10.1 INTRODUCTION

The standard library in C has many file I/O functions. The file-handling functions are easy to use, powerful, and complete. This chapter discusses the use of these functions to manipulate files.

In the chapter *Basic Input-Output,* functions to *read* from the keyboard and *write* to the screen were examined. The functions used to read and write a file are very similar, but there are several important differences between file I/O and console I/O (the term *console* here refers to the screen- keyboard pair). These differences are:

- The console always exists; a particular file may or may not exist.
- In case of a console, the program reads from the keyboard and writes onto the screen. In case of files, it is possible (and often required, say for an editor) to read from and write to the same file.

File I/O is always done in a program in the following sequence:

- Open the file
- Read or write to the file
- Close the file

Opening Files

Before performing any file I/O, the file must be opened. While opening the file, the following are specified:

- The name of the file
- The manner in which it should be opened (i.e., for reading, writing, both reading and writing, appending at the end of the file, overwriting the file, etc.)

The function `fopen` is used to open a file. It accepts two strings, the first is the *name* of the file, and the second is the *mode* in which it should be opened. For example, the following call to `fopen` specifies that the file `outfile1.txt` is to be opened for writing.

```
FILE *fp;
fp = fopen ("outfile1.txt", "w");
```

The first statement declares `fp` the pointer to a `FILE` structure. This structure is defined in `stdio.h`. The function `fopen` returns a pointer to the `FILE` structure which it creates. This pointer must be used in subsequent operations on the file, such as reading from or writing to it. The `FILE` pointer, `fp`, is also said to represent a *stream(or file descriptor).* If a file called `outfile1.txt` already exists, it is deleted and re-written.

Writing to Files

Let us write a message into the file opened with `fopen`. The following statements read an integer and write it to the file opened (i.e., `outfile1.txt`).

```
#include <stdio.h>
void main()
{
   int i;
   FILE* fp = fopen("outfile1.txt", "w");
   printf("Input an integer:  ");
   scanf("%d", &i);
   fprintf(fp, "%d\n", i);
   fclose(fp);
}
```

Assuming that 22 is entered in response to the `scanf` statement, the contents of the file `outfile1.txt` will be 22. The `fprintf` function writes the variable to the file. Just as `printf` writes to the screen, the function `fprintf` writes to a file. The only difference is that `fprintf` accepts an extra parameter, i.e., a pointer to the `FILE` structure returned by `fopen`. Having opened the file and written to it, we need to close the file. This is done by calling the function `fclose` with the pointer returned by `fopen`.

Recall that the function `puts` outputs a string, adding a newline character at the end and `putc` outputs a single character. Similarly, there exist functions `fputs` and `fputc` for writing strings and characters into a file. The functions `fputs` and `fputc` accept a pointer to the `FILE` structure as the first parameter, in addition to the string or character to be output. Consider the following example that uses `fputs` and `fputc`:

```
#include <stdio.h>
void main()
{
   FILE *fp = fopen("outfile2.txt", "w");
   fputs(fp, "string written to file");
   fputc(fp, 'c');
   fclose(fp);
}
```

Execution of this program will create the file `output2.txt`. Typing out the contents of the file (with the command `type outfile2.txt` in DOS or `cat outfile2.txt` in UNIX) will give,

```
string written to file
c
```

Note that `fputs` has written a newline following the string. The `fclose` function should be called only after all manipulations to the file are over. The pointer (`fp` in the above example) ceases to be valid after a call to `fclose`.

Reading from Files

The functions `scanf`, `gets` and `getc` are used to read from the standard input. To read from a file, the functions `fscanf`, `fgets` and `fgetc` are used. All these functions accept a `FILE` pointer (obtained from `fopen`) as their first parameter, in addition to the parameters required by the standard input functions. Following is an example of using the `fscanf` function. It reads the file `outfile1.txt` written in the earlier example and prints the integer value in it.

```
#include <stdio.h>
void main()
{
   int i;
```

```
    FILE*fp = fopen("outfile1.txt", "r");
    fscanf(fp, "%d", &i);
    printf ("The integer in outfile1.txt is %d", i);
    fclose (fp);
}
```

If `outfile1.txt` contains 22, as shown in the first example, the output of this program would be as follows,

```
The integer in outfile1.txt is 22
```

Predefined Streams

When a C program begins execution, the following data streams (pointers to `FILE` structures) will be automatically opened (these files are constants and not variables):

Name	Meaning
`stdin`	standard input device (opened for input)
`stdout`	standard output device (opened for output)
`stderr`	standard error output device (opened for output)
`stdaux`	standard auxiliary device (opened for both input and output)
`stdprn`	standard printer (opened for output)

The output of a function like `printf` goes to the stream `stdout`. A function like `scanf` receives its input from the stream `stdin`. We can write to any of the output streams by using functions like `fputc`, `fputs`, etc., and read from any of the input streams by using functions like `fgetc`, `fgets`, etc. These streams are considered to be files. Usually, `stdout` and `stderr` are connected to the screen, `stdin` is connected to the keyboard, `stdaux` is connected to the serial port `com1`, and `stdprn` is connected to the parallel port `lpt1`.

