- `f` The argument is taken to be a `float` or `double` and converted to decimal notation of the form `[-]mmm.nnnnn` where the length of the string of `n`'s is specified by the precision. The default precision is 6. Note that the precision does not determine the number of significant digits printed in `f` format.
- `g` Use `%e` or `%f`, whichever is shorter; non-significant zeros are not printed.

If the character after the `%` is not a conversion character, that character is printed; thus `%` may be printed by `%%`.

Most of the format conversions are obvious, and have been illustrated in earlier chapters. One exception is precision as it relates to strings. The following table shows the effect of a variety of specifications in printing "hello, world" (12 characters). We have put colons around each field so you can see its extent.

```
:%10s:        :hello, world:
:%-10s:       :hello, world:
:%20s:        :        hello, world:
:%-20s:       :hello, world        :
:%20.10s:     :          hello, wor:
:%-20.10s:    :hello, wor          :
:%.10s:       :hello, wor:
```

a way of putting blank spaces.

A warning: `printf` uses its first argument to decide how many arguments follow and what their types are. It will get confused, and you will get nonsense answers, if there are not enough arguments or if they are the wrong type.

Exercise 7-1. Write a program which will print arbitrary input in a sensible way. As a minimum, it should print non-graphic characters in octal or hex (according to local custom), and fold long lines. □

7.4 Formatted Input — Scanf

The function `scanf` is the input analog of `printf`, providing many of the same conversion facilities in the opposite direction.

```
scanf(control, arg1, arg2, ...)
```

`scanf` reads characters from the standard input, interprets them according to the format specified in `control`, and stores the results in the remaining arguments. The control argument is described below; the other arguments, *each of which must be a pointer,* indicate where the corresponding converted input should be stored.

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

Blanks, tabs or newlines ("white space characters"), which are ignored.

Ordinary characters (not `%`) which are expected to match the next non-white space character of the input stream.

Conversion specifications, consisting of the character `%`, an optional assignment suppression character `*`, an optional number specifying a maximum field width, and a conversion character.

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

The conversion character indicates the interpretation of the input field; the corresponding argument must be a pointer, as required by the call by value semantics of C. The following conversion characters are legal:

`d` a decimal integer is expected in the input; the corresponding argument should be an integer pointer.

`o` an octal integer (with or without a leading zero) is expected in the input; the corresponding argument should be a integer pointer.

`x` a hexadecimal integer (with or without a leading `0x`) is expected in the input; the corresponding argument should be an integer pointer.

`h` a `short` integer is expected in the input; the corresponding argument should be a pointer to a `short` integer.

`c` a single character is expected; the corresponding argument should be a character pointer; the next input character is placed at the indicated spot. The normal skip over white space characters is suppressed in this case; to read the next non-white space character, use `%1s`.

- `s` a character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating `\0` which will be added.
- `f` a floating point number is expected; the corresponding argument should be a pointer to a `float`. The conversion character `e` is a synonym for `f`. The input format for `float`'s is an optional sign, a string of numbers possibly containing a decimal point, and an optional exponent field containing an `E` or `e` followed by a possibly signed integer.

The conversion characters `d`, `o` and `x` may be preceded by `l` (letter ell) to indicate that a pointer to `long` rather than `int` appears in the argument list. Similarly, the conversion characters `e` or `f` may be preceded by `l` to indicate that a pointer to `double` rather than `float` is in the argument list.

For example, the call

```
int  i;
float x;
char name[50];
scanf("%d %f %s", &i, &x, name);
```

with the input line

```
25   54.32E-1   Thompson
```

will assign the value 25 to `i`, the value 5.432 to `x`, and the string `"Thompson"`, properly terminated by `\0`, to `name`. The three input fields may be separated by as many blanks, tabs and newlines as desired. The call

```
int  i;
float x;
char name[50];
scanf("%2d %f %*d %2s", &i, &x, name);
```

with input

```
56789 0123 45a72
```

will assign 56 to `i`, assign 789.0 to `x`, skip over 0123, and place the string `"45"` in `name`. The next call to any input routine will begin searching at the letter `a`. In these two examples, `name` *is* a pointer and thus must *not* be preceded by a `&`.

