, const char *format, ...)
```

`sprintf` is the same as `printf` except that the output is written into the string `s`, terminated with `'\0'`. `s` must be big enough to hold the result. The return count does not include the `'\0'`.

```
int vprintf(const char *format, va_list arg)
int vfprintf(FILE *stream, const char *format, va_list arg)
int vsprintf(char *s, const char *format, va_list arg)
```

The functions `vprintf`, `vfprintf`, and `vsprintf` are equivalent to the corresponding `printf` functions, except that the variable argument list is replaced by `arg`, which has been initialized by the `va_start` macro and perhaps `va_arg` calls. See the discussion of `<stdarg.h>` in Section B.7.

B.1.3 Formatted Input

The `scanf` function deals with formatted input conversion.

```
int fscanf(FILE *stream, const char *format, ...)
```

`fscanf` reads from `stream` under control of `format`, and assigns converted values through subsequent arguments, *each of which must be a pointer*. It returns when `format` is exhausted. `fscanf` returns `EOF` if end of file or an error occurs before any conversion; otherwise it returns the number of input items converted and assigned.

The format string usually contains conversion specifications, which are used to direct interpretation of input. The format string may contain:

- Blanks or tabs, which are not ignored.
- Ordinary characters (not %), which are expected to match the next non-white space character of the input stream.
- Conversion specifications, consisting of a `%`, an optional assignment suppression character `*`, an optional number specifying a maximum field width, an optional `h`, `l`, or `L` indicating the width of the target, and a conversion character.

A conversion specification determines the conversion of the next input field. Normally the result is placed in the variable pointed to by the corresponding argument. If assignment suppression is indicated by `*`, as in `%*s`, however, the input field is simply skipped; no assignment is made. An input field is defined as a string of non-white space characters; it extends either to the next white space character or until the field width, if specified, is exhausted. This implies that `scanf` will read across line boundaries to find its input, since

newlines are white space. (White space characters are blank, tab, newline, carriage return, vertical tab, and formfeed.)

The conversion character indicates the interpretation of the input field. The corresponding argument must be a pointer. The legal conversion characters are shown in Table B.2.

The conversion characters `d`, `i`, `n`, `o`, `u`, and `x` may be preceded by `h` if the argument is a pointer to `short` rather than int, or by `l` (letter ell) if the argument is a pointer to `long`. The conversion characters `e`, `f`, and `g` may be preceded by `l` if a pointer to `double` rather than `float` is in the argument list, and by `L` if a pointer to a `long double`.

Table B.2 *Scanf Conversions*

Character	Input Data; Argument type
`d`	decimal integer; `int*`
`i`	integer; `int*`. The integer may be in octal (leading `0`) or hexadecimal (leading `0x` or `0X`).
`o`	octal integer (with or without leading zero); `int *`.
`u`	unsigned decimal integer; `unsigned int *`.
`x`	hexadecimal integer (with or without leading `0x` or `0X`); `int*`.
`c`	characters; `char*`. The next input characters are placed in the indicated array, up to the number given by the width field; the default is 1. No `'\0'` is added. The normal skip over white space characters is suppressed in this case; to read the next non-white space character, use `%1s`.
`s`	string of non-white space characters (not quoted); `char *`, pointing to an array of characters large enough to hold the string and a terminating `'\0'` that will be added.
`e,f,g`	floating-point number; `float *`. The input format for `float`'s is an optional sign, a string of numbers possibly containing a decimal point, and an optional exponent field containing an `E` or `e` followed by a possibly signed integer.
`p`	pointer value as printed by `printf("%p");`, `void *`.
`n`	writes into the argument the number of characters read so far by this call; `int *`. No input is read. The converted item count is not incremented.
`[...]`	matches the longest non-empty string of input characters from the set between brackets; `char *`. A `'\0'` is added. `[]...]` includes `]` in the set.
`[^...]`	matches the longest non-empty string of input characters *not* from the set between brackets; `char *`. A `'\0'` is added. `[^]...]` includes `]` in the set.
`%`	literal %; no assignment is made.

