The initialization of the `_flag` part of the structure shows that `stdin` is to be read, `stdout` is to be written, and `stderr` is to be written unbuffered.

Exercise 8-3. Rewrite `fopen` and `_fillbuf` with fields instead of explicit bit operations. □

Exercise 8-4. Design and write the routines `_flushbuf` and `fclose`. □

Exercise 8-5. The standard library provides a function

```
fseek(fp, offset, origin)
```

which is identical to `lseek` except that `fp` is a file pointer instead of a file descriptor. Write `fseek`. Make sure that your `fseek` coordinates properly with the buffering done for the other functions of the library. □

8.6 Example — Listing Directories

A different kind of file system interaction is sometimes called for — determining information *about* a file, not what it contains. The UNIX command *ls* ("list directory") is an example — it prints the names of files in a directory, and optionally, other information, such as sizes, permissions, and so on.

Since on UNIX at least a directory is just a file, there is nothing special about a command like *ls*; it reads a file and picks out the relevant parts of the information it finds there. Nonetheless, the format of that information is determined by the system, not by a user program, so *ls* needs to know how the system represents things.

We will illustrate some of this by writing a program called *fsize*. *fsize* is a special form of *ls* which prints the sizes of all files named in its argument list. If one of the files is a directory, *fsize* applies itself recursively to that directory. If there are no arguments at all, it processes the current directory.

To begin, a short review of file system structure. A directory is a file that contains a list of file names and some indication of where they are located. The "location" is actually an index into another table called the "inode table." The inode for a file is where all information about a file except its name is kept. A directory entry consists of only two items, an inode number and the file name. The precise specification comes by including the file `sys/dir.h`, which contains

```
#define   DIRSIZ    14   /* max length of file name */

struct direct  /* structure of directory entry */
{
     ino_t d_ino;    /* inode number */
     char d_name[DIRSIZ];      /* file name */
};
```

The "type" `ino_t` is a `typedef` describing the index into the inode table. It happens to be `unsigned` on PDP-11 UNIX, but this is not the sort of information to embed in a program: it might be different on a different system. Hence the `typedef`. A complete set of "system" types is found in `sys/types.h`.

The function `stat` takes a file name and returns all of the information in the inode for that file (or `-1` if there is an error). That is,

```
struct stat stbuf;
char *name;

stat(name, &stbuf);
```

fills the structure `stbuf` with the inode information for the file `name`. The structure describing the value returned by `stat` is in `sys/stat.h`, and looks like this:

```
struct stat    /* structure returned by stat */
{
     dev_t     st_dev;   /* device of inode */
     ino_t     st_ino;   /* inode number */
     short     st_mode;  /* mode bits */
     short     st_nlink; /* number of links to file */
     short     st_uid;   /* owner's userid */
     short     st_gid;   /* owner's group id */
     dev_t     st_rdev;  /* for special files */
     off_t     st_size;  /* file size in characters */
     time_t    st_atime; /* time last accessed */
     time_t    st_mtime; /* time last modified */
     time_t    st_ctime; /* time originally created */
};
```

Most of these are explained by the comment fields. The `st_mode` entry contains a set of flags describing the file; for convenience, the flag definitions are also part of the file `sys/stat.h`.

```
#define S_IFMT 0160000          /* type of file */
#define   S_IFDIR   0040000     /* directory */
#define   S_IFCHR   0020000     /* character special */
#define   S_IFBLK   0060000     /* block special */
#define   S_IFREG   0100000     /* regular */
#define S_ISUID     04000   /* set user id on execution */
#define S_ISGID     02000   /* set group id on execution */
#define S_ISVTX     01000   /* save swapped text after use */
#define S_IREAD     0400    /* read permission */
#define S_IWRITE    0200    /* write permission */
#define S_IEXEC     0100    /* execute permission */
```

Now we are able to write the program *fsize*. If the mode obtained from `stat` indicates that a file is not a directory, then the size is at hand and can be printed directly. If it is a directory, however, then we have to process that directory one file at a time; it in turn may contain sub-directories, so the process is recursive.