As another example, the rudimentary calculator of Chapter 4 can now be written with `scanf` to do the input conversion:

```
#include  <stdio.h>

main()    /* rudimentary desk calculator */
{
      double sum, v;

      sum = 0;
      while (scanf("%lf", &v) != EOF)
            printf("\t%.2f\n", sum += v);
}
```

`scanf` stops when it exhausts its control string, or when some input fails to match the control specification. It returns as its value the number of successfully matched and assigned input items. This can be used to decide how many input items were found. On end of file, `EOF` is returned; note that this is different from 0, which means that the next input character does not match the first specification in the control string. The next call to `scanf` resumes searching immediately after the last character already returned.

A final warning: the arguments to `scanf` *must* be pointers. By far the most common error is writing

```
scanf("%d", n);
```

instead of

```
scanf("%d", &n);
```

7.5 In-memory Format Conversion

The functions `scanf` and `printf` have siblings called `sscanf` and `sprintf` which perform the corresponding conversions, but operate on a string instead of a file. The general format is

```
sprintf(string, control, arg1, arg2, ...)
sscanf(string, control, arg1, arg2, ...)
```

`sprintf` formats the arguments in `arg1`, `arg2`, etc., according to `control` as before, but places the result in `string` instead of on the standard output. Of course `string` had better be big enough to receive the result. As an example, if `name` is a character array and `n` is an integer, then

```
sprintf(name, "temp%d", n);
```

creates a string of the form `temp`*nnn* in `name`, where *nnn* is the value of `n`.

`sscanf` does the reverse conversions — it scans the `string` according to the format in `control`, and places the resulting values in `arg1`, `arg2`, etc. These arguments must be pointers. The call

```
sscanf(name, "temp%d", &n);
```

sets `n` to the value of the string of digits following `temp` in `name`.

Exercise 7-2. Rewrite the desk calculator of Chapter 4 using `scanf` and/or `sscanf` to do the input and number conversion. □

7.6 File Access

The programs written so far have all read the standard input and written the standard output, which we have assumed are magically pre-defined for a program by the local operating system.

The next step in I/O is to write a program that accesses a file which is *not* already connected to the program. One program that clearly illustrates the need for such operations is *cat*, which concatenates a set of named files onto the standard output. *cat* is used for printing files on the terminal, and as a general-purpose input collector for programs which do not have the capability of accessing files by name. For example, the command

```
cat x.c y.c
```

prints the contents of the files `x.c` and `y.c` on the standard output.

The question is how to arrange for the named files to be read — that is, how to connect the external names that a user thinks of to the statements which actually read the data.

The rules are simple. Before it can be read or written a file has to be *opened* by the standard library function `fopen`. `fopen` takes an external name (like `x.c` or `y.c`), does some housekeeping and negotiation with the operating system (details of which needn't concern us), and returns an internal name which must be used in subsequent reads or write of the file.

This internal name is actually a pointer, called a *file pointer*, to a structure which contains information about the file, such as the location of a buffer, the current character position in the buffer, whether the file is being read or written, and the like. Users don't need to know the details, because part of the standard I/O definitions obtained from `stdio.h` is a structure definition called `FILE`. The only declaration needed for a file pointer is exemplified by

```
FILE *fopen(), *fp;
```

This says that `fp` is a pointer to a `FILE`, and `fopen` returns a pointer to a `FILE`. Notice that `FILE` is a type name, like `int`, not a structure tag; it is implemented as a `typedef`. (Details of how this all works on the UNIX system are given in Chapter 8.)

The actual call to `fopen` in a program is

```
fp = fopen(name, mode);
```

The first argument of `fopen` is the *name* of the file, as a character string. The second argument is the *mode*, also as a character string, which indicates how one intends to use the file. Allowable modes are read (`"r"`), write (`"w"`), or append (`"a"`).

If you open a file which does not exist for writing or appending, it is created (if possible). Opening an existing file for writing causes the old contents to be discarded. Trying to read a file that does not exist is an error, and there may be other causes of error as well (like trying to read a file when you don't have permission). If there is any error, `fopen` will return the null pointer value `NULL` (which for convenience is also defined in `stdio.h`).