`int scanf(const char *format, ...)`

`scanf(...)` is identical to `fscanf(stdin, ...)`.

`int sscanf(const char *s, const char *format, ...)`

`sscanf(s, ...)` is equivalent to `scanf(...)` except that the input characters are taken from the string `s`.

B.1.4 Character Input and Output Functions

`int fgetc(FILE *stream)`

`fgetc` returns the next character of `stream` as an `unsigned char` (converted to an `int`), or `EOF` if end of file or error occurs.

`char *fgets(char *s, int n, FILE *stream)`

`fgets` reads at most the next `n-1` characters into the array `s`, stopping if a newline is encountered; the newline is included in the array, which is terminated by `'\0'`. `fgets` returns `s`, or `NULL` if end of file or error occurs.

`int fputc(int c, FILE *stream)`
: `fputc` writes the character `c` (converted to an `unsigend char`) on `stream`. It returns the character written, or `EOF` for error.

`int fputs(const char *s, FILE *stream)`
: `fputs` writes the string `s` (which need not contain `\n`) on `stream`; it returns non-negative, or `EOF` for an error.

`int getc(FILE *stream)`
: `getc` is equivalent to `fgetc` except that if it is a macro, it may evaluate `stream` more than once.

`int getchar(void)`
: `getchar` is equivalent to `getc(stdin)`.

`char *gets(char *s)`
: `gets` reads the next input line into the array `s`; it replaces the terminating newline with `'\0'`. It returns `s`, or `NULL` if end of file or error occurs.

`int putc(int c, FILE *stream)`
: `putc` is equivalent to `fputc` except that if it is a macro, it may evaluate `stream` more than once.

`int putchar(int c)`
: `putchar(c)` is equivalent to `putc(c,stdout)`.

`int puts(const char *s)`
: `puts` writes the string `s` and a newline to `stdout`. It returns `EOF` if an error occurs, non-negative otherwise.

`int ungetc(int c, FILE *stream)`
: `ungetc` pushes `c` (converted to an `unsigned char`) back onto `stream`, where it will be returned on the next read. Only one character of pushback per stream is guaranteed. `EOF` may not be pushed back. `ungetc` returns the character pushed back, or `EOF` for error.

B.1.5 Direct Input and Output Functions

`size_t fread(void *ptr, size_t size, size_t nobj, FILE *stream)`
: `fread` reads from `stream` into the array `ptr` at most `nobj` objects of size `size`. `fread` returns the number of objects read; this may be less than the number requested. `feof` and `ferror` must be used to determine status.

`size_t fwrite(const void *ptr, size_t size, size_t nobj, FILE *stream)`
: `fwrite` writes, from the array `ptr`, `nobj` objects of size `size` on `stream`. It returns the number of objects written, which is less than `nobj` on error.

B.1.6 File Positioning Functions

`int fseek(FILE *stream, long offset, int origin)`
: `fseek` sets the file position for `stream`; a subsequent read or write will access data beginning at the new position. For a binary file, the position is set to `offset` characters from `origin`, which may be `SEEK_SET` (beginning), `SEEK_CUR` (current position), or `SEEK_END` (end of file). For a text stream, `offset` must be zero, or a value returned by `ftell` (in which case `origin` must be `SEEK_SET`). `fseek` returns non-zero on error.

`long ftell(FILE *stream)`
: `ftell` returns the current file position for `stream`, or `-1` on error.

`void rewind(FILE *stream)`
: `rewind(fp)` is equivalent to `fseek(fp, 0L, SEEK_SET); clearerr(fp)`.

`int fgetpos(FILE *stream, fpos_t *ptr)`
: `fgetpos` records the current position in `stream` in `*ptr`, for subsequent use by `fsetpos`. The type `fpos_t` is suitable for recording such values. `fgetpos` returns non-zero on error.

`int fsetpos(FILE *stream, const fpos_t *ptr)`

`fsetpos` positions `stream` at the position recorded by `fgetpos` in `*ptr`. `fsetpos` returns non-zero on error.

B.1.7 Error Functions

Many of the functions in the library set status indicators when error or end of file occur. These indicators may be set and tested explicitly. In addition, the integer expression `errno` (declared in `<errno.h>`) may contain an error number that gives further information about the most recent error.

`void clearerr(FILE *stream)`
: `clearerr` clears the end of file and error indicators for `stream`.

`int feof(FILE *stream)`
: `feof` returns non-zero if the end of file indicator for `stream` is set.

`int ferror(FILE *stream)`
: `ferror` returns non-zero if the error indicator for `stream` is set.

`void perror(const char *s)`
: `perror(s)` prints `s` and an implementation-defined error message corresponding to the integer in `errno`, as if by

