Notes By: Shailesh Bdr. Pandey, TA, Computer Engineering Department, Nepal Engineering College

Data File and File Handling

Types of Disk Files

Text streams are associated with text-mode files. Text-mode files consist of a sequence of lines. Each line contains zero or more characters and ends with one or more characters that signal end-of-line. The maximum line length is 255 characters. It's important to remember that a "line" isn't a C string; there is no terminating NULL character (\0). When you use a text-mode stream, translation occurs between C's newline character (\n) and whatever character(s) the operating system uses to mark end-of-line on disk files. On DOS systems, it's a carriage-return linefeed (CR-LF) combination. When data is written to a text-mode file, each \n is translated to a CR-LF; when data is read from a disk file, each CR-LF is translated to a \n. On UNIX systems, no translation is done--newline characters remain unchanged.

Binary streams are associated with binary-mode files. Any and all data is written and read unchanged, with no separation into lines and no use of end-of-line characters. The NULL and end-of-line characters have no special significance and are treated like any other byte of data. Some file input/output functions are restricted to one file mode, whereas other functions can use either mode. This chapter teaches you which mode to use with which functions.

Filenames

Every disk file has a name, and you must use filenames when dealing with disk files. Filenames are stored as strings, just like other text data. The rules as to what is acceptable for filenames and what is not differ from one operating system to another.

A filename in a C program also can contain path information. The path specifies the drive and/or directory (or folder) where the file is located. If you specify a filename without a path, it will be assumed that the file is located at whatever location the operating system currently designates as the default. It's good programming practice to always specify path information as part of your filenames.

On PCs, the backslash character is used to separate directory names in a path. For example, to DOS and Windows, the name *"c:\data\list.txt"* refers to a file named LIST.TXT in the directory \DATA on drive C. Remember that the backslash character has a special meaning to C when it's in a string. To represent the backslash character itself, you must precede it with another backslash. Thus, in a C program, you would represent the filename as follows:

```
char *filename = "c:\\data\\list.txt";
```

If you're entering a filename using the keyboard, however, enter only a single backslash. Not all systems use the backslash as the directory separator. For example, UNIX uses the forward slash (/).

Opening a File

The process of creating a stream linked to a disk file is called opening the file. When you open a file, it becomes available for reading (meaning that data is input from the file to the program), writing (meaning that data from the program is saved in the file), or both. When you're done using the file, you must close it. To open a file, you use the fopen() library function. The prototype of fopen() is located in STDIO.H and reads as follows:

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

This prototype tells you that fopen() returns a pointer to type FILE, which is a structure declared in STDIO.H. The members of the FILE structure are used by the program in the various file access operations, but you don't need to be concerned about them. However, for each file that you want to open, you must declare a pointer to type FILE. When you call fopen(), that function creates an instance of the FILE structure and returns a pointer to that structure. You use this pointer in all subsequent operations on the file. If fopen() fails, it returns NULL. Such a failure could be caused, for example, by a hardware error or by trying to open a file on a diskette that hasn't been formatted.

The argument filename is the name of the file to be opened. As noted earlier, filename can--and should--contain a path specification. The filename argument can be a literal string enclosed in double quotation marks or a pointer to a string variable.

The argument mode specifies the mode in which to open the file. In this context, mode controls whether the file is binary or text and whether it is for reading, writing, or both. The permitted values for mode are listed in Table 16.1.

Table 16.1. Values of mode for the fopen() function.

mode	*Meaning*
r	Opens the file for reading. If the file doesn't exist, fopen() returns NULL.
w	Opens the file for writing. If a file of the specified name doesn't exist, it is created. If a file of the specified name does exist, it is deleted without warning, and a new, empty file is created.
a	Opens the file for appending. If a file of the specified name doesn't exist, it is created. If the file does exist, new data is appended to the end of the file.
r+	Opens the file for reading and writing. If a file of the specified name doesn't exist, it is created. If the file does exist, new data is added to the beginning of the file, overwriting existing data.
w+	Opens the file for reading and writing. If a file of the specified name doesn't exist, it is created. If the file does exist, it is overwritten.
a+	Opens a file for reading and appending. If a file of the specified name doesn't exist, it is created. If the file does exist, new data is appended to the end of the file.

The default file mode is text. To open a file in binary mode, you append a b to the mode argument. Thus, a mode argument of a would open a text-mode file for appending, whereas ab would open a binary-mode file for appending.

Remember that fopen() returns NULL if an error occurs. Error conditions that can cause a return value of NULL include the following:

* Using an invalid filename.
* Trying to open a file on a disk that isn't ready (the drive door isn't closed or the disk isn't formatted, for example).
* Trying to open a file in a nonexistent directory or on a nonexistent disk drive.
* Trying to open a nonexistent file in mode r.

Whenever you use fopen(), you need to test for the occurrence of an error. There's no way to tell exactly which error occurred, but you can display a message to the user and try to open the file again, or you can end the program. Most C compilers include non-ANSI extensions that let you obtain information about the nature of the error; refer to your compiler documentation for information. Listing 16.1 demonstrates fopen().