The next thing needed is a way to read or write the file once it is open. There are several possibilities, of which `getc` and `putc` are the simplest. `getc` returns the next character from a file; it needs the file pointer to tell it what file. Thus

```
c = getc(fp)
```

places in `c` the next character from the file referred to by `fp`, and `EOF` when it reaches end of file.

`putc` is the inverse of `getc`:

```
putc(c, fp)
```

puts the character `c` on the file `fp` and returns `c`. Like `getchar` and `putchar`, `getc` and `putc` may be macros instead of functions.

When a program is started, three files are opened automatically, and file pointers are provided for them. These files are the standard input, the standard output, and the standard error output; the corresponding file pointers are called `stdin`, `stdout`, and `stderr`. Normally these are all connected to the terminal, but `stdin` and `stdout` may be redirected to files or pipes as described in section 7.2.

`getchar` and `putchar` can be defined in terms of `getc`, `putc`, `stdin` and `stdout` as follows:

```
#define getchar()   getc(stdin)
#define putchar(c)  putc(c, stdout)
```

For formatted input or output of files, the functions `fscanf` and `fprintf` may be used. These are identical to `scanf` and `printf`, save that the first argument is a file pointer that specifies the file to be read or written; the control string is the second argument.

With these preliminaries out of the way, we are now in a position to write the program *cat* to concatenate files. The basic design is one that has been found convenient for many programs: if there are command-line arguments, they are processed in order. If there are no arguments, the standard input is processed. This way the program can be used stand-alone or as part of a larger process.

```
#include  <stdio.h>

main(argc, argv)    /* cat: concatenate files */
int argc;
char *argv[];
{
     FILE *fp, *fopen();

     if (argc == 1) /* no args; copy standard input */
          filecopy(stdin);
     else
          while (--argc > 0)
               if ((fp = fopen(*++argv, "r")) == NULL) {
                    printf("cat: can't open %s\n", *argv);
                    break;
               } else {
                    filecopy(fp);
                    fclose(fp);
               }
}

filecopy(fp)    /* copy file fp to standard output */
FILE *fp;
{
     int c;

     while ((c = getc(fp)) != EOF)
          putc(c, stdout);
}
```

The file pointers `stdin` and `stdout` are pre-defined in the I/O library as the standard input and standard output; they may be used anywhere an object of type `FILE *` can be. They are constants, however, *not* variables, so don't try to assign to them.

The function `fclose` is the inverse of `fopen`; it breaks the connection between the file pointer and the external name that was established by `fopen`, freeing the file pointer for another file. Since most operating systems have some limit on the number of simultaneously open files that a program may have, it's a good idea to free things when they are no longer needed, as we did in *cat*. There is also another reason for `fclose` on an output file — it flushes the buffer in which `putc` is collecting output. (`fclose` is called automatically for each open file when a program terminates normally.)

7.7 Error Handling — Stderr and Exit

The treatment of errors in *cat* is not ideal. The trouble is that if one of the files can't be accessed for some reason, the diagnostic is printed at the end of the concatenated output. That is acceptable if that output is going to a terminal, but bad if it's going into a file or into another program via a pipeline.

To handle this situation better, a second output file, called `stderr`, is assigned to a program in the same way that `stdin` and `stdout` are. If at all possible, output written on `stderr` appears on the user's terminal even if the standard output is redirected.

Let us revise *cat* to write its error messages on the standard error file.

```
#include  <stdio.h>

main(argc, argv)    /* cat: concatenate files */
int argc;
char *argv[];
{
    FILE *fp, *fopen();

    if (argc == 1) /* no args; copy standard input */
        filecopy(stdin);
    else
        while (--argc > 0)
            if ((fp = fopen(*++argv, "r")) == NULL) {
                fprintf(stderr,
                    "cat: can't open %s\n", *argv);
                exit(1);
            } else {
                filecopy(fp);
                fclose(fp);
            }
    exit(0);
}
```

The program signals errors two ways. The diagnostic output produced by `fprintf` goes onto `stderr`, so it finds its way to the user's terminal instead of disappearing down a pipeline or into an output file.