```
fprintf(stderr, "%s: %s\n", s, "error message");
```

See `strerror` in Section B.3.

B.2 Character Class Tests: <ctype.h>

The header `<ctype.h>` declares functions for testing characters. For each function, the argument list is an `int`, whose value must be `EOF` or representable as an `unsigned char`, and the return value is an `int`. The functions return non-zero (true) if the argument `c` satisfies the condition described, and zero if not.

`isalnum(c)`	`isalpha(c)` or `isdigit(c)` is true
`isalpha(c)`	`isupper(c)` or `islower(c)` is true
`iscntrl(c)`	control character
`isdigit(c)`	decimal digit
`isgraph(c)`	printing character except space
`islower(c)`	lower-case letter
`isprint(c)`	printing character including space
`ispunct(c)`	printing character except space or letter or digit
`isspace(c)`	space, formfeed, newline, carriage return, tab, vertical tab
`isupper(c)`	upper-case letter
`isxdigit(c)`	hexadecimal digit

In the seven-bit ASCII character set, the printing characters are `0x20 (' ')` to `0x7E ('~')`; the control characters are 0 `NUL` to `0x1F` (US), and `0x7F` (DEL).

In addition, there are two functions that convert the case of letters:

`int tolower(c)`	convert `c` to lower case
`int toupper(c)`	convert `c` to upper case

If `c` is an upper-case letter, `tolower(c)` returns the corresponding lower-case letter, `toupper(c)` returns the corresponding upper-case letter; otherwise it returns `c`.

B.3 String Functions: <string.h>

There are two groups of string functions defined in the header `<string.h>`. The first have names beginning with `str`; the second have names beginning with `mem`. Except for `memmove`, the behavior is undefined if copying takes place between overlapping objects. Comparison functions treat arguments as `unsigned char` arrays.

In the following table, variables s and t are of type char *; cs and ct are of type const char *; n is of type size_t; and c is an int converted to char.

char *strcpy(s,ct)	copy string ct to string s, including '\0'; return s.
char *strncpy(s,ct,n)	copy at most n characters of string ct to s; return s. Pad with '\0's if ct has fewer than n characters.
char *strcat(s,ct)	concatenate string ct to end of string s; return s.
char *strncat(s,ct,n)	concatenate at most n characters of string ct to string s, terminate s with '\0'; return s.
int strcmp(cs,ct)	compare string cs to string ct, return <0 if cs<ct, 0 if cs==ct, or >0 if cs>ct.
int strncmp(cs,ct,n)	compare at most n characters of string cs to string ct; return <0 if cs<ct, 0 if cs==ct, or >0 if cs>ct.
char *strchr(cs,c)	return pointer to first occurrence of c in cs or NULL if not present.
char *strrchr(cs,c)	return pointer to last occurrence of c in cs or NULL if not present.
size_t strspn(cs,ct)	return length of prefix of cs consisting of characters in ct.
size_t strcspn(cs,ct)	return length of prefix of cs consisting of characters *not* in ct.
char *strpbrk(cs,ct)	return pointer to first occurrence in string cs of any character string ct, or NULL if not present.
char *strstr(cs,ct)	return pointer to first occurrence of string ct in cs, or NULL if not present.
size_t strlen(cs)	return length of cs.
char *strerror(n)	return pointer to implementation-defined string corresponding to error n.
char *strtok(s,ct)	strtok searches s for tokens delimited by characters from ct; see below.

A sequence of calls of strtok(s,ct) splits s into tokens, each delimited by a character from ct. The first call in a sequence has a non-NULL s, it finds the first token in s consisting of characters not in ct; it terminates that by overwriting the next character of s with '\0' and returns a pointer to the token. Each subsequent call, indicated by a NULL value of s, returns the next such token, searching from just past the end of the previous one. strtok returns NULL when no further token is found. The string ct may be different on each call.

The mem... functions are meant for manipulating objects as character arrays; the intent is an interface to efficient routines. In the following table, s and t are of type void *; cs and ct are of type const void *; n is of type size_t; and c is an int converted to an unsigned char.