Listing 16.1. Using fopen() to open disk files in various modes.

```
1:  /* Demonstrates the fopen() function. */
2:  #include <stdlib.h>
3:  #include <stdio.h>
4:
5:  main()
6:  {
7:      FILE *fp;
8:      char filename[40], mode[4];
9:
10:     while (1)
11:     {
12:
13:         /* Input filename and mode. */
14:
15:         printf("\nEnter a filename: ");
16:         gets(filename);
17:         printf("\nEnter a mode (max 3 characters): ");
18:         gets(mode);
19:
20:         /* Try to open the file. */
21:
22:         if ( (fp = fopen( filename, mode )) != NULL )
23:         {
24:             printf("\nSuccessful opening %s in mode %s.\n",
25:                     filename, mode);
26:             fclose(fp);
27:             puts("Enter x to exit, any other to continue.");
```

```
28:           if ( (getc(stdin)) == `x')
29:              break;
30:           else
31:              continue;
32:        }
33:        else
34:        {
35:           fprintf(stderr, "\nError opening file %s in mode %s.\n",
36:                 filename, mode);
37:           puts("Enter x to exit, any other to try again.");
38:           if ( (getc(stdin)) == `x')
39:              break;
40:           else
41:              continue;
42:        }
43:     }
44: }
Enter a filename: junk.txt
Enter a mode (max 3 characters): w
Successful opening junk.txt in mode w.
Enter x to exit, any other to continue.
j
Enter a filename: morejunk.txt
Enter a mode (max 3 characters): r
Error opening morejunk.txt in mode r.
Enter x to exit, any other to try again.
x
```

ANALYSIS: This program prompts you for both the filename and the mode specifier on lines 15 through 18. After getting the names, line 22 attempts to open the file and assign its file pointer to fp. As an example of good programming practice, the if statement on line 22 checks to see that the opened file's pointer isn't equal to NULL. If fp isn't equal to NULL, a message stating that the open was successful and that the user can continue is printed. If the file pointer is NULL, the else condition of the if loop executes. The else condition on lines 33 through 42 prints a message stating that there was a problem. It then prompts the user to determine whether the program should continue.

You can experiment with different names and modes to see which ones give you an error. In the output just shown, you can see that trying to open MOREJUNK.TXT in mode r resulted in an error because the file didn't exist on the disk. If an error occurs, you're given the choice of entering the information again or quitting the program. To force an error, you could enter an invalid filename such as [].

Writing and Reading File Data

A program that uses a disk file can write data to a file, read data from a file, or a combination of the two. You can write data to a disk file in three ways:

* You can use formatted output to save formatted data to a file. You should use formatted output only with text-mode files. The primary use of formatted output is to create files containing text and numeric data to be read by other programs such as spreadsheets or databases. You rarely, if ever, use formatted output to create a file to be read again by a C program.
* You can use character output to save single characters or lines of characters to a file. Although technically it's possible to use character output with binary-mode files, it can be tricky. You should restrict character-mode output to text files. The main use of character output is to save text (but not numeric) data in a form that can be read by C, as well as other programs such as word processors.
* You can use direct output to save the contents of a section of memory directly to a disk file. This method is for binary files only. Direct output is the best way to save data for later use by a C program.

When you want to read data from a file, you have the same three options: formatted input, character input, or direct input. The type of input you use in a particular case depends almost entirely on the nature of the file being read. Generally, you will read data in the same mode that it was saved in, but this is not a requirement. However, reading a file in a mode different from the one it was written in requires a thorough knowledge of C and file formats.

The previous descriptions of the three types of file input and output suggest tasks best suited for each type of output. This is by no means a set of strict rules. The C language is very flexible (this is one of its advantages!), so a clever programmer can make any type of file output suit almost any need. As a beginning programmer, it might make things easier if you follow these guidelines, at least initially.

Formatted File Input and Output

Formatted file input/output deals with text and numeric data that is formatted in a specific way. It is directly analogous to formatted keyboard input and screen output done with the printf() and scanf() functions, as described on Day 14. I'll discuss formatted output first, followed by input.

Formatted File Output

Formatted file output is done with the library function fprintf(). The prototype of fprintf() is in the header file STDIO.H, and it reads as follows:

```
int fprintf(FILE *fp, char *fmt, ...);
```

The first argument is a pointer to type FILE. To write data to a particular disk file, you pass the pointer that was returned when you opened the file with fopen().

The second argument is the format string. The format string used by fprintf() follows exactly the same rules as printf().

The final argument is .... What does that mean? In a function prototype, ellipses represent a variable number of arguments. In other words, in addition to the file

pointer and the format string arguments, fprintf() takes zero, one, or more additional arguments. This is just like printf(). These arguments are the names of the variables to be output to the specified stream.

Remember, fprintf() works just like printf(), except that it sends its output to the stream specified in the argument list. In fact, if you specify a stream argument of stdout, fprintf() is identical to printf(). Listing 16.2 demonstrates the use of fprintf().