Thus, the statements,

```
int i;
scanf( "%d", &i );
fprintf(stdprn, "%d\n", i);
```

will send the output to the printer. If the printer is online, the number entered will appear on the paper. Note the use of `stdprn`, instead of the pointer to the `FILE` structure which is obtained through `fopen`. Since `stdprn` is already opened at the beginning of program execution, there is no need to open it again.

Closing Files

During a write to a file, the data written is not put on the disk immediately. It is stored in a buffer. When the buffer is full, all its contents are actually written to the disk. The process of emptying the buffer by writing its contents to disk is called *flushing the buffer*.

Closing the file flushes the buffer and releases the space taken by the `FILE` structure which is returned by `fopen`. Operating systems normally impose a limit on the number of files that can be opened by a process at a time. Hence, closing a file means that another can be opened in its place. Hence, if a particular `FILE` pointer is not required after a certain point in a program, pass it to `fclose` and close the file.

10.2 FILE STRUCTURE

I/O functions available are similar to their console counterparts; `scanf` becomes `fscanf`, `printf` becomes `fprintf`, etc. These functions read and write from file streams. As an example, the file stream structure `FILE` defined in the header file `stdio.h` in DOS is shown below. It is not a good practice to use the members of the `FILE` structure.

```
/* stream.c: Definition of the control structure for streams */
typedef struct
{
   int level;                          /* fill/empty level of buffer */
   unsigned  flags;                    /* File status flags          */
   char  fd;                           /* File descriptor(handle)    */
   unsigned char  hold;                /* Ungetc char if no buffer   */
   int  bsize;                         /* Buffer size                */
   unsigned char _FAR *buffer;         /* Data transfer buffer       */
   unsigned char _FAR *curp;           /* Current active pointer     */
   unsigned  istemp;                   /* Temporary file indicator   */
   short  token;                       /* Used for validity checking */
}  FILE;                               /* This is the FILE object    */
```

End of File

Here is a simple program, that reads names and marks of a number of students from the keyboard and stores them in a file called `STUREC`. The program uses the function `fprintf` to write to the file and `fscanf` to read from the file. When input from the file ends, all the file-reading functions return `EOF` (a constant defined in `stdio.h`). Hence, the reading loop has `fscanf` as part of the `while` loop condition, its return value being checked for `EOF`.

Note the use of the `FILE` pointer variable `fp`: after reading from the keyboard and writing to the `STUREC` file, `fp` is closed (which flushes the buffers). Then the file is reopened in read-only mode.

Example 10.1

```
/* fprintf.c: program to read and write to a file */
#include <stdio.h>

void main(void)
{
   FILE   *fp;
   char   name[20];
   unsigned int mark;

   /* File STUREC is opened in the 'append' mode  */
   fp = fopen ("STUREC", "a");

   printf ("Enter students' name and mark:\n");
   /* Ctrl+D  in UNIX and  Ctrl+Z in  DOS  */
   printf (" Use Ctrl+D to stop entry \n");

   while(( scanf("%s %u", name, &mark))!= EOF)
      /*Writing student names to file STUREC  */
      fprintf (fp, "%s %u", name, mark);

   fclose(fp);
   printf ("\n");
```

```
    /* Opening the file for read  */
    fp = fopen ("STUREC", "rt");
    printf("   Name       Mark\n");
    printf(" ---------------- \n");
    while ((fscanf (fp,"%s %u",name, &mark)) != EOF)
       /* Displaying file contents */
    printf ("%-10s %3u\n",name, mark);
    fclose(fp);
}
```

Run:

```
Enter students' name and mark:
Use Ctrl+D to stop entry
vivek      87
Rajkumar   98
Anand      67
babu       45
teja       90
^D
Name          Mark
----------------
vivek      87
Rajkumar   98
Anand      67
babu       45
teja       90
```

`FILE *fp` declares `fp` a pointer to the `FILE` stream structure. This structure is used to manipulate the file `STUREC`. Note the use of the mode string `"a"` to open the file in append mode (for writing data at the end), and `"r"` to open it for reading.

10.3 FILE-HANDLING FUNCTIONS

This section examines the file handling functions encountered so far in greater detail.

(i) `fopen`

Declaration: `FILE *fopen(const char *filename, const char *mode);`

`fopen` returns a pointer to the opened file stream. The parameter `filename` is the name of the file to be opened. When you want to specify the full pathname for the file, use double backslash or a single frontslash (as in UNIX) and not a single backslash. Single backslashes are interpreted as part of an escape sequence. Always use a single frontslash in UNIX.

For example, if the full pathname is: `c:\user\myfile.dat` then the string used in the `fopen` function should be: `c:\\user\\myfile.dat` or `c:/user/myfile.dat`.

The mode string used in calls to `fopen` can have one of the values shown in Table 10.1.

To specify that a given file is being opened or created in the text mode, append `t` to the mode string. For example: `rt`, `w+t`, `etc`. To specify the binary mode, append `b` to the string (`wb`, `a+b`, etc.). If neither `t` nor `b` have been specified, the file is opened in the text mode. Text and binary files are explained later in this chapter.

Return value

On success, it returns a pointer to the opened stream.

On error, it returns `NULL`.