void *memcpy(s,ct,n)	copy n characters from ct to s, and return s.
void *memmove(s,ct,n)	same as memcpy except that it works even if the objects overlap.
int memcmp(cs,ct,n)	compare the first n characters of cs with ct; return as with strcmp.
void *memchr(cs,c,n)	return pointer to first occurrence of character c in cs, or NULL if not present among the first n characters.
void *memset(s,c,n)	place character c into first n characters of s, return s.

B.4 Mathematical Functions: <math.h>

The header <math.h> declares mathematical functions and macros.

The macros EDOM and ERANGE (found in <errno.h>) are non-zero integral constants that are used to signal domain and range errors for the functions; HUGE_VAL is a positive double value. A *domain error* occurs if an argument is outside the domain over which the function is defined. On a domain error, errno is set to EDOM; the return value is implementation-defined. A *range error* occurs if the result of the function cannot be represented as a double. If the

result overflows, the function returns `HUGE_VAL` with the right sign, and `errno` is set to `ERANGE`. If the result underflows, the function returns zero; whether `errno` is set to `ERANGE` is implementation-defined.

In the following table, `x` and `y` are of type `double`, `n` is an `int`, and all functions return `double`. Angles for trigonometric functions are expressed in radians.

`sin(x)`	sine of *x*
`cos(x)`	cosine of *x*
`tan(x)`	tangent of *x*
`asin(x)`	$\sin^{-1}(x)$ in range [-pi/2,pi/2], x in [-1,1].
`acos(x)`	$\cos^{-1}(x)$ in range [0,pi], x in [-1,1].
`atan(x)`	$\tan^{-1}(x)$ in range [-pi/2,pi/2].
`atan2(y,x)`	$\tan^{-1}(y/x)$ in range [-pi,pi].
`sinh(x)`	hyperbolic sine of *x*
`cosh(x)`	hyperbolic cosine of *x*
`tanh(x)`	hyperbolic tangent of *x*
`exp(x)`	exponential function e^x
`log(x)`	natural logarithm ln(x), $x>0$.
`log10(x)`	base 10 logarithm $\log_{10}(x)$, $x>0$.
`pow(x,y)`	x^y. A domain error occurs if $x=0$ and $y<=0$, or if $x<0$ and y is not an integer.
`sqrt(x)`	sqare root of x, x>=0.
`ceil(x)`	smallest integer not less than `x`, as a `double`.
`floor(x)`	largest integer not greater than `x`, as a `double`.
`fabs(x)`	absolute value |x|
`ldexp(x,n)`	$x*2^n$
`frexp(x, int *ip)`	splits *x* into a normalized fraction in the interval [1/2,1) which is returned, and a power of 2, which is stored in `*exp`. If *x* is zero, both parts of the result are zero.
`modf(x, double *ip)`	splits *x* into integral and fractional parts, each with the same sign as *x*. It stores the integral part in `*ip`, and returns the fractional part.
`fmod(x,y)`	floating-point remainder of *x/y*, with the same sign as *x*. If *y* is zero, the result is implementation-defined.

B.5 Utility Functions: <stdlib.h>

The header `<stdlib.h>` declares functions for number conversion, storage allocation, and similar tasks.

`double atof(const char *s)`

`atof` converts `s` to `double`; it is equivalent to `strtod(s, (char**)NULL)`.

`int atoi(const char *s)`

converts `s` to `int`; it is equivalent to `(int)strtol(s, (char**)NULL, 10)`.

`long atol(const char *s)`

converts `s` to `long`; it is equivalent to `strtol(s, (char**)NULL, 10)`.

`double strtod(const char *s, char **endp)`

`strtod` converts the prefix of `s` to `double`, ignoring leading white space; it stores a pointer to any unconverted suffix in `*endp` unless `endp` is `NULL`. If the answer would overflow, `HUGE_VAL` is returned with the proper sign; if the answer would underflow, zero is returned. In either case `errno` is set to `ERANGE`.

`long strtol(const char *s, char **endp, int base)`

`strtol` converts the prefix of `s` to `long`, ignoring leading white space; it stores a pointer to any unconverted suffix in `*endp` unless `endp` is `NULL`. If `base` is between 2 and 36, conversion is done assuming that the input is written in that base. If `base` is zero, the base is 8, 10, or 16; leading 0 implies octal and leading `0x` or `0X` hexadecimal.

Letters in either case represent digits from 10 to base-1; a leading 0x or 0X is permitted in base 16. If the answer would overflow, LONG_MAX or LONG_MIN is returned, depending on the sign of the result, and errno is set to ERANGE.