Listing 16.2. The equivalence of fprintf() formatted output to both a file and to stdout.

```
1:  /* Demonstrates the fprintf() function. */
2:  #include <stdlib.h>
3:  #include <stdio.h>
4:
5:  void clear_kb(void);
6:
7:  main()
8:  {
9:      FILE *fp;
10:     float data[5];
11:     int count;
12:     char filename[20];
13:
14:     puts("Enter 5 floating-point numerical values.");
15:
16:     for (count = 0; count < 5; count++)
17:        scanf("%f", &data[count]);
18:
19:     /* Get the filename and open the file. First clear stdin */
20:     /* of any extra characters. */
21:
22:     flushall();
23:
24:     puts("Enter a name for the file.");
25:     gets(filename);
26:
27:     if ( (fp = fopen(filename, "w")) == NULL)
28:     {
29:        fprintf(stderr, "Error opening file %s.", filename);
30:        exit(1);
31:     }
32:
33:     /* Write the numerical data to the file and to stdout. */
34:
35:     for (count = 0; count < 5; count++)
36:     {
37:        fprintf(fp, "\ndata[%d] = %f", count, data[count]);
38:        fprintf(stdout, "\ndata[%d] = %f", count, data[count]);
```

```
39:     }
40:     fclose(fp);
41:     printf("\n");
42:     return(0);
43: }
44:
45: void clear_kb(void)
46: /* Clears stdin of any waiting characters. */
47: {
48:     char junk[80];
49:     gets(junk);
50: }
Enter 5 floating-point numerical values.
3.14159
9.99
1.50
3.
1000.0001
Enter a name for the file.
numbers.txt
data[0] = 3.141590
data[1] = 9.990000
data[2] = 1.500000
data[3] = 3.000000
data[4] = 1000.000122
```

ANALYSIS: You might wonder why the program displays 1000.000122 when the value you entered was 1000.0001. This isn't an error in the program. It's a normal consequence of the way C stores numbers internally. Some floating-point values can't be stored exactly, so minor inaccuracies such as this one sometimes result.

This program uses fprintf() on lines 37 and 38 to send some formatted text and numeric data to stdout and to the disk file whose name you specified. The only difference between the two lines is the first argument--that is, the stream to which the data is sent. After running the program, use your editor to look at the contents of the file NUMBERS.TXT (or whatever name you assigned to it), which will be in the same directory as the program files. You'll see that the text in the file is an exact copy of the text that was displayed on-screen.

Note that Listing 16.2 uses the flushall( ) function This is necessary to remove from the keyboard buffer any extra characters that might be left over from the call to scanf(). If you don't clear the keyboard buffer, these extra characters (specifically, the newline) are read by the gets( ) that inputs the filename, and the result is a file creation error.

Formatted File Input

For formatted file input, use the fscanf() library function, which is used like scanf()except that input comes from a specified stream instead of from stdin. The prototype for fscanf() is

int fscanf(FILE *fp, const char *fmt, ...);

The argument fp is the pointer to type FILE returned by fopen(), and fmt is a pointer to the format string that specifies how fscanf() is to read the input. The components of the format string are the same as for scanf(). Finally, the ellipses (...) indicate one or more additional arguments, the addresses of the variables where fscanf() is to assign the input. The function fscanf() works exactly the same as scanf(), except that characters are taken from the specified stream rather than from stdin.

To demonstrate fscanf(), you need a text file containing some numbers or strings in a format that can be read by the function. Use your editor to create a file named INPUT.TXT, and enter five floating-point numbers with some space between them (spaces or newlines). For example, your file might look like this:

123.45 87.001
100.02
0.00456 1.0005

Now, compile and run Listing 16.3.

Listing 16.3. Using fscanf() to read formatted data from a disk file.

```
1:  /* Reading formatted file data with fscanf(). */
2:  #include <stdlib.h>
3:  #include <stdio.h>
4:
5:  main()
6:  {
7:      float f1, f2, f3, f4, f5;
8:      FILE *fp;
9:
10:     if ( (fp = fopen("INPUT.TXT", "r")) == NULL)
11:     {
12:         fprintf(stderr, "Error opening file.\n");
13:         exit(1);
14:     }
15:
16:     fscanf(fp, "%f %f %f %f %f", &f1, &f2, &f3, &f4, &f5);
17:     printf("The values are %f, %f, %f, %f, and %f\n.",
18:             f1, f2, f3, f4, f5);
19:
20:     fclose(fp);
21:      return(0);
22: }
```

The values are 123.45, 87.0001, 100.02, 0.00456, and 1.0005.

ANALYSIS: This program reads the five values from the file you created and then displays them on-screen. The fopen() call on line 10 opens the file for read mode. It also checks to see that the file opened correctly. If the file wasn't opened, an error message is displayed on line 12, and the program exits (line 13). Line 16 demonstrates the use of the fscanf() function. With the exception of the first parameter, fscanf() is identical to scanf(). The first parameter points to the file that you want the program to read. You can do further experiments with fscanf(), creating input files with your programming editor and seeing how fscanf() reads the data.

Character Input and Output

When used with disk files, the term character I/O refers to single characters as well as lines of characters. Remember, a line is a sequence of zero or more characters terminated by the newline character. Use character I/O with text-mode files. The following sections describe character input/output functions, and then you'll see a demonstration program.

Character Input

There are three character input functions: getc() and fgetc() for single characters, and fgets() for lines.

The getc( ) and fgetc( ) Functions

The functions getc() and fgetc() are identical and can be used interchangeably. They input a single character from the specified stream. Here is the prototype of getc(), which is in STDIO.H:

int getc(FILE *fp);

The argument fp is the pointer returned by fopen() when the file is opened. The function returns the character that was input or EOF on error.

This is another example of the flexibility of C's streams--the same function can be used for keyboard or file input.

If getc() and fgetc() return a single character, why are they prototyped to return a type int? The reason is that, when reading files, you need to be able to read in the end-of-file marker, which on some systems isn't a type char but a type int. You'll see getc() in action later, in Listing 16.10.