String	Description
r	Open for reading only
w	Create for writing and if the file already exists, it will be overwritten
a	Append, open for writing at end of file(create for writing if it doesn't exist)
r+	Open an existing file for update (reading and writing)
w+	Create a new file for update. If it already exists, it will be overwritten
a+	Open for append; open for update at the end of the file

Table 10.1 Mode strings used in fopen

(ii) **fprintf**

Declaration: `int fprintf(FILE *stream, const char *format [, argument, ...]);`
This is the same as `printf` except that a pointer to a file stream must be specified. The function `fprintf` writes to the file associated with the file stream.

(iii) **fclose**

Declaration: `int fclose(FILE *stream);`
The buffers associated with the stream are flushed before being closed. System-allocated buffers are also released upon closing. Buffers assigned with `setbuf` or `setvbuf` are not automatically released (but if `setvbuf` has passed `NULL` for the buffer pointer, it will release the buffer upon closing). The functions `setbuf` and `setvbuf` are explained later. `stdout`, `stdin`, etc., can also be closed using `fclose`. Only the function `freopen` can reopen these streams.

Return value

On success, it returns `0`.
On error, it returns `EOF`.

(iv) **fscanf**

Declaration: `int fscanf (FILE *stream, const char *format [, address, ...]);`
The function `fscanf` reads from the file associated with the file stream. The arguments passed are similar to `scanf`, except that a pointer to the file stream must be specified.

In the above program, if `stdout` or `stdin` is used instead of `fp` in the `fprintf/fscanf` functions, the effect will be the same as using a `printf` or `scanf` function. In the previous program, `fprintf` and `fscanf` are used to perform the file I/O. These two functions are used to do formatted file I/O. The functions `fgetc` and `fputc` are better alternatives for manipulating the files, character by character.

The following program reads a string from the keyboard and saves it in a file (`str.txt`). This file is opened and the number of characters in it are counted. The program uses `fputc` and `fgetc` to do the required job.

Example 10.2

```
/* charcount.c: Create and count number of characters in a file */
#include <stdio.h>
void main(void)
{
```

```
    FILE *fp;
    int c, count = 0 ;
    printf ("Enter characters\n");
    /*  Ctrl+D in UNIX  and  Ctrl+Z  in DOS  */
    printf ("Press Ctrl+Z to stop entry. \n");
    fp = fopen ("str.txt", "w");
    while( (c = getchar()) != EOF )
       fputc (c,fp);
    fclose (fp);
    printf ("\n");
    fp = fopen ("str.txt", "r");
    while ( (c = fgetc(fp)) != EOF )
    {
       printf ("%c", (char)c);
       ++count;
    }
    fclose (fp);
    printf ("\nNumber of characters in the file: %d.\n", count);
}
```

Run:

```
Enter characters
Press Ctrl+Z to stop entry.
I saw the cat click the mouse.
^Z
Number of characters in the file: 32
```

Even though the number of characters in the string is 30, the size of the file is 32 bytes, since the newline and the end-of-file character (^Z) are also written into the file after the string.

(v) `fgetc`

Declaration: `int fgetc(FILE *stream);`

`fgetc` returns the next character in the named input stream. It is a function style of the macro `getc`.

Return value

On success, it returns the character read, after converting it to an `int` without sign extension.
On *end-of-file* or error, it returns `EOF`.

(vi) `fputc`

Declaration: `int fputc(int c, FILE *stream);`

The function `fputc` outputs the value of the character `c` to the specified stream.

Return value

On success, it returns the character `c`.
On error, it returns `EOF`.

`EOF` is a signed integer having the value -1. This integer is actually not present at the end-of-file, but is the value returned by the file I/O functions, when there is no character left in the file to be read. In the above program, `c` is an `int` and not a `char` variable. The reason being: `EOF` is an integer and to test `c` against `EOF`, `c` must also be an integer. The loop which writes to the file is:

```
while( (c = getchar()) != EOF )
   fputc (c, fp);
```

The above two lines are the compact representation of:

```
c = getchar();
while( c != EOF )
{
    fputc (c, fp);
    c = getchar();
}
```

Here, first `c` is assigned the return value of `getchar()` and then `c` is tested against `EOF` for inequality. Only, if the inequality is *true*, the function `fputc` is executed. The parentheses, enclosing the expression `c = getchar()`, are needed since `!=` has a higher precedence over them.

10.4 FILE TYPES

DOS treats files in two different ways viz., as binary or text files. Almost all UNIX systems do not make any distinction between the two. If a file is specified as the binary type, the file I/O functions do not interpret the contents of the file when they read from, or write to the file. But, if the file is specified as the text type, the file I/O functions interpret the contents of the file. The basic differences in these two types of files are:

- ASCII `0x1a` is considered an end-of-file character by the file I/O functions when reading from a text file and they assume that the end of file has been reached.
- In case of DOS, a newline is stored as the sequence `0x0d 0x0a` on the disk in case of text files (but in memory the newline is stored as `0x0a`). UNIX stores `\n` as `0x0a` both on disk and in memory.

The file handling functions used are shown in Table 10.2.

