```
static Header base;        /* empty list to get started */
static Header *freep = NULL;      /* start of free list */

/* malloc:  general-purpose storage allocator */
void *malloc(unsigned nbytes)
{
    Header *p, *prevp;
    Header *morecore(unsigned);
    unsigned nunits;

    nunits = (nbytes+sizeof(Header)-1)/sizeof(Header) + 1;
    if ((prevp = freep) == NULL) {  /* no free list yet */
        base.s.ptr = freep = prevp = &base;
        base.s.size = 0;
    }
    for (p = prevp->s.ptr; ; prevp = p, p = p->s.ptr) {
        if (p->s.size >= nunits) {     /* big enough */
            if (p->s.size == nunits)      /* exactly */
                prevp->s.ptr = p->s.ptr;
            else {              /* allocate tail end */
                p->s.size -= nunits;
                p += p->s.size;
                p->s.size = nunits;
            }
            freep = prevp;
            return (void *)(p+1);
        }
        if (p == freep)  /* wrapped around free list */
            if ((p = morecore(nunits)) == NULL)
                return NULL;    /* none left */
    }
}
```

The function `morecore` obtains storage from the operating system. The details of how it does this vary from system to system. Since asking the system for memory is a comparatively expensive operation, we don't want to do that on every call to `malloc`, so `morecore` requests at least `NALLOC` units; this larger block will be chopped up as needed. After setting the size field, `morecore` inserts the additional memory into the arena by calling `free`.

The UNIX system call `sbrk(n)` returns a pointer to `n` more bytes of storage. `sbrk` returns `-1` if there was no space, even though `NULL` would have been a better design. The `-1` must be cast to `char *` so it can be compared with the return value. Again, casts make the function relatively immune to the details of pointer representation on different machines. There is still one assumption, however, that pointers to different blocks returned by `sbrk` can be meaningfully compared. This is not guaranteed by the standard, which permits pointer comparisons only within an array. Thus this version of `malloc` is portable only among machines for which general pointer comparison is meaningful.

```
#define NALLOC  1024    /* minimum #units to request */

/* morecore:  ask system for more memory */
static Header *morecore(unsigned nu)
{
    char *cp, *sbrk(int);
    Header *up;

    if (nu < NALLOC)
        nu = NALLOC;
    cp = sbrk(nu * sizeof(Header));
    if (cp == (char *) -1)  /* no space at all */
        return NULL;
    up = (Header *) cp;
    up->s.size = nu;
    free((void *)(up+1));
    return freep;
}
```

`free` itself is the last thing. It scans the free list, starting at `freep`, looking for the place to insert the free block. This is either between two existing blocks or at one end of the list. In any case, if the block being freed is adjacent to either neighbor, the adjacent blocks are combined. The only troubles are keeping the pointers pointing to the right things and the sizes correct.

```
/* free:  put block ap in free list */
void free(void *ap)
{
    Header *bp, *p;

    bp = (Header *)ap - 1;    /* point to block header */
    for (p = freep; !(bp > p && bp < p->s.ptr); p = p->s.ptr)
        if (p >= p->s.ptr && (bp > p || bp < p->s.ptr))
            break;  /* freed block at start or end of arena */

    if (bp + bp->s.size == p->s.ptr) { /* join to upper nbr */
        bp->s.size += p->s.ptr->s.size;
        bp->s.ptr = p->s.ptr->s.ptr;
    } else
        bp->s.ptr = p->s.ptr;
    if (p + p->s.size == bp) {          /* join to lower nbr */
        p->s.size += bp->s.size;
        p->s.ptr = bp->s.ptr;
    } else
        p->s.ptr = bp;
    freep = p;
}
```

Although storage allocation is intrinsically machine-dependent, the code above illustrates how the machine dependencies can be controlled and confined to a very small part of the program. The use of `typedef` and `union` handles

alignment (given that `sbrk` supplies an appropriate pointer). Casts arrange that pointer conversions are made explicit, and even cope with a badly-designed system interface. Even though the details here are related to storage allocation, the general approach is applicable to other situations as well.

Exercise 8-6. The standard library function `calloc(n,size)` returns a pointer to `n` objects of size `size`, with the storage initialized to zero. Write `calloc`, by calling `malloc` or by modifying it. □

Exercise 8-7. `malloc` accepts a size request without checking its plausibility; `free` believes that the block it is asked to free contains a valid size field. Improve these routines so they take more pains with error checking. □

Exercise 8-8 Write a routine `bfree(p,n)` that will free an arbitrary block `p` of `n` characters into the free list maintained by `malloc` and `free`. By using `bfree`, a user can add a static or external array to the free list at any time. □

APPENDIX A: Reference Manual

A1. Introduction

This manual describes the C language specified by the draft submitted to ANSI on 31 October, 1988, for approval as "American National 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.

A2. Lexical Conventions

A program consists of one or more *translation units* stored in files. It is translated in several phases, which are described in §A12. The first phases do low-level lexical transformations, carry out directives introduced by lines beginning with the # character, and perform macro definition and expansion. When the preprocessing of §A12 is complete, the program has been reduced to a sequence of tokens.

A2.1 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.

A2.2 Comments

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

A2.3 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 make more characters significant. Internal identifiers include preprocessor macro names and all other names that do not have external linkage (§A11.2). Identifiers with external linkage are more restricted: implementations may make as few as the first six characters as significant, and may ignore case distinctions.