The fgets() Function

To read a line of characters from a file, use the fgets() function. The prototype is

char *fgets(char *str, int n, FILE *fp);

The argument str is a pointer to a buffer in which the input is to be stored, n is the maximum number of characters to be input, and fp is the pointer to type FILE that was returned by fopen() when the file was opened.

When called, fgets() reads characters from fp into memory, starting at the location pointed to by str. Characters are read until a newline is encountered or until n-1 characters have been read, whichever occurs first. By setting n equal to the number of bytes allocated for the buffer str, you prevent input from overwriting memory beyond allocated space. (The n-1 is to allow space for the terminating \0 that fgets() adds to the end of the string.) If successful, fgets() returns str. Two types of errors can occur, as indicated by the return value of NULL:
* If a read error or EOF is encountered before any characters have been assigned to str, NULL is returned, and the memory pointed to by str is unchanged.
* If a read error or EOF is encountered after one or more characters have been assigned to str, NULL is returned, and the memory pointed to by str contains garbage.

You can see that fgets() doesn't necessarily input an entire line (that is, everything up to the next newline character). If n-1 characters are read before a newline is encountered, fgets() stops. The next read operation from the file starts where the last one leaves off. To be sure that fgets() reads in entire strings, stopping only at newlines, be sure that the size of your input buffer and the corresponding value of n passed to fgets() are large enough.

Character Output

You need to know about two character output functions: putc() and fputs().

The putc() Function

The library function putc() writes a single character to a specified stream. Its prototype in STDIO.H reads

int putc(int ch, FILE *fp);

The argument ch is the character to output. As with other character functions, it is formally called a type int, but only the lower-order byte is used. The argument fp is the pointer associated with the file (the pointer returned by fopen() when the file was opened). The function putc() returns the character just written if successful or EOF if an error occurs. The symbolic constant EOF is defined in STDIO.H, and it has the value -1. Because no "real" character has that numeric value, EOF can be used as an error indicator (with text-mode files only).

The fputs() Function

To write a line of characters to a stream, use the library function fputs(). This function works just like puts(). The only difference is that with fputs() you can specify the output stream. Also, fputs() doesn't add a newline to the end of the string; if you want it, you must explicitly include it. Its prototype in STDIO.H is

char fputs(char *str, FILE *fp);

The argument str is a pointer to the null-terminated string to be written, and fp is the pointer to type FILE returned by fopen() when the file was opened. The string pointed to by str is written to the file, minus its terminating \0. The function fputs() returns a nonnegative value if successful or EOF on error.

Direct File Input and Output

You use direct file I/O most often when you save data to be read later by the same or a different C program. Direct I/O is used only with binary-mode files. With direct output, blocks of data are written from memory to disk. Direct input reverses the process: A block of data is read from a disk file into memory. For example, a single direct-output function call can write an entire array of type double to disk, and a single direct-input function call can read the entire array from disk back into memory. The direct I/O functions are fread() and fwrite().

The fwrite() Function

The fwrite() library function writes a block of data from memory to a binary-mode file. Its prototype in STDIO.H is

int fwrite(void *buf, int size, int count, FILE *fp);

The argument buf is a pointer to the region of memory holding the data to be written to the file. The pointer type is void; it can be a pointer to anything.

The argument size specifies the size, in bytes, of the individual data items, and count specifies the number of items to be written. For example, if you wanted to save a 100-element integer array, size would be 2 (because each int occupies 2 bytes) and count would be 100 (because the array contains 100 elements). To obtain the size argument, you can use the sizeof() operator.

The argument fp is, of course, the pointer to type FILE, returned by fopen() when the file was opened. The fwrite() function returns the number of items written on success; if the value returned is less than count, it means that an error has occurred. To check for errors, you usually program fwrite() as follows:

```
if( (fwrite(buf, size, count, fp)) != count)
        fprintf(stderr, "Error writing to file.");
```

Here are some examples of using fwrite(). To write a single type double variable x to a file, use the following:

fwrite(&x, sizeof(double), 1, fp);

To write an array data[] of 50 structures of type address to a file,
you have two choices:

```
fwrite(data, sizeof(address), 50, fp);
fwrite(data, sizeof(data), 1, fp);
```

The first method writes the array as 50 elements, with each element having the size of a single type address structure. The second method treats the array as a single element. The two methods accomplish exactly the same thing.

The following section explains fread() and then presents a program demonstrating fread() and fwrite().

The fread() Function

The fread() library function reads a block of data from a binary-mode file into memory. Its prototype in STDIO.H is

```
int fread(void *buf, int size, int count, FILE *fp);
```

The argument buf is a pointer to the region of memory that receives the data read from the file. As with fwrite(), the pointer type is void.

The argument size specifies the size, in bytes, of the individual data items being read, and count specifies the number of items to read. Note how these arguments parallel the arguments used by fwrite(). Again, the sizeof() operator is typically used to provide the size of the argument. The argument fp is (as always) the pointer to type FILE that was returned by fopen() when the file was opened. The fread() function returns the number of items read; this can be less than count if end-of-file was reached or an error occurred. Listing 16.4 demonstrates the use of fwrite() and fread().