Writing	Reading	Used on
`fputc()`	`fgetc()`	Individual characters
`fputs()`	`fgets()`	Character strings
`fprintf()`	`fscanf`	Formatted ASCII
`fwrite()`	`fread()`	Binary files
`write()`	`read()`	Low level binary

Table 10.2

Now, we will see how to write to and retrieve records from files. For this purpose we will use binary files, and hence use the functions `fwrite` and `fread`. Note the use of the character `b` in the mode string used to open the file in binary mode.

Example 10.3

```
/* wret.c: program to write and retrieve records from a file */
#include <stdio.h>
typedef struct
{
    unsigned int reg_no;
    char name[50];
    unsigned int mark;
} st_rec;
```

```
void main(void)
{
   st_rec rec;
   FILE *fp;

   fp = fopen("class.rec", "wb");

   printf("\nEnter the register number, name and mark\n");
   printf("Type Ctrl+Z to stop\n");
   /* Read from keyboard and write to file */
   while(scanf("%u %s %u", &rec.reg_no, rec.name, &rec.mark)
                  != EOF)
   fwrite(&rec, sizeof(rec), 1, fp);

   fclose(fp);
   printf("\n");
   /* Read from file and write on screen */
   fp = fopen("class.rec", "rb");
   /* while loop terminates when fread returns 0 */
   while( fread(&rec, sizeof(rec), 1, fp ) )
   printf("%5u %-10s %3u\n", rec.reg_no, rec.name, rec.mark);
   fclose(fp);
}
```

Run:

```
Enter the register number, name and mark
Type Ctrl+Z to stop
4005 babu            85
4007 Rajkumar        98
4012 raju            77
4028 teja            93
4029 vivek           87
4005 babu            85
4007 Rajkumar        98
4012 raju            77
4028 teja            93
4029 vivek           87
```

(vii) **`fwrite`**

Declaration:

```
size_t fwrite(const void* ptr, size_t size, size_t n,
                    FILE* stream);
```

`fwrite` appends a specified number of equal-sized data items to an output file. Table 10.3 describes the various parameters. The total number of bytes written are `(n*size)`.

Argument	Definition/Meaning
`ptr`	Pointer: the data to be written
`size`	Length of each item of data
`n`	Number of data items to be appended
`stream`	Specifies output file

Table 10.3

Return value

On success, it returns the number of items (not bytes) actually written.
On error, it returns a short count (possibly 0).

(viii) **`fread`**

Declaration: `size_t fread(void *ptr, size_t size, size_t n, FILE *stream);`

`fread` reads a specified number of equal-sized data items from an input stream into a block. The parameters are described in Table 10.4. The total number of bytes read are (`n*size`).

Argument	Definition/Meaning
`ptr`	Pointer to a block into which data is read
`size`	Length of each item read, in bytes
`n`	Number of items read
`stream`	Points to input stream

Table 10.4

Return value

On success, it returns the number of items (not bytes) actually read.
On end-of-file or error, it returns 0.
The data type `size_t` is defined as :

```
typedef unsigned int size_t;
```

in `stdio.h`.

On execution of the above program, a file named `class.rec` is created. This file cannot be printed on the screen, since it is in the same format as the records in memory and not in the ASCII format. The function `fread` is used to get back the information into the record and `printf` is used for the formatted output. The file size, on the disk, will be a multiple of the record size. The record size in this case is:

54 bytes in DOS (2 bytes for `reg_no` + 50 bytes for `name` + 2 bytes for `mark`)
58 bytes in UNIX (4 bytes for `reg_no` + 50 bytes for `name` + 4 bytes for `mark`)

10.5 UNBUFFERED AND BUFFERED FILES

The characters written to a stream are immediately output to the file or device in *unbuffered* files. The characters written to a stream are *accumulated* and then written as a block to the file or device in *buffered* files. The disk is a slow device, hence file I/O is slower than memory access. Buffering, to some extent, speeds up file I/O. When buffering is used, the buffer used is a part of the main memory. The data from the disk is loaded into this buffer, on the first disk read. In subsequent reads, the data is obtained from the buffer, thus the data access time is decreased. `stdin` and `stdout` are unbuffered if they are *not* redirected. Otherwise, they are fully buffered.

Note: A common cause of error, is allocating the buffer as an a local variable and then fail to close the file before returning from the function where the buffer was declared.

Position Pointer

A *position pointer* is always associated with a file while reading or writing. Read and write operations are done at the position pointed to by this pointer. The earlier programs read the file sequentially. In order to perform *random access* while reading a file, the functions `fseek`, `ftell`, `fsetpos` and `fgetpos` are used. Function `rewind` can be used, as the name suggests, to reset the file position pointer to the beginning of the file.

The following program uses the file `class.rec` created by the previous program. The user is prompted for the record number that is to be retrieved. The function `fseek` is used to position the pointer to the desired record in the file.