The main routine as usual deals primarily with command-line arguments; it hands each argument to the function `fsize` in a big buffer.

```
#include <stdio.h>
#include <sys/types.h>    /* typedefs */
#include <sys/dir.h>      /* directory entry structure */
#include <sys/stat.h>     /* structure returned by stat */
#define   BUFSIZE   256

main(argc, argv)     /* fsize: print file sizes */
char *argv[];
{
     char buf[BUFSIZE];

     if (argc == 1) {     /* default: current directory */
          strcpy(buf, ".");
          fsize(buf);
     } else
          while (--argc > 0) {
               strcpy(buf, *++argv);
               fsize(buf);
          }
}
```

The function `fsize` prints the size of the file. If the file is a directory, however, `fsize` first calls `directory` to handle all the files in it. Note the use of the flag names `S_IFMT` and `S_IFDIR` from `stat.h`.

```
fsize(name)     /* print size for name */
char *name;
{
     struct stat stbuf;

     if (stat(name, &stbuf) == -1) {
          fprintf(stderr, "fsize: can't find %s\n", name);
          return;
     }
     if ((stbuf.st_mode & S_IFMT) == S_IFDIR)
          directory(name);
     printf("%8ld %s\n", stbuf.st_size, name);
}
```

The function `directory` is the most complicated. Much of it is concerned, however, with creating the full pathname of the file being dealt with.

```
directory(name)      /* fsize for all files in name */
char *name;
{
     struct direct dirbuf;
     char *nbp, *nep;
     int i, fd;

     nbp = name + strlen(name);
     *nbp++ = '/';  /* add slash to directory name */
     if (nbp+DIRSIZ+2 >= name+BUFSIZE)  /* name too long */
          return;
     if ((fd = open(name, 0)) == -1)
          return;
     while (read(fd, (char *)&dirbuf, sizeof(dirbuf))>0) {
          if (dirbuf.d_ino == 0)   /* slot not in use */
               continue;
          if (strcmp(dirbuf.d_name, ".") == 0
            || strcmp(dirbuf.d_name, "..") == 0)
               continue; /* skip self and parent */
          for (i=0, nep=nbp; i < DIRSIZ; i++)
               *nep++ = dirbuf.d_name[i];
          *nep++ = '\0';
          fsize(name);
     }
     close(fd);
     *--nbp = '\0'; /* restore name */
}
```

If a directory slot is not currently in use (because a file has been removed), the inode entry is zero, and this position is skipped. Each directory also contains entries for itself, called `"."`, and its parent, `".."`; clearly

these must also be skipped, or the program will run for quite a while.

Although the *fsize* program is rather specialized, it does indicate a couple of important ideas. First, many programs are not "system programs"; they merely use information whose form or content is maintained by the operating system. Second, for such programs, it is crucial that the representation of the information appear only in standard "header files" like `stat.h` and `dir.h`, and that programs include those files instead of embedding the actual declarations in themselves.

8.7 Example — A Storage Allocator

In Chapter 5, we presented a simple-minded version of `alloc`. The version which we will now write is unrestricted: calls to `alloc` and `free` may be intermixed in any order; `alloc` calls upon the operating system to obtain more memory as necessary. Besides being useful in their own right, these routines illustrate some of the considerations involved in writing machine-dependent code in a relatively machine-independent way, and also show a real-life application of structures, unions and `typedef`.

Rather than allocating from a compiled-in fixed-sized array, `alloc` will request space from the operating system as needed. Since other activities in the program may also request space asynchronously, the space `alloc` manages may not be contiguous. Thus its free storage is kept as a chain of free blocks. Each block contains a size, a pointer to the next block, and the space itself. The blocks are kept in order of increasing storage address, and the last block (highest address) points to the first, so the chain is actually a ring.

When a request is made, the free list is scanned until a big enough block is found. If the block is exactly the size requested it is unlinked from the list and returned to the user. If the block is too big, it is split, and the proper amount is returned to the user while the residue is put back on the free list. If no big enough block is found, another block is obtained from the operating system and linked into the free list; searching then resumes.

Freeing also causes a search of the free list, to find the proper place to insert the block being freed. If the block being freed is adjacent to a free list block on either side, it is coalesced with it into a single bigger block, so storage does not become too fragmented. Determining adjacency is easy because the free list is maintained in storage order.