The program also uses the standard library function `exit`, which terminates program execution when it is called. The argument of `exit` is available to whatever process called this one, so the success or failure of the program can be tested by another program that uses this one as a subprocess. By convention, a return value of 0 signals that all is well, and various non-zero values signal abnormal situations.

`exit` calls `fclose` for each open output file, to flush out any buffered output, then calls a routine named `_exit`. The function `_exit` causes immediate termination without any buffer flushing; of course it may be called directly if desired.

7.8 Line Input and Output

The standard library provides a routine `fgets` which is quite similar to the `getline` function that we have used throughout the book. The call

```
fgets(line, MAXLINE, fp)
```

reads the next input line (including the newline) from file `fp` into the character array `line`; at most `MAXLINE-1` characters will be read. The resulting line is terminated with `\0`. Normally `fgets` returns `line`; on end of file it returns `NULL`. (Our `getline` returns the line length, and zero for end of file.)

For output, the function `fputs` writes a string (which need not contain a newline) to a file:

```
fputs(line, fp)
```

To show that there is nothing magic about functions like `fgets` and `fputs`, here they are, copied directly from the standard I/O library:

```
#include  <stdio.h>

char *fgets(s, n, iop)  /* get at most n chars from iop */
char *s;
int n;
register FILE *iop;
{
	register int c;
	register char *cs;

	cs = s;
	while (--n > 0 && (c = getc(iop)) != EOF)
		if ((*cs++ = c) == '\n')
			break;
	*cs = '\0';
	return((c == EOF && cs == s) ? NULL : s);
}
```

```
fputs(s, iop)  /* put string s on file iop */
register char *s;
register FILE *iop;
{
	register int c;

	while (c = *s++)
		putc(c, iop);
}
```

Exercise 7-3. Write a program to compare two files, printing the first line and character position where they differ. □

Exercise 7-4. Modify the pattern finding program of Chapter 5 to take its input from a set of named files or, if no files are named as arguments, from the standard input. Should the file name be printed when a matching line is found? □

Exercise 7-5. Write a program to print a set of files, starting each new one on a new page, with a title and a running page count for each file. □

7.9 Some Miscellaneous Functions

The standard library provides a variety of functions, a few of which stand out as especially useful. We have already mentioned the string functions `strlen`, `strcpy`, `strcat` and `strcmp`. Here are some others.

Character Class Testing and Conversion

Several macros perform character tests and conversions:

`isalpha(c)`	non-zero if `c` is alphabetic, 0 if not.
`isupper(c)`	non-zero if `c` is upper case, 0 if not.
`islower(c)`	non-zero if `c` is lower case, 0 if not.
`isdigit(c)`	non-zero if `c` is digit, 0 if not.
`isspace(c)`	non-zero if `c` is blank, tab or newline, 0 if not.
`toupper(c)`	convert `c` to upper case.
`tolower(c)`	convert `c` to lower case.

Ungetc

The standard library provides a rather restricted version of the function `ungetch` which we wrote in Chapter 4; it is called `ungetc`.

```
ungetc(c, fp)
```

pushes the character `c` back onto file `fp`. Only one character of pushback is allowed per file. `ungetc` may be used with any of the input functions and macros like scanf, getc, or getchar.

System Call

The function system(s) executes the command contained in the character string s, then resumes execution of the current program. The contents of s depend strongly on the local operating system. As a trivial example, on UNIX, the line

```
system("date");
```

causes the program date to be run; it prints the date and time of day.

Storage Management

The function calloc is rather like the alloc we have used in previous chapters.

```
calloc(n, sizeof(object))
```

returns a pointer to enough space for n objects of the specified size, or NULL if the request cannot be satisfied. The storage is initialized to zero.

The pointer has the proper alignment for the object in question, but it should be cast into the appropriate type, as in

```
char *calloc();
int  *ip;

ip = (int *) calloc(n, sizeof(int));
```

cfree(p) frees the space pointed to by p, where p is originally obtained by a call to calloc. There are no restrictions on the order in which space is freed, but it is a ghastly error to free something not obtained by calling calloc.

Chapter 8 shows the implementation of a storage allocator like calloc, in which allocated blocks may be freed in any order.