Example 10.4

```
/* fseek.c: position the pointer in a file */
#include <stdio.h>
typedef struct
{
   unsigned int reg_no;
   char name[50];
   unsigned int mark;
} st_rec;
void main(void)
{
   int sz = sizeof(st_rec);
   unsigned long len, i, n;
   st_rec rec;
   FILE *fp;
   fp = fopen("class.rec", "rb");
   fseek(fp, 0, SEEK_END);
   len = ftell(fp);
   printf("\nThe file class.rec is of %li bytes in length.\n", len);
   n = len/sz;
   printf("There are %li records in the file.\n", n);
   do
   {
      printf("Enter the record number to read( 0 to stop):");
      fflush(stdin);
      scanf("%i", &i);
      printf("%li\n", i);
      if( i > 0 && i <= n)
      {
         fseek(fp, (i - 1) * sz, SEEK_SET);
         fread(&rec, sz, 1, fp );
         printf("%5u %-10s %3u\n", rec.reg_no, rec.name, rec.mark);
      }
   }while( i );
   fclose(fp);
}
```

Run:

```
The file class.rec is of 270 bytes in length.
There are 5 records in the file.
Enter the record number to read( 0 to stop): 3
 4012 raju        77
Enter the record number to read( 0 to stop): 1
 4005 babu        85
Enter the record number to read( 0 to stop): 5
 4029 vivek       87
Enter the record number to read( 0 to stop): 2
 4007 Rajkumar    98
Enter the record number to read( 0 to stop): 4
 4028 teja        93
Enter the record number to read( 0 to stop): 0
```

(i) `fseek`

Declaration: `int fseek(FILE *stream, long offset, int whence);`

The function `fseek` sets the file pointer associated with a stream to a new position. The parameters are described in Table 10.5 and 10.6. They are defined in `stdio.h`.

Argument	Description/Meaning
`stream`	Stream whose file pointer fseek sets
`offset`	Difference in bytes between `whence` (a file pointer position) and the new position for text mode streams offset should be 0 or a value returned by ftell.
`whence`	One of three SEEK_xxx file pointer locations

Table 10.5

Constant	Value	File location
SEEK_SET	0	Seeks from beginning of file
SEEK_CUR	1	Seeks from current position
SEEK_END	2	Seeks from end of file

Table 10.6

The function `fseek` discards any character pushed back using `ungetc`. It is used with stream I/O. After `fseek`, the next operation on an update file can be either input or output. It can return 0 (indicating that the pointer has been moved successfully), though it has actually failed. The reason being DOS, which actually resets the pointer, does not verify the setting.

Return value

On success (the pointer is successfully moved), it returns 0.
On failure, it returns a non-zero value. `fseek` returns an error code only on an unopened file or when a non-existing device is accessed.

(ii) **ftell**

Declaration: `long ftell(FILE *stream);`

`ftell` returns the current file pointer for `stream`. If the file is binary, the offset is measured in bytes from the beginning of the file. The value returned by `ftell` can be used in a subsequent call to `fseek`.

Return value

On success, the function `ftell` returns the current file pointer position.
On error, it returns `-1` and sets `errno` to a positive value.

(iii) **fflush**

Declaration: `int fflush(FILE *stream);`

If the given stream has a buffered output, `fflush` writes the output of the stream to the associated file. The stream remains open after `fflush` has been executed. `fflush` has no effect on an unbuffered stream.

Return value

On *success*, it returns `0`.
On *error*, it returns `EOF`.

In the previous program, just after the file is opened we are *seeking* the end of the file, i.e., positioning the pointer at the end of the file, the value returned by the function `ftell` gives the file length. The `scanf` function does not read the fields beyond what is specified in the format string, hence, `\n` and other characters, if any, remain unread. The next `scanf` will read these characters and cause an error. Using `fflush` will eliminate these errors.

(iv) **setvbuf**

Declaration: `int setvbuf(FILE *stream, char *buf, int type, size_t size);`

`setvbuf` causes the buffer `buf` to be used for I/O buffering. If `buf` is `NULL`, a buffer will be allocated using `malloc`. The `setvbuf` buffer will use `size` to indicate the amount of memory allocated and will be automatically released on closing the file. The parameters are described in Table 10.7.

Argument	Definition/Meaning
`stream`	Stream to which buffering is assigned
`buf`	Points to a buffer to be used for I/O buffering The buffer is used after stream has been opened
`type`	One of the buffer-types (_IOFBF, _IOLBF, or IONBF)
`size`	Specifies the setvbuf buffer size. 0 <size <32,767

Table 10.7

Buffer-type:
Constants for defining buffering styles which can be used with a file are described in Table 10.8.

Return value

On success, it returns 0.
On error, it returns non-zero (if an invalid value is given for `type` or `size`, or if there is not enough space to allocate a buffer)

Name	Meaning
_IOFBF	The file is fully buffered. When a buffer is empty, the next input operation will attempt to fill the entire buffer. On output, the buffer will be completely filled before any data is written to the file
_IOLBF	The file is line buffered. When a buffer is empty, the next input operation will still attempt to fill the entire buffer. On output, however, the buffer will be flushed whenever a newline character is written to the file
_IONBF	The file is unbuffered. The `buf` and `size` parameters are ignored. Each input operation will read directly from the file, and each output operation will immediately write the data to the file.

Table 10.8

10.6 ERROR HANDLING

In the above programs, the following assumptions are made:

- The file that is opened exists
- There is enough space on the disk for the file
- The functions `fseek`, `fread`, `fwrite`, etc., carry out the requested job successfully

These assumptions are not always true. In the above program, if the file `class.rec` does not exist, then some junk characters are output on the screen. Here, the `fopen` function would have returned unsuccessfully. We can find out whether the function was successful or unsuccessful, by inspecting its return value. The return value would be `NULL` in case of error. When a function returns unsuccessfully, *exception* processing must be performed.