```
unsigned long strtoul(const char *s, char **endp, int base)
```

strtoul is the same as strtol except that the result is unsigned long and the error value is ULONG_MAX.

```
int rand(void)
```

rand returns a pseudo-random integer in the range 0 to RAND_MAX, which is at least 32767.

```
void srand(unsigned int seed)
```

srand uses seed as the seed for a new sequence of pseudo-random numbers. The initial seed is 1.

```
void *calloc(size_t nobj, size_t size)
```

calloc returns a pointer to space for an array of nobj objects, each of size size, or NULL if the request cannot be satisfied. The space is initialized to zero bytes.

```
void *malloc(size_t size)
```

malloc returns a pointer to space for an object of size size, or NULL if the request cannot be satisfied. The space is uninitialized.

```
void *realloc(void *p, size_t size)
```

realloc changes the size of the object pointed to by p to size. The contents will be unchanged up to the minimum of the old and new sizes. If the new size is larger, the new space is uninitialized. realloc returns a pointer to the new space, or NULL if the request cannot be satisfied, in which case *p is unchanged.

```
void free(void *p)
```

free deallocates the space pointed to by p; it does nothing if p is NULL. p must be a pointer to space previously allocated by calloc, malloc, or realloc.

```
void abort(void)
```

abort causes the program to terminate abnormally, as if by raise(SIGABRT).

```
void exit(int status)
```

exit causes normal program termination. atexit functions are called in reverse order of registration, open files are flushed, open streams are closed, and control is returned to the environment. How status is returned to the environment is implementation-dependent, but zero is taken as successful termination. The values EXIT_SUCCESS and EXIT_FAILURE may also be used.

```
int atexit(void (*fcn)(void))
```

atexit registers the function fcn to be called when the program terminates normally; it returns non-zero if the registration cannot be made.

```
int system(const char *s)
```

system passes the string s to the environment for execution. If s is NULL, system returns non-zero if there is a command processor. If s is not NULL, the return value is implementation-dependent.

```
char *getenv(const char *name)
```

getenv returns the environment string associated with name, or NULL if no string exists. Details are implementation-dependent.

```
void *bsearch(const void *key, const void *base,
              size_t n, size_t size,
              int (*cmp)(const void *keyval, const void *datum))
```

bsearch searches base[0]...base[n-1] for an item that matches *key. The function cmp must return negative if its first argument (the search key) is less than its second (a table entry), zero if equal, and positive if greater. Items in the array base must be in ascending order. bsearch returns a pointer to a matching item, or NULL if none exists.

```
void qsort(void *base, size_t n, size_t size,
```

```
int (*cmp)(const void *, const void *))
```

`qsort` sorts into ascending order an array `base[0]...base[n-1]` of objects of size `size`. The comparison function `cmp` is as in `bsearch`.

```
int abs(int n)
```

`abs` returns the absolute value of its `int` argument.

```
long labs(long n)
```

`labs` returns the absolute value of its `long` argument.

```
div_t div(int num, int denom)
```

`div` computes the quotient and remainder of `num/denom`. The results are stored in the `int` members `quot` and `rem` of a structure of type `div_t`.

```
ldiv_t ldiv(long num, long denom)
```

`ldiv` computes the quotient and remainder of `num/denom`. The results are stored in the `long` members `quot` and `rem` of a structure of type `ldiv_t`.

B.6 Diagnostics: <assert.h>

The `assert` macro is used to add diagnostics to programs:

```
void assert(int expression)
```

If *expression* is zero when

```
assert(expression)
```

is executed, the `assert` macro will print on `stderr` a message, such as

`Assertion failed:` *expression*, `file` *filename*, `line` *nnn*

It then calls `abort` to terminate execution. The source filename and line number come from the preprocessor macros `__FILE__` and `__LINE__`.

If `NDEBUG` is defined at the time `<assert.h>` is included, the assert macro is ignored.

B.7 Variable Argument Lists: <stdarg.h>

The header `<stdarg.h>` provides facilities for stepping through a list of function arguments of unknown number and type.

Suppose `lastarg` is the last named parameter of a function `f` with a variable number of arguments. Then declare within `f` a variable of type `va_list` that will point to each argument in turn:

```
va_list ap;
```

`ap` must be initialized once with the macro `va_start` before any unnamed argument is accessed:

```
va_start(va_list ap, lastarg);
```

Thereafter, each execution of the macro `va_arg` will produce a value that has the type and value of the next unnamed argument, and will also modify `ap` so the next use of `va_arg` returns the next argument:

```
type va_arg(va_list ap, type);
```

The macro

```
void va_end(va_list ap);
```

must be called once after the arguments have been processed but before `f` is exited.

B.8 Non-local Jumps: <setjmp.h>

The declarations in `<setjmp.h>` provide a way to avoid the normal function call and return sequence, typically to permit an immediate return from a deeply nested function call.

`int setjmp(jmp_buf env)`

The macro `setjmp` saves state information in `env` for use by `longjmp`. The return is zero from a direct call of `setjmp`, and non-zero from a subsequent call of `longjmp`. A call to `setjmp` can only occur in certain contexts, basically the test of `if`, `switch`, and loops, and only in simple relational expressions.