Listing 16.4. Using fwrite() and fread() for direct file access.

```
1:  /* Direct file I/O with fwrite() and fread(). */
2:  #include <stdlib.h>
3:  #include <stdio.h>
4:
5:  #define SIZE 20
6:
7:  main()
8:  {
9:      int count, array1[SIZE], array2[SIZE];
10:     FILE *fp;
11:
12:     /* Initialize array1[]. */
13:
14:     for (count = 0; count < SIZE; count++)
15:         array1[count] = 2 * count;
16:
17:     /* Open a binary mode file. */
18:
19:     if ( (fp = fopen("direct.txt", "wb")) == NULL)
```

```
20:     {
21:        fprintf(stderr, "Error opening file.");
22:        exit(1);
23:     }
24:     /* Save array1[] to the file. */
25:
26:     if (fwrite(array1, sizeof(int), SIZE, fp) != SIZE)
27:     {
28:        fprintf(stderr, "Error writing to file.");
29:        exit(1);
30:     }
31:
32:     fclose(fp);
33:
34:     /* Now open the same file for reading in binary mode. */
35:
36:     if ( (fp = fopen("direct.txt", "rb")) == NULL)
37:     {
38:        fprintf(stderr, "Error opening file.");
39:        exit(1);
40:     }
41:
42:     /* Read the data into array2[]. */
43:
44:     if (fread(array2, sizeof(int), SIZE, fp) != SIZE)
45:     {
46:        fprintf(stderr, "Error reading file.");
47:        exit(1);
48:     }
49:
50:     fclose(fp);
51:
52:     /* Now display both arrays to show they're the same. */
53:
54:     for (count = 0; count < SIZE; count++)
55:        printf("%d\t%d\n", array1[count], array2[count]);
56:     return(0);
57: }
0       0
2       2
4       4
6       6
8       8
10      10
12      12
14      14
16      16
18      18
20      20
```

```
22    22
24    24
26    26
28    28
30    30
32    32
34    34
36    36
38    38
```

ANALYSIS: Listing 16.4 demonstrates the use of the fwrite() and fread() functions. This program initializes an array on lines 14 and 15. It then uses fwrite() on line 26 to save the array to disk. The program uses fread() on line 44 to read the data into a different array. Finally, it displays both arrays on-screen to show that they now hold the same data (lines 54 and 55).

When you save data with fwrite(), not much can go wrong besides some type of disk error. With fread(), you need to be careful, however. As far as fread() is concerned, the data on the disk is just a sequence of bytes. The function has no way of knowing what it represents. For example, on a 16-bit system, a block of 100 bytes could be 100 char variables, 50 int variables, 25 long variables, or 25 float variables. If you ask fread() to read that block into memory, it obediently does so. However, if the block was saved from an array of type int and you retrieve it into an array of type float, no error occurs, but you get strange results. When writing programs, you must be sure that fread() is used properly, reading data into the appropriate types of variables and arrays. Notice that in Listing 16.4, all calls to fopen(), fwrite(), and fread() are checked to ensure that they worked correctly.

File Buffering: Closing and Flushing Files

When you're done using a file, you should close it using the fclose() function. You saw fclose() used in programs presented earlier in this chapter. Its prototype is

```
int fclose(FILE *fp);
```

The argument fp is the FILE pointer associated with the stream; fclose() returns 0 on success or -1 on error. When you close a file, the file's buffer is flushed (written to the file). You can also close all open streams except the standard ones (stdin, stdout, stdprn, stderr, and stdaux) by using the fcloseall() function. Its prototype is

```
int fcloseall(void);
```

This function also flushes any stream buffers and returns the number of streams closed.

When a program terminates (either by reaching the end of main() or by executing the exit() function), all streams are automatically flushed and closed. However, it's a good idea to close streams explicitly--particularly those linked to disk files--as soon as you're finished with them. The reason has to do with stream buffers.

When you create a stream linked to a disk file, a buffer is automatically created and associated with the stream. A buffer is a block of memory used for temporary storage of data being written to and read from the file. Buffers are needed because disk drives are block-oriented devices, which means that they operate most efficiently when data is read and written in blocks of a certain size. The size of the ideal block differs, depending on the specific hardware in use. It's typically on the order of a few hundred to a thousand bytes. You don't need to be concerned about the exact block size, however.

The buffer associated with a file stream serves as an interface between the stream (which is character-oriented) and the disk hardware (which is block-oriented). As your program writes data to the stream, the data is saved in the buffer until the buffer is full, and then the
entire contents of the buffer are written, as a block, to the disk. An analogous process occurs when reading data from a disk file. The creation and operation of the buffer are handled by the operating system and are entirely automatic; you don't have to be concerned with them. (C does offer some functions for buffer manipulation, but they are beyond the scope of this book.)

In practical terms, this buffer operation means that, during program execution, data that your program wrote to the disk might still be in the buffer, not on the disk. If your program hangs up, if there's a power failure, or if some other problem occurs, the data that's still in the buffer might be lost, and you won't know what's contained in the disk file.

You can flush a stream's buffers without closing it by using the fflush() or flushall() library functions. Use fflush() when you want a file's buffer to be written to disk while still using the file. Use flushall() to flush the buffers of all open streams. The prototypes of these two functions are as follows:

```
int fflush(FILE *fp);
int flushall(void);
```

The argument fp is the FILE pointer returned by fopen() when the file was opened. If a file was opened for writing, fflush() writes its buffer to disk. If the file was opened for reading, the buffer is cleared. The function fflush() returns 0 on success or EOF if an error occurred. The function flushall() returns the number of open streams.