The global variable `errno` is set to a specific value meant to designate the type of error, whenever an error occurs in a file I/O function. This value can be used to perform the required exception processing. The function `perror` can be used to print the nature of the error. The function `strerror` can be used to obtain a `char` pointer to the error string. The function `clearerr` can be used to reset the error indicator of a stream.

Example 10.5

The following program is the modified version of the previous example. This program takes care of all the possible error conditions that may occur. It also uses a global array `f_buf` to buffer the file. This buffer is set using `setvbuf`.

```
/* global1.c: header file has declaration of the global variable errno */
#include <errno.h>
#include <stdio.h>
#define BUFSIZE  (16384u)
typedef struct
{
   unsigned int reg_no;
   char name[50];
   unsigned int mark;
} st_rec;
char f_buf[BUFSIZE];
```

```
/* main returns an int value  */
int main(void)
{
   int sz = sizeof(st_rec);
   unsigned long len, i, n;
   st_rec rec;
   FILE *fp;
   if( !(fp = fopen("class.rec", "rb")) )
   {
      perror("Error opening class.rec");
      return errno;
   }
   /* No error can occur in this call to setvbuf because we are
      using a global array as the buffer.*/
  setvbuf(fp, f_buf, _IOFBF, BUFSIZE);
  if( fseek(fp, 0, SEEK_END) )
  {
     /*If last part of the file is on a bad sector then this error
       occurs*/
     perror("Error while seeking to the end of file");
     return errno;
  }
  if( (len = ftell(fp)) == -1 )
  {
     perror("Error finding the value of the file pointer");
     return errno;
  }
 printf("\nThe file class.rec is of %li bytes in length.\n", len);
 n = len/sz;
 printf("There are %li records in the file.\n", n);
 do
 {
    printf("Enter the record number to read( 0 to stop)\n");
    fflush(stdin);
    scanf("%I", &i);
    printf("%li\n", i);
    if( i > 0)
       if( fseek(fp, (i - 1) * sz, SEEK_SET) )
          printf("Seek failed,record number%i does not exist.\n", i);
   else
      {
         if( sz != fread(&rec, sz, 1, fp ) )
         {
            perror("Error reading the file class.rec");
            return errno;
         }
         printf("%5u %-10s %3u\n", rec.reg_no, rec.name, rec.mark);
      }
   }while( i );
```

```
    if( fclose(fp) )
    {
       perror("Error closing the file class.rec");
       return errno;
    }
    return 0;
} /* end main */
```

(i) **error**

Declaration: `extern int errno;`

Whenever an error occurs in a system call, `errno` is set to a value indicating the type of error.

(ii) **perror**

Declaration: `void perror(const char *s);`

`perror` prints on the `stderr` stream (normally the screen), the system error message corresponding to the last library routine that produced the error. It prints the argument `s`, a colon, the message corresponding to the current value of `errno`, and then a newline. The convention is to pass the file name of the program as the argument `s`.

Use `sys_errlist` to access the array of error message strings. You can use `errno` as an index into the array to find the string corresponding to the error number. None of the strings include a newline character. `sys_nerr` contains the number of entries in the array.

A typical sequence of statements for opening a file would be:

```
if( !( fp = fopen( "class.rec", "r" ) )
{
   perror( "Error opening class.rec" );
   return errno;
}
```

In this program, we have used the declaration:

```
int main(void)
```

Here, we are returning the error code to the operating system for further processing. The errors generally encountered in file manipulations are the following and is shown in Table 10.9.

Error	Reason
File not found	Wrong file name used (spelling mistake, error in the path specified) or File actually not present
Disk full	When writing to a disk with no more free sectors
Disk write protected	The disk has its write protect tag put on or some software has disabled writes to the disk
Unexpected `EOF`	File length is zero or file length is less than that expected
Sector not found	(i) Reading and writing to two different streams, one as read only and the other write only, but the file used for these streams is the same (ii) The disk head is not aligned on the physical sectors

Table 10.9

(iii) **clearerr**

Declaration: `void clearerr(FILE *stream);`

`clearerr` resets the named `streams` error and end-of-file indicators to `0`. Once the error indicator is set, stream operations continue to return the error status until a call is made to `clearerr` or `rewind`. The end-of-file indicator is reset with each input operation.

(iv) **ferror**

Declaration: `int ferror(FILE *stream);`

`ferror` is a macro that tests the given stream for a read or write error. If the streams error indicator has been set, it remains so until `clearerr` or `rewind` is called, or until the `stream` is closed.

Return value

`ferror` returns a non-zero if an error is detected on the named `stream`.

10.7 MORE ON FILE HANDLING FUNCTIONS

(i) **fcloseall**

Declaration: `int fcloseall(void);`

`fcloseall` closes all open streams except `stdin`, `stdout`, `stdprn`, `stderr`, and `stdaux`.

Return value

On success, it returns the total number of streams it closed.
On error, it returns EOF.