```
if (setjmp(env) == 0)
    /* get here on direct call */
else
    /* get here by calling longjmp */
```

`void longjmp(jmp_buf env, int val)`

`longjmp` restores the state saved by the most recent call to `setjmp`, using the information saved in `env`, and execution resumes as if the `setjmp` function had just executed and returned the non-zero value `val`. The function containing the `setjmp` must not have terminated. Accessible objects have the values they had at the time `longjmp` was called, except that non-`volatile` automatic variables in the function calling `setjmp` become undefined if they were changed after the `setjmp` call.

B.9 Signals: <signal.h>

The header `<signal.h>` provides facilities for handling exceptional conditions that arise during execution, such as an interrupt signal from an external source or an error in execution.

`void (*signal(int sig, void (*handler)(int)))(int)`

`signal` determines how subsequent signals will be handled. If `handler` is `SIG_DFL`, the implementation-defined default behavior is used, if it is `SIG_IGN`, the signal is ignored; otherwise, the function pointed to by `handler` will be called, with the argument of the type of signal. Valid signals include

`SIGABRT`	abnormal termination, e.g., from `abort`
`SIGFPE`	arithmetic error, e.g., zero divide or overflow
`SIGILL`	illegal function image, e.g., illegal instruction
`SIGINT`	interactive attention, e.g., interrupt
`SIGSEGV`	illegal storage access, e.g., access outside memory limits
`SIGTERM`	termination request sent to this program

`signal` returns the previous value of `handler` for the specific signal, or `SIG_ERR` if an error occurs.

When a signal `sig` subsequently occurs, the signal is restored to its default behavior; then the signal-handler function is called, as if by `(*handler)(sig)`. If the handler returns, execution will resume where it was when the signal occurred.

The initial state of signals is implementation-defined.

`int raise(int sig)`

`raise` sends the signal `sig` to the program; it returns non-zero if unsuccessful.

B.10 Date and Time Functions: <time.h>

The header `<time.h>` declares types and functions for manipulating date and time. Some functions process *local time*, which may differ from calendar time, for example because of time

zone. `clock_t` and `time_t` are arithmetic types representing times, and `struct tm` holds the components of a calendar time:

`int tm_sec;`	seconds after the minute (0,61)
`int tm_min;`	minutes after the hour (0,59)
`int tm_hour;`	hours since midnight (0,23)
`int tm_mday;`	day of the month (1,31)
`int tm_mon;`	months *since* January (0,11)
`int tm_year;`	years since 1900
`int tm_wday;`	days since Sunday (0,6)
`int tm_yday;`	days since January 1 (0,365)
`int tm_isdst;`	Daylight Saving Time flag

`tm_isdst` is positive if Daylight Saving Time is in effect, zero if not, and negative if the information is not available.

`clock_t clock(void)`
: `clock` returns the processor time used by the program since the beginning of execution, or `-1` if unavailable. `clock()/CLK_PER_SEC` is a time in seconds.

`time_t time(time_t *tp)`
: `time` returns the current calendar time or `-1` if the time is not available. If `tp` is not `NULL`, the return value is also assigned to `*tp`.

`double difftime(time_t time2, time_t time1)`
: `difftime` returns `time2-time1` expressed in seconds.

`time_t mktime(struct tm *tp)`
: `mktime` converts the local time in the structure `*tp` into calendar time in the same representation used by `time`. The components will have values in the ranges shown. `mktime` returns the calendar time or `-1` if it cannot be represented.

The next four functions return pointers to static objects that may be overwritten by other calls.

`char *asctime(const struct tm *tp)`
: asctime</tt< converts the time in the structure *tp into a string of the form

```
Sun Jan  3 15:14:13 1988\n\0
```

`char *ctime(const time_t *tp)`
: ctime converts the calendar time *tp to local time; it is equivalent to