A2.4 Keywords

The following identifiers are reserved for 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.

A2.5 Constants

There are several kinds of constants. Each has a data type; §A4.2 discusses the basic types.

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

A2.5.1 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 §A4 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`.

> 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.

A2.5.2 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*	`\x`*hh*
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 `\x`*hh* 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, octal or 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.

A2.5.3 Floating Constants

A floating constant consists of an integer part, a decimal point, a fraction part, an `e` or `E`, an optionally signed integer exponent and an optional type suffix, 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`.

> Suffixes on floating constants are new.

A2.5.4 Enumeration Constants

Identifiers declared as enumerators (see §A8.4) are constants of type `int`.

A2.6 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 §A4 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.

A3. 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 §A13.

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

A4. 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 §A11.

A4.1 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 (§A9.3), 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*.

A4.2 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 (§A8.4). Enumerations behave like integers, but it is common for a compiler to issue a warning when an object of a particular enumeration type 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.

A4.3 Derived Types

Besides 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 one of several objects of various types.

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

A4.4 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 §A8.2.

A5. 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: for example, if `E` is an expression of pointer type, then `*E` is an lvalue expression referring to the object to which `E` points. The name "lvalue" comes from the assignment expression `E1 = E2` in which the left operand `E1` must be an lvalue expression. The discussion of each operator specifies whether it expects lvalue operands and whether it yields an lvalue.

A6. 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. §A6.5 summarizes the conversions demanded by most ordinary operators; it will be supplemented as required by the discussion of each operator.

A6.1 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*.

A6.2 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.

A6.3 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.

A6.4 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.

A6.5 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.

A6.6 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 (§A7.7).

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 (§A7.7).

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-dependent aspects. They must be specified by an explicit type-conversion operator, or

cast (§§A7.5 and A8.8).

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.

An object of integral type may be explicitly converted to a pointer. The mapping always carries a sufficiently wide integer converted from a pointer back to the same pointer, but is otherwise 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 §A6.8, 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 (§§A4.4, A8.2) 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.

A6.7 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 (§A9.2) or as the left operand of a comma operator (§A7.18).

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.

A6.8 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 §A6.6, 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.

A7. 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 + (§A7.7) are those expressions defined in §§A7.1-A7.6. 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 in §A13 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 an 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.

A7.1 Pointer Generation

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 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*."

A7.2 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 (§A5) 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 §A2.5.

A string literal is a primary expression. Its type is originally "array of `char`" (for wide-character strings, "array of `wchar_t`"), but following the rule given in §A7.1, 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 §A8.7.

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

A7.3 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
argument-expression-list `,` *assignment-expression*

A7.3.1 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 §A8.6.2 for further discussion.

A7.3.2 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 (§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 implicit declaration and pointer generation, §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 is also called a function prototype. In the old style, parameter types are not specified. Function declaration is discussed in §§A8.6.3 and A10.1.

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 (§A6.1) 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 argument 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 prototype visible at the call must agree with the corresponding 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.

A7.3.3 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 a 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 §A8.3.

> 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.

A7.3.4 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 (§A7.7) and assignment (§A7.17) for further constraints on the operand and details of the operation. The result is not an lvalue.

A7.4 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
& * + - ~ !

A7.4.1 Prefix Incrementation Operators

A unary expression preceded 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 (§A7.7) and assignment (§A7.17) for further constraints on the operand and details of the operation. The result is not an lvalue.

A7.4.2 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*."

A7.4.3 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*.

A7.4.4 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 unary -.

A7.4.5 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.

A7.4.6 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.

A7.4.7 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`.

A7.4.8 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`.

A7.5 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*. Type names are described in §A8.8. The effects of conversions are described in §A6. An expression with a cast is not an lvalue.

A7.6 Multiplicative Operators

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

multiplicative-expression:
cast-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.

A7.7 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 depends on the implementation, but 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.

A7.8 Shift Operators

The shift operators `<<` and `>>` group left-to-right. For both operators, each operand must be integral, and is subject to the integral 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 if it has a non-negative value; otherwise the result is implementation-defined.

A7.9 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 `a<b` 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 are to members of the same union, they compare equal; if the pointers refer to members of an array, the comparison is equivalent to comparison of 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.

A7.10 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 §A6.6.

A7.11 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.

A7.12 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.

A7.13 Bitwise Inclusive OR Operator

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

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

A7.14 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`.

A7.15 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 compares 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`.

A7.16 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 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 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 (§A8.2) in the type to which the pointer points are insignificant, but the result type inherits qualifiers from both arms of the conditional.

A7.17 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.

A7.18 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 evaluation of the right operand. In contexts where comma is given a special meaning, for example in lists of function arguments (§A7.3.2) and lists of initializers (§A8.7), 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.

A7.19 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 an 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 or 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 §A12.5.

A8. 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 (§A8.5); 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.

A8.1 Storage Class Specifiers

The storage class specifiers are:

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

The meanings of the storage classes were discussed in §A4.

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 §A11.2.

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 §A11.2.

The `typedef` specifier does not reserve storage and is called a storage class specifier only for syntactic convenience; it is discussed in §A8.9.

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 §§A10-A11.

A8.2 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-independent 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.