(ii) **fdopen and freopen**

Declarations: `FILE *fdopen(int handle, char *type);`

`FILE *freopen(const char *filename, const char *mode, FILE *stream);`

`fdopen` associates a stream with a file handle obtained from `creat`, `dup`, `dup2`, or `open`. These functions are examined in the next section.

`freopen` substitutes a file for the open stream and closes the stream, regardless of the success of the open stream. `freopen` is useful for changing the file attached to `stdin`, `stdout` or `stderr`. Parameters are described in Table 10.10.

Argument	Function	Description/Meaning
`handle`	`fdopen`	File handle from creat, dup, dup2, or open
`type`	`fdopen`	Type of the stream; must match the mode of the open handle
`filename`	`freopen`	File substituted in place of the open stream
`mode`	`freopen`	Mode string defines the mode of the opened file (r, w+, a+, etc.)
`stream`	`freopen`	Open stream, substituted by filename

Table 10.10

When a file is opened for update, both input and output can be done on the resulting stream. However, note that:

- An output cannot be directly followed by an input without an intervening `fseek` or `rewind`
- An input cannot be directly followed by an output without an intervening `fseek` or `rewind`, or an input that encounters end-of-file

Return value

On success, `fdopen` returns a pointer to the newly opened stream and `freopen` returns the argument stream.

On error, these functions return `NULL`.

(iii) **`feof`**

Declaration: `int feof(FILE *stream);`

`feof` is a macro that checks the given stream for an end-of-file indicator. Once the indicator is set, read operations on the file returns the indicator until `rewind` is called, or the file is closed. The end-of-file indicator is reset with each input operation.

Return value

Returns a non-zero if an *end-of-file* indicator is detected upon the last input operation on the named stream.

Returns `0` if *end-of-file* is not reached.

(iv) **`remove`**

Declaration: `int remove(const char *filename);`

`remove` deletes the file specified by `filename`. It is a macro that simply translates as a call to `unlink`. If your file is open, be sure to close it before removing it. The string `*filename` may include a full path.

Return value

On success, `remove` returns 0.

On error, it returns `-1`, and sets `errno` to one of the following:

ENOENT	no such file of directory
EACCES	permission denied

(v) **`rename`**

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

`rename` changes the name of a file from `oldname` to `newname`. If a drive specifier is given in `newname`, it must be the same as that given in `oldname`. Directories in `oldname` and `newname` need not be the same, hence, `rename` can be used to move a file from one directory to another. Wildcards are not allowed.

Return value

On success, it returns 0.

On error, it returns -1 and sets `errno` to one of the following:

ENOENT	no such file or directory
EACCES	permission denied
ENOTSAM	not same device

(vi) **`rewind`**

Declaration: **`void rewind(FILE *stream);`**

`rewind(stream)` is equivalent to `fseek(stream, 0L, SEEK_SET)`, except that `rewind` clears the end-of-file and error indicators, while `fseek` clears only the *end-of-file* indicator. After rewind, the next operation on an update file can be either an input or output.

(vii) **tempnam**

Declaration: `char *tempnam(char *dir, char *prefix);`

`tempnam` creates a unique file name in arbitrary directories. It attempts to use the following directories, in the order shown, while creating the file:

a) the directory specified by the `TMP` (`TEMP` in DOS) environment variable
b) the `dir` argument passed
c) the `P_tmpdir` definition in `stdio.h`: if you edit `stdio.h` and change this definition, `tempnam` will not use the new definition
d) the current working directory

`dir`: specifies the directory where the file will be created.

`prefix`: Specifies the first part of the file name; it cannot be longer than 5 characters, and should not contain a period. If any of these directories is `NULL`, undefined, or does not exist; it is skipped. `tempnam` creates an unique file name by concatenating the directory name, the prefix, and 6 unique characters. Space for the resulting file name is allocated by `malloc`; the caller should call `free` to *release* this file name when no longer needed. The unique file is not actually created; `tempnam` only verifies that it does not currently exist.

Return value

On success, it returns a pointer to the temporary file name.
Returns `NULL` if it cannot create a unique file name.

(viii) **tmpfile**

Declaration: `FILE *tmpfile(void);`

`tmpfile` creates a temporary binary file and opens it for update (i.e., mode `w+b`). The file is automatically removed when closed or when the program is terminated.

Return value

On success, `tmpfile` returns a pointer to the stream of the temporary file created.
On *error* (if the file cannot be created), `tmpfile` returns `NULL`.

(ix) **tmpnam**

Declaration: `char *tmpnam(char *sptr);`

`tmpnam` creates a unique file name, which can be safely used as the name of a temporary file. `tmpnam` generates a different string each time you call it, for a maximum of `TMP_MAX` times. `sptr` is either `NULL` or a pointer to an array of at least `L_tmpnam` characters. If `sptr` is `NULL`, `tmpnam` leaves the temporary file name in an internal static object and returns a pointer to it. If `sptr` is not `NULL`, `tmpnam` places the temporary file name in the array `*sptr` and returns `sptr`.

Return value

If `sptr` is `NULL`, it returns a pointer to an internal static object.
If `sptr` is not `NULL`, returns `sptr`.

(x) **rmtmp**

Declaration: `int rmtmp(void);`

`rmtmp` closes and deletes all open temporary file streams, which were previously created with `tmpfile`. The current directory must be the same as it was when the files are created, else the files will not be deleted.