CHAPTER 8: THE UNIX SYSTEM INTERFACE

The material in this chapter is concerned with the interface between C programs and the UNIX† operating system. Since most C users are on UNIX systems, this should be helpful to a majority of readers. Even if you use C on a different machine, however, you should be able to glean more insight into C programming from studying these examples.

The chapter is divided into three major areas: input/output, file system, and a storage allocator. The first two parts assume a modest familiarity with the external characteristics of UNIX.

Chapter 7 was concerned with a system interface that is uniform across a variety of operating systems. On any particular system the routines of the standard library have to be written in terms of the I/O facilities actually available on the host system. In the next few sections we will describe the basic system entry points for I/O on the UNIX operating system, and illustrate how parts of the standard library can be implemented with them.

8.1 File Descriptors

In the UNIX operating system, all input and output is done by reading or writing files, because all peripheral devices, even the user's terminal, are files in the file system. This means that a single, homogeneous interface handles all communication between a program and peripheral devices.

In the most general case, before reading or writing a file, it is necessary to inform the system of your intent to do so, a process called "opening" the file. If you are going to write on a file it may also be necessary to create it. The system checks your right to do so (Does the file exist? Do you have permission to access it?), and if all is well, returns to the program a small positive integer called a *file descriptor*. Whenever I/O is to be done on the file, the file descriptor is used instead of the name to identify the file. (This is roughly analogous to the use of READ(5,...) and WRITE(6,...) in Fortran.) All information about an open file is maintained by the system; the user

† UNIX is a Trademark of Bell Laboratories.

program refers to the file only by the file descriptor.

Since input and output involving the user's terminal is so common, special arrangements exist to make this convenient. When the command interpreter (the "shell") runs a program, it opens three files, with file descriptors 0, 1, and 2, called the standard input, the standard output, and the standard error output. All of these are normally connected to the terminal, so if a program reads file descriptor 0 and writes file descriptors 1 and 2, it can do terminal I/O without worrying about opening the files.

The user of a program can *redirect* I/O to and from files with < and >:

```
prog <infile >outfile
```

In this case, the shell changes the default assignments for file descriptors 0 and 1 from the terminal to the named files. Normally file descriptor 2 remains attached to the terminal, so error messages can go there. Similar observations hold if the input or output is associated with a pipe. In all cases, it must be noted, the file assignments are changed by the shell, not by the program. The program does not know where its input comes from nor where its output goes, so long as it uses file 0 for input and 1 and 2 for output.

8.2 Low Level I/O — Read and Write

The lowest level of I/O in UNIX provides no buffering or any other services; it is in fact a direct entry into the operating system. All input and output is done by two functions called `read` and `write`. For both, the first argument is a file descriptor. The second argument is a buffer in your program where the data is to come from or go to. The third argument is the number of bytes to be transferred. The calls are

```
n_read = read(fd, buf, n);

n_written = write(fd, buf, n);
```

Each call returns a byte count which is the number of bytes actually transferred. On reading, the number of bytes returned may be less than the number asked for. A return value of zero bytes implies end of file, and -1 indicates an error of some sort. For writing, the returned value is the number of bytes actually written; it is generally an error if this isn't equal to the number supposed to be written.

The number of bytes to be read or written is quite arbitrary. The two most common values are 1, which means one character at a time ("unbuffered"), and 512, which corresponds to a physical blocksize on many peripheral devices. This latter size will be most efficient, but even character at a time I/O is not inordinately expensive.

Putting these facts together, we can write a simple program to copy its input to its output, the equivalent of the file copying program written for Chapter 1. In UNIX, this program will copy anything to anything, since the input and output can be redirected to any file or device.

```
#define   BUFSIZE   512  /* best size for PDP-11 UNIX */

main()    /* copy input to output */
{
     char buf[BUFSIZE];
     int  n;

     while ((n = read(0, buf, BUFSIZE)) > 0)
          write(1, buf, n);
}
```

If the file size is not a multiple of `BUFSIZE`, some `read` will return a smaller number of bytes to be written by `write`; the next call to `read` after that will return zero.