Sequential Versus Random File Access

Every open file has a file position indicator associated with it. The position indicator specifies where read and write operations take place in the file. The position is always given in terms of bytes from the beginning of the file. When a new file is opened, the position indicator is always at the beginning of the file, position 0. (Because the file is new and has a length of 0, there's no other location to indicate.) When an existing file is opened, the position indicator is at the end of the file if the file was opened in append mode, or at the beginning of the file if the file was opened in any other mode.

The file input/output functions covered earlier in this chapter make use of the position indicator, although the manipulations go on behind the scenes. Writing and reading operations occur at the location of the position indicator and update the position indicator as well. For example, if you open a file for reading, and 10 bytes are read, you input the first 10 bytes in the file (the bytes at positions 0 through 9). After the read operation, the position indicator is at position 10, and the next read operation begins there. Thus, if you want to read all the data in a file sequentially or write data to a file sequentially, you don't need to be concerned about the position indicator, because the stream I/O functions take care of it automatically.

When you need more control, use the C library functions that let you determine and change the value of the file position indicator. By controlling the position indicator, you can per-form random file access. Here, random means that you can read data from, or write data to, any position in a file without reading or writing all the preceding data.

The ftell() and rewind() Functions

To set the position indicator to the beginning of the file, use the library function rewind(). Its prototype, in STDIO.H, is

void rewind(FILE *fp);

The argument fp is the FILE pointer associated with the stream. After rewind() is called, the file's position indicator is set to the beginning of the file (byte 0). Use rewind() if you've read some data from a file and you want to start reading from the beginning of the file again without closing and reopening the file.

To determine the value of a file's position indicator, use ftell(). This function's prototype, located in STDIO.H, reads

long ftell(FILE *fp);

The argument fp is the FILE pointer returned by fopen() when the file was opened. The function ftell() returns a type long that gives the current file position in bytes from the start of the file (the first byte is at position 0). If an error occurs, ftell() returns -1L (a type long -1).

To get a feel for the operation of rewind() and ftell(), look at Listing 16.5.

Listing 16.5. Using ftell() and rewind().

```
1:  /* Demonstrates ftell() and rewind(). */
2:  #include <stdlib.h>
3:  #include <stdio.h>
4:
5:  #define BUFLEN 6
6:
7:  char msg[] = "abcdefghijklmnopqrstuvwxyz";
```

```
8:
9:  main()
10: {
11:     FILE *fp;
12:     char buf[BUFLEN];
13:
14:     if ( (fp = fopen("TEXT.TXT", "w")) == NULL)
15:     {
16:        fprintf(stderr, "Error opening file.");
17:        exit(1);
18:     }
19:
20:     if (fputs(msg, fp) == EOF)
21:     {
22:        fprintf(stderr, "Error writing to file.");
23:        exit(1);
24:     }
25:
26:     fclose(fp);
27:
28:     /* Now open the file for reading. */
29:
30:     if ( (fp = fopen("TEXT.TXT", "r")) == NULL)
31:     {
32:        fprintf(stderr, "Error opening file.");
33:        exit(1);
34:     }
35:     printf("\nImmediately after opening, position = %ld", ftell(fp));
36:
37:     /* Read in 5 characters. */
38:
39:     fgets(buf, BUFLEN, fp);
40:     printf("\nAfter reading in %s, position = %ld", buf, ftell(fp));
41:
42:     /* Read in the next 5 characters. */
43:
44:     fgets(buf, BUFLEN, fp);
45:     printf("\n\nThe next 5 characters are %s, and position now = %ld",
46:           buf, ftell(fp));
47:
48:     /* Rewind the stream. */
49:
50:     rewind(fp);
51:
52:     printf("\n\nAfter rewinding, the position is back at %ld",
53:           ftell(fp));
54:
55:     /* Read in 5 characters. */
56:
```

```
57:     fgets(buf, BUFLEN, fp);
58:     printf("\nand reading starts at the beginning again: %s\n", buf);
59:     fclose(fp);
60:     return(0);
61: }
```

```
Immediately after opening, position = 0
After reading in abcde, position = 5
The next 5 characters are fghij, and position now = 10
After rewinding, the position is back at 0
and reading starts at the beginning again: abcde
```

ANALYSIS: This program writes a string, msg, to a file called TEXT.TXT. The message consists of the 26 letters of the alphabet, in order. Lines 14 through 18 open TEXT.TXT for writing and test to ensure that the file was opened successfully. Lines 20 through 24 write msg to the file using fputs() and check to ensure that the write was successful. Line 26 closes the file with fclose(), completing the process of creating a file for the rest of the program to use.

Lines 30 through 34 open the file again, only this time for reading. Line 35 prints the return value of ftell(). Notice that this position is at the beginning of the file. Line 39 performs a gets() to read five characters. The five characters and the new file position are printed on line 40. Notice that ftell() returns the correct offset. Line 50 calls rewind() to put the pointer back at the beginning of the file, before line 52 prints the file position again. This should confirm for you that rewind() resets the position. An additional read on line 57 further confirms that the program is indeed back at the beginning of the file. Line 59 closes the file before ending the program.

The fseek() Function

More precise control over a stream's position indicator is possible with the fseek() library function. By using fseek(), you can set the position indicator anywhere in the file. The function prototype, in STDIO.H, is

```
int fseek(FILE *fp, long offset, int origin);
```

The argument fp is the FILE pointer associated with the file. The distance that the position indicator is to be moved is given by offset in bytes. The argument origin specifies the move's relative starting point. There can be three values for origin, with symbolic constants defined in IO.H, as shown in Table 16.2.