One problem, which we alluded to in Chapter 5, is to ensure that the storage returned by `alloc` is aligned properly for the objects that will be stored in it. Although machines vary, for each machine there is a most restrictive type: if the most restricted type can be stored at a particular address, all other types may be also. For example, on the IBM 360/370, the Honeywell 6000, and many other machines, any object may be stored on a boundary appropriate for a `double`; on the PDP-11, `int` suffices.

A free block contains a pointer to the next block in the chain, a record of the size of the block, and then the free space itself; the control information at the beginning is called the "header." To simplify alignment, all blocks are multiples of the header size, and the header is aligned properly. This is achieved by a union that contains the desired header structure and an instance of the most restrictive alignment type:

```
typedef int ALIGN;  /* forces alignment on PDP-11 */

union header { /* free block header */
     struct {
          union header *ptr;  /* next free block */
          unsigned size; /* size of this free block */
     } s;
     ALIGN     x;   /* force alignment of blocks */
};

typedef union header HEADER;
```

In `alloc`, the requested size in characters is rounded up to the proper number of header-sized units; the actual block that will be allocated contains one more unit, for the header itself, and this is the value recorded in the `size` field of the header. The pointer returned by `alloc` points at the free space, not at the header itself.

```
static HEADER base; /* empty list to get started */
static HEADER *allocp = NULL; /* last allocated block */

char *alloc(nbytes) /* general-purpose storage allocator */
unsigned nbytes;
{
     HEADER *morecore();
     register HEADER *p, *q;
     register int nunits;

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

The variable `base` is used to get started; if `allocp` is `NULL`, as it is at the first call of `alloc`, then a degenerate free list is created: it contains one block of size zero, and points to itself. In any case, the free list is then searched. The search for a free block of adequate size begins at the point (`allocp`) where the last block was found; this strategy helps keep the list homogeneous. If a too-big block is found, the tail end is returned to the user; in this way the header of the original needs only to have its size adjusted. In all cases, the pointer returned to the user is to the actual free area, which is one unit beyond the header. Notice that `p` is converted to a character pointer before being returned by `alloc`.

The function `morecore` obtains storage from the operating system. The details of how this is done of course vary from system to system. In UNIX, the system entry `sbrk(n)` returns a pointer to `n` more bytes of storage. (The pointer satisfies all alignment restrictions.) Since asking the

system for memory is a comparatively expensive operation, we don't want to do that on every call to `alloc`, so `morecore` rounds up the number of units requested of it to a larger value; this larger block will be chopped up as needed. The amount of scaling is a parameter that can be tuned as needed.

```
#define   NALLOC    128  /* #units to allocate at once */

static HEADER *morecore(nu)   /* ask system for memory */
unsigned nu;
{
     char *sbrk();
     register char *cp;
     register HEADER *up;
     register int rnu;

     rnu = NALLOC * ((nu+NALLOC-1) / NALLOC);
     cp = sbrk(rnu * sizeof(HEADER));
     if ((int)cp == -1)  /* no space at all */
          return(NULL);
     up = (HEADER *)cp;
     up->s.size = rnu;
     free((char *)(up+1));
     return(allocp);
}
```

`sbrk` returns −1 if there was no space, even though `NULL` would have been a better choice. The −1 must be converted to an `int` so it can be safely compared. Again, casts are heavily used so the function is relatively immune to the details of pointer representation on different machines.

`free` itself is the last thing. It simply scans the free list, starting at `allocp`, 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(ap)  /* put block ap in free list */
char *ap;
{
     register HEADER *p, *q;

     p = (HEADER *)ap - 1;     /* point to header */
     for (q=allocp; !(p > q && p < q->s.ptr); q=q->s.ptr)
          if (q >= q->s.ptr && (p > q || p < q->s.ptr))
               break;     /* at one end or other */

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

Although storage allocation is intrinsically machine dependent, the code shown 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`, using `alloc` either as a model or as a function to be called. □

Exercise 8-7. `alloc` 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 to take more pains with error checking. □

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

APPENDIX A: C REFERENCE MANUAL

1. Introduction

This manual describes the C language on the DEC PDP-11, the Honeywell 6000, the IBM System/370, and the Interdata 8/32. Where differences exist, it concentrates on the PDP-11, but tries to point out implementation-dependent details. With few exceptions, these dependencies follow directly from the underlying properties of the hardware; the various compilers are generally quite compatible.

2. Lexical conventions

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

If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters which could possibly constitute a token.

2.1 Comments

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

2.2 Identifiers (Names)

An identifier is 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. No more than the first eight characters are significant, although more may be used. External identifiers, which are used by various assemblers and loaders, are more restricted:

DEC PDP-11	7 characters, 2 cases
Honeywell 6000	6 characters, 1 case
IBM 360/370	7 characters, 1 case
Interdata 8/32	8 characters, 2 cases

2.3 Keywords

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