It is instructive to see how `read` and `write` can be used to construct higher level routines like `getchar`, `putchar`, etc. For example, here is a version of `getchar` which does unbuffered input.

```
#define   CMASK     0377 /* for making char's > 0 */

getchar() /* unbuffered single character input */
{
     char c;

     return((read(0, &c, 1) > 0) ? c & CMASK : EOF);
}
```

`c` *must* be declared `char`, because `read` accepts a character pointer. The character being returned must be masked with `0377` to ensure that it is positive; otherwise sign extension may make it negative. (The constant `0377` is appropriate for the PDP-11 but not necessarily for other machines.)

The second version of `getchar` does input in big chunks, and hands out the characters one at a time.

```
#define   CMASK     0377 /* for making char's > 0 */
#define   BUFSIZE   512

getchar() /* buffered version */
{
     static char    buf[BUFSIZE];
     static char    *bufp = buf;
     static int     n = 0;

     if (n == 0) {   /* buffer is empty */
          n = read(0, buf, BUFSIZE);
          bufp = buf;
     }
     return((--n >= 0) ? *bufp++ & CMASK : EOF);
}
```

8.3 Open, Creat, Close, Unlink

Other than the default standard input, output and error files, you must explicitly open files in order to read or write them. There are two system entry points for this, `open` and `creat` [sic].

`open` is rather like the `fopen` discussed in Chapter 7, except that instead of returning a file pointer, it returns a file descriptor, which is just an `int`.

```
int fd;

fd = open(name, rwmode);
```

As with `fopen`, the `name` argument is a character string corresponding to the external file name. The access mode argument is different, however: `rwmode` is 0 for read, 1 for write, and 2 for read and write access. `open` returns -1 if any error occurs; otherwise it returns a valid file descriptor.

It is an error to try to `open` a file that does not exist. The entry point `creat` is provided to create new files, or to re-write old ones.

```
fd = creat(name, pmode);
```

returns a file descriptor if it was able to create the file called `name`, and -1 if not. If the file already exists, `creat` will truncate it to zero length; it is not an error to `creat` a file that already exists.

If the file is brand new, `creat` creates it with the *protection mode* specified by the `pmode` argument. In the UNIX file system, there are nine bits of protection information associated with a file, controlling read, write and execute permission for the owner of the file, for the owner's group, and for all others. Thus a three-digit octal number is most convenient for specifying the permissions. For example, 0755 specifies read, write and execute permission for the owner, and read and execute permission for the group and everyone else.

To illustrate, here is a simplified version of the UNIX utility *cp*, a program which copies one file to another. (The main simplification is that our version copies only one file, and does not permit the second argument to be a directory.)

```
#define NULL 0
#define BUFSIZE 512
#define PMODE 0644 /* RW for owner, R for group, others */

main(argc, argv)    /* cp: copy f1 to f2 */
int argc;
char *argv[];
{
     int  f1, f2, n;
     char buf[BUFSIZE];

     if (argc != 3)
          error("Usage: cp from to", NULL);
     if ((f1 = open(argv[1], 0)) == -1)
          error("cp: can't open %s", argv[1]);
     if ((f2 = creat(argv[2], PMODE)) == -1)
          error("cp: can't create %s", argv[2]);

     while ((n = read(f1, buf, BUFSIZE)) > 0)
          if (write(f2, buf, n) != n)
               error("cp: write error", NULL);
     exit(0);
}

error(s1, s2)  /* print error message and die */
char *s1, *s2;
{
     printf(s1, s2);
     printf("\n");
     exit(1);
}
```

There is a limit (typically 15-25) on the number of files which a program may have open simultaneously. Accordingly, any program which intends to process many files must be prepared to re-use file descriptors. The routine `close` breaks the connection between a file descriptor and an open file, and frees the file descriptor for use with some other file. Termination of a program via `exit` or return from the main program closes all open files.

The function `unlink(filename)` removes the file `filename` from the file system.