Table 16.2. Possible origin values for fseek().

Constant	Value	Description
SEEK_SET	0	Moves the indicator offset bytes from the beginning of the file.
SEEK_CUR	1	Moves the indicator offset bytes from its current position.
SEEK_END	2	Moves the indicator offset bytes from the end of the file.

The function fseek() returns 0 if the indicator was successfully moved or nonzero if an error occurred. Listing 16.6 uses fseek() for random file access.

Listing 16.6. Random file access with fseek().

```
1:  /* Random access with fseek(). */
2:
3:  #include <stdlib.h>
4:  #include <stdio.h>
5:
6:  #define MAX 50
7:
8:  main()
9:  {
10:    FILE *fp;
11:    int data, count, array[MAX];
12:    long offset;
13:
14:    /* Initialize the array. */
15:
16:    for (count = 0; count < MAX; count++)
17:       array[count] = count * 10;
18:
19:    /* Open a binary file for writing. */
20:
21:    if ( (fp = fopen("RANDOM.DAT", "wb")) == NULL)
22:    {
23:       fprintf(stderr, "\nError opening file.");
24:       exit(1);
25:    }
26:
27:    /* Write the array to the file, then close it. */
28:
29:    if ( (fwrite(array, sizeof(int), MAX, fp)) != MAX)
30:    {
31:       fprintf(stderr, "\nError writing data to file.");
32:       exit(1);
33:    }
34:
35:    fclose(fp);
36:
37:    /* Open the file for reading. */
38:
39:    if ( (fp = fopen("RANDOM.DAT", "rb")) == NULL)
40:    {
41:       fprintf(stderr, "\nError opening file.");
42:       exit(1);
43:    }
44:
```

```
45:    /* Ask user which element to read. Input the element */
46:    /* and display it, quitting when -1 is entered. */
47:
48:    while (1)
49:    {
50:       printf("\nEnter element to read, 0-%d, -1 to quit: ",MAX-1);
51:       scanf("%ld", &offset);
52:
53:       if (offset < 0)
54:          break;
55:       else if (offset > MAX-1)
56:          continue;
57:
58:       /* Move the position indicator to the specified element. */
59:
60:       if ( (fseek(fp, (offset*sizeof(int)), SEEK_SET)) != 0)
61:       {
62:          fprintf(stderr, "\nError using fseek().");
63:          exit(1);
64:       }
65:
66:       /* Read in a single integer. */
67:
68:       fread(&data, sizeof(int), 1, fp);
69:
70:       printf("\nElement %ld has value %d.", offset, data);
71:    }
72:
73:    fclose(fp);
74:    return(0);
75: }
Enter element to read, 0-49, -1 to quit: 5
Element 5 has value 50.
Enter element to read, 0-49, -1 to quit: 6
Element 6 has value 60.
Enter element to read, 0-49, -1 to quit: 49
Element 49 has value 490.
Enter element to read, 0-49, -1 to quit: 1
Element 1 has value 10.
Enter element to read, 0-49, -1 to quit: 0
Element 0 has value 0.
Enter element to read, 0-49, -1 to quit: -1
```

ANALYSIS: Lines 14 through 35 are similar to Listing 16.5. Lines 16 and 17 initialize an array called data with 50 type int values. The value stored in each array element is equal to 10 times the index. Then the array is written to a binary file called RANDOM.DAT. You know it is binary because the file was opened with mode "wb" on line 21.

Line 39 reopens the file in binary read mode before going into an infinite while loop. The while loop prompts users to enter the number of the array element that they want to read. Notice that lines 53 through 56 check to see that the entered element is within the range of the file. Does C let you read an element that is beyond the end of the file? Yes. Like going beyond the end of an array with values, C also lets you read beyond the end of a file. If you do read beyond the end (or before the beginning), your results are unpredictable. It's always best to check what you're doing (as lines 53 through 56 do in this listing).

After you have input the element to find, line 60 jumps to the appropriate offset with a call to fseek(). Because SEEK_SET is being used, the seek is done from the beginning of the file. Notice that the distance into the file is not just offset, but offset multiplied by the size of the elements being read. Line 68 then reads the value, and line 70 prints it.

Detecting the End of a File

Sometimes you know exactly how long a file is, so there's no need to be able to detect the file's end. For example, if you used fwrite() to save a 100-element integer array, you know the file is 200 bytes long (assuming 2-byte integers). At other times, however, you don't know how long the file is, but you still want to read data from the file, starting at the beginning and proceeding to the end. There are two ways to detect end-of-file.

When reading from a text-mode file character-by-character, you can look for the end-of-file character. The symbolic constant EOF is defined in STDIO.H as -1, a value never used by a "real" character. When a character input function reads EOF from a text-mode stream, you can be sure that you've reached the end of the file. For example, you could write the following:

```
while ( (c = fgetc( fp )) != EOF )
```

With a binary-mode stream, you can't detect the end-of-file by looking for -1, because a byte of data from a binary stream could have that value, which would result in premature end of input. Instead, you can use the library function feof(), which can be used for both binary- and text-mode files:

```
int feof(FILE *fp);
```

The argument fp is the FILE pointer returned by fopen() when the file was opened. The function feof() returns 0 if the end of file fp hasn't been reached, or a nonzero value if end-of-file has been reached. If a call to feof() detects end-of-file, no further read operations are permitted until a rewind() has been done, fseek() is called, or the file is closed and reopened.