```
int         extern      else
char        register    for
float       typedef     do
double      static      while
struct      goto        switch
union       return      case
long        sizeof      default
short       break       entry
unsigned    continue
auto        if
```

The `entry` keyword is not currently implemented by any compiler but is reserved for future use. Some implementations also reserve the words `fortran` and `asm`.

2.4 Constants

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

2.4.1 Integer constants

An integer constant consisting of a sequence of digits is taken to be octal if it begins with `0` (digit zero), decimal otherwise. The digits `8` and `9` have octal value 10 and 11 respectively. A sequence of digits preceded by `0x` or `0X` (digit zero) is taken to be a hexadecimal integer. The hexadecimal digits include `a` or `A` through `f` or `F` with values 10 through 15. A decimal constant whose value exceeds the largest signed machine integer is taken to be `long`; an octal or hex constant which exceeds the largest unsigned machine integer is likewise taken to be `long`.

2.4.2 Explicit long constants

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

2.4.3 Character constants

A character constant is a character enclosed in single quotes, as in `'x'`. The value of a character constant is the numerical value of the character in the machine's character set.

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

newline	NL (LF)	`\n`
horizontal tab	HT	`\t`
backspace	BS	`\b`
carriage return	CR	`\r`
form feed	FF	`\f`
backslash	\	`\\`
single quote	'	`\'`
bit pattern	*ddd*	`\`*ddd*

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

2.4.4 Floating constants

A floating constant consists of an integer part, a decimal point, a fraction part, an `e` or `E`, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing; either the decimal point or the `e` and the exponent (not both) may be missing. Every floating constant is taken to be double-precision.

2.5 Strings

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

2.6 Hardware characteristics

The following table summarizes certain hardware properties which vary from machine to machine. Although these affect program portability, in practice they are less of a problem than might be thought *a priori*.

	DEC PDP-11	Honeywell 6000	IBM 370	Interdata 8/32
	ASCII	ASCII	EBCDIC	ASCII
`char`	8 bits	9 bits	8 bits	8 bits
`int`	16	36	32	32
`short`	16	36	16	16
`long`	32	36	32	32
`float`	32	36	32	32
`double`	64	72	64	64
range	$\pm 10^{\pm 38}$	$\pm 10^{\pm 38}$	$\pm 10^{\pm 76}$	$\pm 10^{\pm 76}$

For these four machines, floating point numbers have 8 bit exponents.

3. Syntax notation

In the syntax notation used in this manual, syntactic categories are indicated by *italic* type, and literal words and characters in `bold` type. Alternative categories are listed on separate lines. An optional terminal or non-terminal symbol is indicated by the subscript "opt," so that

{ *expression*$_{opt}$ }

indicates an optional expression enclosed in braces. The syntax is summarized in §18.

4. What's in a name?

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

There are four declarable storage classes: automatic, static, external, and register. Automatic variables are local to each invocation of a block (§9.2), and are discarded upon exit from the block; static variables are local to a block, but retain their values upon reentry to a block even after control has left the block; external variables exist and retain their values throughout the execution of the entire program, and may be used for communication between functions, even separately compiled functions. Register variables are (if possible) stored in the fast registers of the machine; like automatic variables they are local to each block and disappear on exit from the block.

C supports several fundamental types of objects:

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

Up to three sizes of integer, declared `short int`, `int`, and `long int`, are available. Longer integers provide no less storage than shorter ones, but the implementation may make either short integers, or long integers, or both, equivalent to

plain integers. "Plain" integers have the natural size suggested by the host machine architecture; the other sizes are provided to meet special needs.

Unsigned integers, declared `unsigned`, obey the laws of arithmetic modulo 2^n where n is the number of bits in the representation. (On the PDP-11, unsigned long quantities are not supported.)

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

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

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

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

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

5. Objects and lvalues

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

6. Conversions

A number of operators may, depending on their operands, cause conversion of the value of an operand from one type to another. This section explains the result to be expected from such conversions. §6.6 summarizes the conversions demanded by most ordinary operators; it will be supplemented as required by the discussion of each operator.

6.1 Characters and integers

A character or a short integer may be used wherever an integer may be used. In all cases the value is converted to an integer. Conversion of a shorter integer to a longer always involves sign extension; integers are signed quantities. Whether or not sign-extension occurs for characters is machine dependent, but it is guaranteed that a member of the standard character set is non-negative. Of the machines treated by this manual, only the PDP-11 sign-extends. On the PDP-11, character variables range in value from −128 to 127; the characters of the ASCII alphabet are all positive. A character constant specified with an octal escape suffers sign extension and may appear negative; for example, `'\377'` has the value −1.