Exercise 8-1. Rewrite the program `cat` from Chapter 7 using `read`, `write`, `open` and `close` instead of their standard library equivalents. Perform experiments to determine the relative speeds of the two versions. □

8.4 Random Access — Seek and Lseek

File I/O is normally sequential: each `read` or `write` takes place at a position in the file right after the previous one. When necessary, however, a file can be read or written in any arbitrary order. The system call `lseek` provides a way to move around in a file without actually reading or writing:

```
lseek(fd, offset, origin);
```

forces the current position in the file whose descriptor is `fd` to move to position `offset`, which is taken relative to the location specified by `origin`. Subsequent reading or writing will begin at that position. `offset` is a `long`; `fd` and `origin` are `int`'s. `origin` can be 0, 1, or 2 to specify that `offset` is to be measured from the beginning, from the current position, or from the end of the file respectively. For example, to append to a file, seek to the end before writing:

```
lseek(fd, 0L, 2);
```

To get back to the beginning ("rewind"),

```
lseek(fd, 0L, 0);
```

Notice the `0L` argument; it could also be written as `(long) 0`.

With `lseek`, it is possible to treat files more or less like large arrays, at the price of slower access. For example, the following simple function reads any number of bytes from any arbitrary place in a file.

```
get(fd, pos, buf, n) /* read n bytes from position pos */
int fd, n;
long pos;
char *buf;
{
	lseek(fd, pos, 0);  /* get to pos */
	return(read(fd, buf, n));
}
```

In pre-version 7 UNIX, the basic entry point to the I/O system is called `seek`. `seek` is identical to `lseek`, except that its `offset` argument is an `int` rather than a `long`. Accordingly, since PDP-11 integers have only 16 bits, the `offset` specified for `seek` is limited to 65,535; for this reason, `origin` values of 3, 4, 5 cause `seek` to multiply the given offset by 512 (the number of bytes in one physical block) and then interpret `origin` as if it were 0, 1, or 2 respectively. Thus to get to an arbitrary place in a large file requires two seeks, first one which selects the block, then one which has `origin` equal to 1 and moves to the desired byte within the block.

Exercise 8-2. Clearly, `seek` can be written in terms of `lseek`, and vice versa. Write each in terms of the other. □

8.5 Example — An Implementation of Fopen and Getc

Let us illustrate how some of these pieces fit together by showing an implementation of the standard library routines `fopen` and `getc`.

Recall that files in the standard library are described by file pointers rather than file descriptors. A file pointer is a pointer to a structure that contains several pieces of information about the file: a pointer to a buffer, so the file can be read in large chunks; a count of the number of characters left in the buffer; a pointer to the next character position in the buffer; some flags describing read/write mode, etc.; and the file descriptor.

The data structure that describes a file is contained in the file `stdio.h`, which must be included (by `#include`) in any source file that uses routines from the standard library. It is also included by functions in that library. In the following excerpt from `stdio.h`, names which are intended for use only by functions of the library begin with an underscore so they are less likely to collide with names in a user's program.

```
#define   _BUFSIZE  512
#define   _NFILE    20    /* #files that can be handled */

typedef struct _iobuf {
     char *_ptr;     /* next character position */
     int  _cnt;      /* number of characters left */
     char *_base;    /* location of buffer */
     int  _flag;     /* mode of file access */
     int  _fd;       /* file descriptor */
} FILE;
extern FILE _iob[_NFILE];

#define   stdin     (&_iob[0])
#define   stdout    (&_iob[1])
#define   stderr    (&_iob[2])

#define   _READ     01    /* file open for reading */
#define   _WRITE    02    /* file open for writing */
#define   _UNBUF    04    /* file is unbuffered */
#define   _BIGBUF   010   /* big buffer allocated */
#define   _EOF 020  /* EOF has occurred on this file */
#define   _ERR 040  /* error has occurred on this file */
#define   NULL 0
#define   EOF  (-1)
```

```
#define   getc(p)    (--(p)->_cnt >= 0 \
                ? *(p)->_ptr++ & 0377 : _fillbuf(p))
#define   getchar() getc(stdin)

#define   putc(x,p) (--(p)->_cnt >= 0 \
                ? *(p)->_ptr++ = (x) : _flushbuf((x),p))
#define   putchar(x)     putc(x,stdout)
```

The `getc` macro normally just decrements the count, advances the pointer, and returns the character. (A long `#define` is continued with a backslash.) If the count goes negative, however, `getc` calls the function `_fillbuf` to replenish the buffer, re-initialize the structure contents, and return a character. A function may present a portable interface, yet itself contain non-portable constructs: `getc` masks the character with `0377`, which defeats the sign extension done by the PDP-11 and ensures that all characters will be positive.