```
asctime(localtime(tp))
```

`struct tm *gmtime(const time_t *tp)`
: gmtime converts the calendar time *tp into Coordinated Universal Time (UTC). It returns NULL if UTC is not available. The name gmtime has historical significance.

`struct tm *localtime(const time_t *tp)`
: localtime converts the calendar time *tp into local time.

`size_t strftime(char *s, size_t smax, const char *fmt, const struct tm *tp)`
: strftime formats date and time information from *tp into s according to fmt, which is analogous to a printf format. Ordinary characters (including the terminating '\0') are copied into s. Each %c is replaced as described below, using values appropriate for the local environment. No more than smax characters are placed into s. strftime returns the number of characters, excluding the '\0', or zero if more than smax characters were produced.

  - %a abbreviated weekday name.
  - %A full weekday name.
  - %b abbreviated month name.
  - %B full month name.
  - %c local date and time representation.
  - %d day of the month (`01-31`).
  - %H hour (24-hour clock) (`00-23`).

%I	hour (12-hour clock) (01-12).
%j	day of the year (001-366).
%m	month (01-12).
%M	minute (00-59).
%p	local equivalent of AM or PM.
%S	second (00-61).
%U	week number of the year (Sunday as 1st day of week) (00-53).
%w	weekday (0-6, Sunday is 0).
%W	week number of the year (Monday as 1st day of week) (00-53).
%x	local date representation.
%X	local time representation.
%y	year without century (00-99).
%Y	year with century.
%Z	time zone name, if any.
%%	%

B.11 Implementation-defined Limits: <limits.h> and <float.h>

The header <limits.h> defines constants for the sizes of integral types. The values below are acceptable minimum magnitudes; larger values may be used.

CHAR_BIT	8	bits in a char
CHAR_MAX	UCHAR_MAX *or* SCHAR_MAX	maximum value of char
CHAR_MIN	0 *or* SCHAR_MIN	maximum value of char
INT_MAX	32767	maximum value of int
INT_MIN	-32767	minimum value of int
LONG_MAX	2147483647	maximum value of long
LONG_MIN	-2147483647	minimum value of long
SCHAR_MAX	+127	maximum value of signed char
SCHAR_MIN	-127	minimum value of signed char
SHRT_MAX	+32767	maximum value of short
SHRT_MIN	-32767	minimum value of short
UCHAR_MAX	255	maximum value of unsigned char
UINT_MAX	65535	maximum value of unsigned int
ULONG_MAX	4294967295	maximum value of unsigned long
USHRT_MAX	65535	maximum value of unsigned short

The names in the table below, a subset of <float.h>, are constants related to floating-point arithmetic. When a value is given, it represents the minimum magnitude for the corresponding quantity. Each implementation defines appropriate values.

FLT_RADIX	2	radix of exponent, representation, e.g., 2, 16
FLT_ROUNDS		floating-point rounding mode for addition
FLT_DIG	6	decimal digits of precision
FLT_EPSILON	1E-5	smallest number *x* such that 1.0+x != 1.0
FLT_MANT_DIG		number of base FLT_RADIX in mantissa
FLT_MAX	1E+37	maximum floating-point number
FLT_MAX_EXP		maximum *n* such that $\texttt{FLT\_RADIX}^{n-1}$ is representable
FLT_MIN	1E-37	minimum normalized floating-point number
FLT_MIN_EXP		minimum *n* such that 10^n is a normalized number
DBL_DIG	10	decimal digits of precision
DBL_EPSILON	1E-9	smallest number *x* such that 1.0+x != 1.0
DBL_MANT_DIG		number of base FLT_RADIX in mantissa

`DBL_MAX`	`1E+37`	maximum `double` floating-point number
`DBL_MAX_EXP`		maximum *n* such that `FLT_RADIX`$^{n-1}$ is representable
`DBL_MIN`	`1E-37`	minimum normalized `double` floating-point number
`DBL_MIN_EXP`		minimum *n* such that 10^n is a normalized number

Appendix C - Summary of Changes

Since the publication of the first edition of this book, the definition of the C language has undergone changes. Almost all were extensions of the original language, and were carefully designed to remain compatible with existing practice; some repaired ambiguities in the original description; and some represent modifications that change existing practice. Many of the new facilities were announced in the documents accompanying compilers available from AT&T, and have subsequently been adopted by other suppliers of C compilers. More recently, the ANSI committee standardizing the language incorporated most of the changes, and also introduced other significant modifications. Their report was in part participated by some commercial compilers even before issuance of the formal C standard.

This Appendix summarizes the differences between the language defined by the first edition of this book, and that expected to be defined by the final standard. It treats only the language itself, not its environment and library; although these are an important part of the standard, there is little to compare with, because the first edition did not attempt to prescribe an environment or library.

- Preprocessing is more carefully defined in the Standard than in the first edition, and is extended: it is explicitly token based; there are new operators for concatenation of tokens (`##`), and creation of strings (`#`); there are new control lines like `#elif` and `#pragma`; redeclaration of macros by the same token sequence is explicitly permitted; parameters inside strings are no longer replaced. Splicing of lines by `\` is permitted everywhere, not just in strings and macro definitions. See Par.A.12.

- The minimum significance of all internal identifiers increased to 31 characters; the smallest mandated significance of identifiers with external linkage remains 6 monocase letters. (Many implementations provide more.)