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

6.2 Float and double

All floating arithmetic in C is carried out in double-precision; whenever a `float` appears in an expression it is lengthened to `double` by zero-padding its fraction. When a `double` must be converted to `float`, for example by an assignment, the `double` is rounded before truncation to `float` length.

6.3 Floating and integral

Conversions of floating values to integral type tend to be rather machine-dependent; in particular the direction of truncation of negative numbers varies from machine to machine. The result is undefined if the value will not fit in the space provided.

Conversions of integral values to floating type are well behaved. Some loss of precision occurs if the destination lacks sufficient bits.

6.4 Pointers and integers

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

Two pointers to objects of the same type may be subtracted; in this case the result is converted to an integer as specified in the discussion of the subtraction operator.

6.5 Unsigned

Whenever an unsigned integer and a plain integer are combined, the plain integer is converted to unsigned and the result is unsigned. The value is the least unsigned integer congruent to the signed integer (modulo 2^{wordsize}). In a 2's complement representation, this conversion is conceptual and there is no actual change in the bit pattern.

When an unsigned integer is converted to `long`, the value of the result is the same numerically as that of the unsigned integer. Thus the conversion amounts to padding with zeros on the left.

6.6 Arithmetic conversions

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

First, any operands of type `char` or `short` are converted to `int`, and any of type `float` are converted to `double`.
Then, if either operand is `double`, the other is converted to `double` and that is the type of the result.
Otherwise, if either operand is `long`, the other is converted to `long` and that is the type of the result.
Otherwise, if either operand is `unsigned`, the other is converted to `unsigned` and that is the type of the result.
Otherwise, both operands must be `int`, and that is the type of the result.

7. Expressions

The precedence of expression operators is the same as the order of the major subsections of this section, highest precedence first. Thus, for example, the expressions referred to as the operands of + (§7.4) are those expressions defined in §§7.1-7.3. Within each subsection, the operators have the same precedence. Left- or right-associativity is specified in each subsection for the operators discussed therein. The precedence and associativity of all the expression operators is summarized in the grammar of §18.

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

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

7.1 Primary expressions

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

primary-expression:
- *identifier*
- *constant*
- *string*
- (*expression*)
- *primary-expression* [*expression*]
- *primary-expression* ($expression\text{-}list_{opt}$)
- *primary-lvalue* . *identifier*
- *primary-expression* -> *identifier*

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

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

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

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

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

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

A function call is a primary expression followed by parentheses containing a possibly empty, comma-separated list of expressions which constitute the actual arguments to the function. The primary expression must be of type "function returning ...", and the result of the function call is of type "...". As indicated below, a hitherto unseen identifier followed immediately by a left parenthesis is contextually declared to represent a function returning an integer; thus in the most common case, integer-valued functions need not be declared.

Any actual arguments of type `float` are converted to `double` before the call; any of type `char` or `short` are converted to `int`; and as usual, array names are converted to pointers. No other conversions are performed automatically; in particular, the compiler does not compare the types of actual arguments with those of formal arguments. If conversion is needed, use a cast; see §7.2, 8.7.

In preparing for the call to a function, a copy is made of each actual parameter; thus, all argument-passing in C is strictly by value. A function may change the values of its formal parameters, but these changes cannot affect the values of the actual parameters. On the other hand, it is possible to pass a pointer on the understanding that the function may change the value of the object to which the pointer points. An array name is a pointer expression. The order of evaluation of arguments is undefined by the language; take note that the various compilers differ.

Recursive calls to any function are permitted.

A primary expression followed by a dot followed by an identifier is an expression. The first expression must be an lvalue naming a structure or a union, and the identifier must name a member of the structure or union. The result is an lvalue referring to the named member of the structure or union.