Listing 16.7 demonstrates the use of feof(). When you're prompted for a filename, enter the name of any text file--one of your C source files, for example, or a header file such as STDIO.H. Just be sure that the file is in the current directory, or else enter

a path as part of the filename. The program reads the file one line at a time, displaying each line on stdout, until feof() detects end-of-file.

Listing 16.7. Using feof() to detect the end of a file.

```
1:  /* Detecting end-of-file. */
2:  #include <stdlib.h>
3:  #include <stdio.h>
4:
5:  #define BUFSIZE 100
6:
7:  main()
8:  {
9:      char buf[BUFSIZE];
10:     char filename[60];
11:     FILE *fp;
12:
13:     puts("Enter name of text file to display: ");
14:     gets(filename);
15:
16:     /* Open the file for reading. */
17:     if ( (fp = fopen(filename, "r")) == NULL)
18:     {
19:         fprintf(stderr, "Error opening file.");
20:         exit(1);
21:     }
22:
23:     /* If end of file not reached, read a line and display it. */
24:
25:     while ( !feof(fp) )
26:     {
27:         fgets(buf, BUFSIZE, fp);
28:         printf("%s",buf);
29:     }
30:
31:     fclose(fp);
32:     return(0);
33: }
Enter name of text file to display:
hello.c
#include <stdio.h>
main()
{
   printf("Hello, world.");
   return(0);
}
```

ANALYSIS: The while loop in this program (lines 25 through 29) is typical of a while used in more complex programs that do sequential processing. As long as the

end of the file hasn't been reached, the code within the while statement (lines 27 and 28) continues to execute repeatedly. When the call to feof() returns a nonzero value, the loop ends, the file is closed, and the program ends.

File Management Functions

The term file management refers to dealing with existing files—not reading from or writing to them, but deleting, renaming, and copying them. The C standard library contains functions for deleting and renaming files, and you can also write your own file-copying program.

Deleting a File

To delete a file, you use the library function remove(). Its prototype is in STDIO.H, as follows:

int remove(const char *filename);

The variable *filename is a pointer to the name of the file to be deleted. (See the section on filenames earlier in this chapter.) The specified file must not be open. If the file exists, it is deleted (just as if you used the DEL command from the DOS prompt or the rm command in UNIX), and remove() returns 0. If the file doesn't exist, if it's read-only, if you don't have sufficient access rights, or if some other error occurs, remove() returns -1.

Listing 16.8 demonstrates the use of remove(). Be careful: If you remove() a file, it's gone forever.

Listing 16.8. Using the remove() function to delete a disk file.

```
1:  /* Demonstrates the remove() function. */
2:
3:  #include <stdio.h>
4:
5:  main()
6:  {
7:      char filename[80];
8:
9:      printf("Enter the filename to delete: ");
10:     gets(filename);
11:
12:     if ( remove(filename) == 0)
13:         printf("The file %s has been deleted.\n", filename);
14:     else
15:         fprintf(stderr, "Error deleting the file %s.\n", filename);
16:     return(0);
17: }
Enter the filename to delete: *.bak
Error deleting the file *.bak.
```

```
Enter the filename to delete: list1414.bak
The file list1414.bak has been deleted.
```

ANALYSIS: This program prompts the user on line 9 for the name of the file to be deleted. Line 12 then calls remove() to delete the entered file. If the return value is 0, the file was removed, and a message is displayed stating this fact. If the return value is not zero, an error occurred, and the file was not removed.

Renaming a File

The rename() function changes the name of an existing disk file. The function prototype, in STDIO.H, is as follows:

```
int rename( const char *oldname, const char *newname );
```

The filenames pointed to by oldname and newname follow the rules given earlier in this chapter. The only restriction is that both names must refer to the same disk drive; you can't rename a file to a different disk drive. The function rename() returns 0 on success, or -1 if an error occurs. Errors can be caused by the following conditions (among others):

* The file oldname does not exist.
* A file with the name newname already exists.
* You try to rename to another disk.

Listing 16.9 demonstrates the use of rename().

Listing 16.9. Using rename() to change the name of a disk file.

```
1:  /* Using rename() to change a filename. */
2:
3:  #include <stdio.h>
4:
5:  main()
6:  {
7:      char oldname[80], newname[80];
8:
9:      printf("Enter current filename: ");
10:     gets(oldname);
11:     printf("Enter new name for file: ");
12:     gets(newname);
13:
14:     if ( rename( oldname, newname ) == 0 )
15:         printf("%s has been renamed %s.\n", oldname, newname);
16:     else
17:         fprintf(stderr, "An error has occurred renaming %s.\n", oldname);
18:     return(0);
19: }
Enter current filename: list1609.c
Enter new name for file: rename.c
list1609.c has been renamed rename.c.
```

ANALYSIS: Listing 16.9 shows how powerful C can be. With only 18 lines of code, this program replaces an operating system command, and it's a much friendlier function. Line 9 prompts for the name of the file to be renamed. Line 11 prompts for the new filename. The call to the rename() function is wrapped in an if statement on line 14. The if statement checks to ensure that the renaming of the file was carried out correctly. If so, line 15 prints an affirmative message; otherwise, line 17 prints a message stating that there was an error.