Although we will not discuss any details, we have included the definition of `putc` to show that it operates in much the same way as `getc`, calling a function `_flushbuf` when its buffer is full.

The function `fopen` can now be written. Most of `fopen` is concerned with getting the file opened and positioned at the right place, and setting the flag bits to indicate the proper state. `fopen` does not allocate any buffer space; this is done by `_fillbuf` when the file is first read.

```
#include  <stdio.h>
#define   PMODE     0644 /* R/W for owner; R for others */

FILE *fopen(name, mode)  /* open file, return file ptr */
register char *name, *mode;
{
     register int fd;
     register FILE *fp;

     if (*mode != 'r' && *mode != 'w' && *mode != 'a') {
          fprintf(stderr, "illegal mode %s opening %s\n",
               mode, name);
          exit(1);
     }
     for (fp = _iob; fp < _iob + _NFILE; fp++)
          if ((fp->_flag & (_READ | _WRITE)) == 0)
               break;    /* found free slot */
     if (fp >= _iob + _NFILE) /* no free slots */
          return(NULL);

     if (*mode == 'w')   /* access file */
          fd = creat(name, PMODE);
     else if (*mode == 'a') {
          if ((fd = open(name, 1)) == -1)
               fd = creat(name, PMODE);
          lseek(fd, 0L, 2);
     } else
          fd = open(name, 0);
     if (fd == -1)  /* couldn't access name */
          return(NULL);

     fp->_fd = fd;
     fp->_cnt = 0;
     fp->_base = NULL;
     fp->_flag &= ~(_READ | _WRITE);
     fp->_flag |= (*mode == 'r') ? _READ : _WRITE;
     return(fp);
}
```

The function `_fillbuf` is rather more complicated. The main complexity lies in the fact that `_fillbuf` attempts to permit access to the file even though there may not be enough memory to buffer the I/O. If space for a new buffer can be obtained from `calloc`, all is well; if not, `_fillbuf` does unbuffered I/O using a single character stored in a private array.

```
#include  <stdio.h>

_fillbuf(fp)  /* allocate and fill input buffer */
register FILE *fp;
{
     static char smallbuf[_NFILE]; /* for unbuffered I/O */
     char *calloc();

     if ((fp->_flag&_READ)==0 || (fp->_flag&(_EOF|_ERR))!=0)
          return(EOF);
     while (fp->_base == NULL)  /* find buffer space */
          if (fp->_flag & _UNBUF)  /* unbuffered */
               fp->_base = &smallbuf[fp->_fd];
          else if ((fp->_base=calloc(_BUFSIZE, 1)) == NULL)
               fp->_flag |= _UNBUF; /* can't get big buf */
          else
               fp->_flag |= _BIGBUF; /* got big one */
     fp->_ptr = fp->_base;
     fp->_cnt = read(fp->_fd, fp->_ptr,
                    fp->_flag & _UNBUF ? 1 : _BUFSIZE);
     if (--fp->_cnt < 0) {
          if (fp->_cnt == -1)
               fp->_flag |= _EOF;
          else
               fp->_flag |= _ERR;
          fp->_cnt = 0;
          return(EOF);
     }
     return(*fp->_ptr++ & 0377);  /* make char positive */
}
```

The first call to `getc` for a particular file finds a count of zero, which forces a call of `_fillbuf`. If `_fillbuf` finds that the file is not open for reading, it returns `EOF` immediately. Otherwise, it tries to allocate a large buffer, and, failing that, a single character buffer, setting the buffering information in `_flag` appropriately.

Once the buffer is established, `_fillbuf` simply calls `read` to fill it, sets the count and pointers, and returns the character at the beginning of the buffer. Subsequent calls to `_fillbuf` will find a buffer allocated.

The only remaining loose end is how everything gets started. The array `_iob` must be defined and initialized for `stdin`, `stdout` and `stderr`:

```
FILE _iob[_NFILE] ={
     { NULL, 0, NULL, _READ, 0 },  /* stdin */
     { NULL, 0, NULL, _WRITE, 1 }, /* stdout */
     { NULL, 0, NULL, _WRITE | _UNBUF, 2 }   /* stderr */
};
```