A primary expression followed by an arrow (built from a – and a >) followed by an identifier is an expression. The first expression must be a pointer to a structure or a union and the identifier must name a member of that structure or union. The result is an lvalue referring to the named member of the structure or union to which the pointer expression points.

Thus the expression `E1->MOS` is the same as `(*E1).MOS`. Structures and unions are discussed in §8.5. The rules given here for the use of structures and unions are not enforced strictly, in order to allow an escape from the typing mechanism. See §14.1.

7.2 Unary operators

Expressions with unary operators group right-to-left.

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

The unary * operator means *indirection*: the expression must be a pointer, and the result is an lvalue referring to the object to which the expression points. If the type of the expression is "pointer to ...", the type of the result is "...".

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

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

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

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

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

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

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

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

An expression preceded by the parenthesized name of a data type causes conversion of the value of the expression to the named type. This construction is called a *cast*. Type names are described in §8.7.

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

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

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

7.3 Multiplicative operators

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

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

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

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

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

7.4 Additive operators

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

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

The result of the `+` operator is the sum of the operands. A pointer to an object in an array and a value of any integral type may be added. The latter is in all cases converted to an address offset by multiplying it by the length of the object to which the pointer points. The result is a pointer of the same type as the original pointer, and which points to another object in the same array, appropriately offset from the original object. Thus if `P` is a pointer to an object in an array, the expression `P+1` is a pointer to the next object in the array.

No further type combinations are allowed for pointers.

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

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

If two pointers to objects of the same type are subtracted, the result is converted (by division by the length of the object) to an `int` representing the number of objects separating the pointed-to objects. This conversion will in general give unexpected results unless the pointers point to objects in the same array, since pointers, even to objects of the same type, do not necessarily differ by a multiple of the object-length.

7.5 Shift operators

The shift operators << and >> group left-to-right. Both perform the usual arithmetic conversions on their operands, each of which must be integral. Then the right operand is converted to `int`; the type of the result is that of the left operand. The result is undefined if the right operand is negative, or greater than or equal to the length of the object in bits.

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

The value of `E1<<E2` is `E1` (interpreted as a bit pattern) left-shifted `E2` bits; vacated bits are 0-filled. The value of `E1>>E2` is `E1` right-shifted `E2` bit positions. The right shift is guaranteed to be logical (0-fill) if `E1` is `unsigned`; otherwise it may be (and is, on the PDP-11) arithmetic (fill by a copy of the sign bit).

7.6 Relational operators

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

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

The operators < (less than), > (greater than), <= (less than or equal to) and >= (greater than or equal to) all yield 0 if the specified relation is false and 1 if it is true. The type of the result is `int`. The usual arithmetic conversions are performed. Two pointers may be compared; the result depends on the relative locations in the address space of the pointed-to objects. Pointer comparison is portable only when the pointers point to objects in the same array.

7.7 Equality operators

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

The == (equal to) and the != (not equal to) operators are exactly analogous to the relational operators except for their lower precedence. (Thus `a<b == c<d` is 1 whenever `a<b` and `c<d` have the same truth-value).

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

7.8 Bitwise AND operator

and-expression:
expression & *expression*

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

7.9 Bitwise exclusive OR operator

exclusive-or-expression:
expression ^ *expression*

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

7.10 Bitwise inclusive OR operator

inclusive-or-expression:
expression | *expression*

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

7.11 Logical AND operator

logical-and-expression:
expression && *expression*

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

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