- Trigraph sequences introduced by `??` allow representation of characters lacking in some character sets. Escapes for `#\^[]{}|~` are defined, see Par.A.12.1. Observe that the introduction of trigraphs may change the meaning of strings containing the sequence `??`.

- New keywords (`void`, `const`, `volatile`, `signed`, `enum`) are introduced. The stillborn `entry` keyword is withdrawn.

- New escape sequences, for use within character constants and string literals, are defined. The effect of following `\` by a character not part of an approved escape sequence is undefined. See Par.A.2.5.2.

- Everyone's favorite trivial change: `8` and `9` are not octal digits.

- The standard introduces a larger set of suffixes to make the type of constants explicit: `U` or `L` for integers, `F` or `L` for floating. It also refines the rules for the type of unsiffixed constants (Par.A.2.5).

- Adjacent string literals are concatenated.

- There is a notation for wide-character string literals and character constants; see Par.A.2.6.

- Characters as well as other types, may be explicitly declared to carry, or not to carry, a sign by using the keywords `signed` or `unsigned`. The locution `long float` as a

synonym for `double` is withdrawn, but `long double` may be used to declare an extra-precision floating quantity.

- For some time, type `unsigned char` has been available. The standard introduces the `signed` keyword to make signedness explicit for `char` and other integral objects.
- The `void` type has been available in most implementations for some years. The Standard introduces the use of the `void *` type as a generic pointer type; previously `char *` played this role. At the same time, explicit rules are enacted against mixing pointers and integers, and pointers of different type, without the use of casts.
- The Standard places explicit minima on the ranges of the arithmetic types, and mandates headers (`<limits.h>` and `<float.h>`) giving the characteristics of each particular implementation.
- Enumerations are new since the first edition of this book.
- The Standard adopts from C++ the notion of type qualifier, for example `const` (Par.A.8.2).
- Strings are no longer modifiable, and so may be placed in read-only memory.
- The ``usual arithmetic conversions'' are changed, essentially from ``for integers, `unsigned` always wins; for floating point, always use `double`'' to ``promote to the smallest capacious-enough type.'' See Par.A.6.5.
- The old assignment operators like `=+` are truly gone. Also, assignment operators are now single tokens; in the first edition, they were pairs, and could be separated by white space.
- A compiler's license to treat mathematically associative operators as computationally associative is revoked.
- A unary `+` operator is introduced for symmetry with unary `-`.
- A pointer to a function may be used as a function designator without an explicit `*` operator. See Par.A.7.3.2.
- Structures may be assigned, passed to functions, and returned by functions.
- Applying the address-of operator to arrays is permitted, and the result is a pointer to the array.
- The `sizeof` operator, in the first edition, yielded type `int`; subsequently, many implementations made it `unsigned`. The Standard makes its type explicitly implementation-dependent, but requires the type, `size_t`, to be defined in a standard header (`<stddef.h>`). A similar change occurs in the type (`ptrdiff_t`) of the difference between pointers. See Par.A.7.4.8 and Par.A.7.7.
- The address-of operator `&` may not be applied to an object declared `register`, even if the implementation chooses not to keep the object in a register.
- The type of a shift expression is that of the left operand; the right operand can't promote the result. See Par.A.7.8.
- The Standard legalizes the creation of a pointer just beyond the end of an array, and allows arithmetic and relations on it; see Par.A.7.7.

- The Standard introduces (borrowing from C++) the notion of a function prototype declaration that incorporates the types of the parameters, and includes an explicit recognition of variadic functions together with an approved way of dealing with them. See Pars. A.7.3.2, A.8.6.3, B.7. The older style is still accepted, with restrictions.

- Empty declarations, which have no declarators and don't declare at least a structure, union, or enumeration, are forbidden by the Standard. On the other hand, a declaration with just a structure or union tag redeclares that tag even if it was declared in an outer scope.

- External data declarations without any specifiers or qualifiers (just a naked declarator) are forbidden.

- Some implementations, when presented with an `extern` declaration in an inner block, would export the declaration to the rest of the file. The Standard makes it clear that the scope of such a declaration is just the block.

- The scope of parameters is injected into a function's compound statement, so that variable declarations at the top level of the function cannot hide the parameters.

- The name spaces of identifiers are somewhat different. The Standard puts all tags in a single name space, and also introduces a separate name space for labels; see Par.A.11.1. Also, member names are associated with the structure or union of which they are a part. (This has been common practice from some time.)

- Unions may be initialized; the initializer refers to the first member.

- Automatic structures, unions, and arrays may be initialized, albeit in a restricted way.

- Character arrays with an explicit size may be initialized by a string literal with exactly that many characters (the `\0` is quietly squeezed out).

- The controlling expression, and the case labels, of a switch may have any integral type.