Return value

Returns the total number of temporary files closed and deleted.

(xi) **unlink**

Declaration: `int unlink(const char *filename);`

`unlink` deletes a file specified by filename. In DOS any drive, path, and file name can be used as a filename. Wildcards are not allowed. Read-only files cannot be deleted by this call. To remove read-only files, first use `chmod` to change the read-only attribute. If the file is open, be sure to close it before unlinking it.

Return value

On success, `unlink` returns 0.

On error, it returns `-1` and sets `errno` to one of the following:

ENOENT	path or file name not found
EACCES	permission denied

This is not an exhaustive listing of the available functions. Depending on the compiler used, there may be more functions defined in `stdio.h`. But it is unlikely that you will using any functions other than the ones mentioned above.

10.8 LOW LEVEL FILE I\O

When the functions `fread`, `fwrite`, etc., are used, data transfer occurs first to the buffers and then to the disk. In low level file functions, the data is not buffered but directly written to the file. The low level file functions use file handles. As mentioned earlier, every stream has a handle, hence, the functions that directly act on handles are faster than the analogous file functions, in `stdio.h`, which require streams.

`open`, `read`, `write`, etc., are the unbuffered counterparts of `fopen`, `fread`, `fwrite`, etc.

From the following program, we can see that `fwrite` is buffered and `write` is not buffered.

```
/* fwrite.c: program to illustrate buffered write */
#include <io.h>
#include <stdio.h>
void main()
{
   FILE *fp;
   fp = fopen("testbuf.txt", "wt");
   fwrite("1. This is fwrite\n", 1, 18, fp);
   write(fileno(fp), "2. This is write\n", 17);
   fclose(fp);
}
```

Both the functions `fwrite` and `write`, write to the file `testbuf.txt`. After the execution of this program, the file `testbuf.txt` will contain:

```
2. This is write.
1. This is fwrite.
```

As can be seen, the string written using `write` is written first into the file. This is because, `write` is not buffered and hence, all writes from the function `write` go to the file immediately. The data passed to `fwrite` will be written to the file only when `fclose` is called (or when its buffer overflows).

10.9 REDIRECTION AND PIPING

Redirection is used to substitute files for standard input(keyboard) and standard output(screen).

Output Redirection

The output of `printf` and similar functions go to the screen; this output can be redirected to files by using the output redirection operator: > (at the command line).

```
myprog  >  file.dat
```

`myprog` is an executable program. The output from `myprog` goes to the file `file.dat`. The functions such as `printf, puts`, etc., output to the standard output; with *redirection*, the file is treated as a standard output. When > is used for redirection, the output file will be created, and if it already exists, the previous contents will be lost. To append to the file, we should use >> as the redirection operator.

Input Redirection

Functions like `scanf, gets`, etc., accept input from the standard input, which by default is the keyboard. With input *redirection*, the standard input is a file.

The input redirection operator is an <.

```
myprog < file.dat
```

In this example, the contents of the file `file.dat` serve as an input to the program `myprog`.

Piping

The process of feeding the output of one program to the input of another program, so that both the programs run simultaneously, is called piping. The piping operator is | (at the command line).

```
myprog | otherprog
```

In this example, the standard output of the program `myprog` is piped to the standard input of `otherprog`. The following command uses *redirection* and *piping*.

```
prog1 < file1.dat | prog2 > file2.dat
```

Here, the contents of `file1.dat` is input to the program `prog1`, its output is fed to program `prog2`, and the output of the program `prog2` is saved in `file2.dat`.

Note: The redirection and piping operations are performed by the operating system. `stderr` cannot be redirected in the above described manner. Redirection and piping can also be done in the C program by using functions like `dup, dup2`, and `pipe`.

The following program will run another executable program, and any output to the printer from the executable program is saved in a file, i.e., it is an implementation of printer to file redirection.

Example 10.6

```
/* prtfile.c: printer to file redirection */
#include <io.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
int main( int argc, char *argv[])
{
   char filnam[] = "OUTFILE.PRN";
   int old_prn_han, new_prn_han;
```

```
    if(argc < 2)
    {
       printf("Usage:%s Executable_prog[Output_PRN_filename]\n",argv[0]);
       return 1;
    }
    /* Get the handle of the stream stdprn and duplicate it. */
    old_prn_han = dup(fileno(stdprn));
    /* If command line argument present use the specified filename */
    if(argc > 2)
       strcpy(filnam, argv[2]);
    /* Open the file for write as binary  */
    if( (new_prn_han = open(filnam, O_CREAT | O_BINARY)) == -1)
    {
       perror(filnam);
       return 1;
    }
    /* Make the handle of stdprn same as that of the opened file */
    /* Now any write into stdprn(i.e., printer) will be redirected to
    the file */
    dup2(new_prn_han, fileno(stdprn));
    /* Execute the specified program  */
    if(system(argv[1]))
    {
       perror(argv[1]);
       return 1;
    }
    close(fileno(stdprn));
    /* Restore the original stdprn handle */
    dup2(old_prn_han, fileno(stdprn));
    return 0;
}
```

The header file `fcntl.h` defines *open flags* for `open` and other unbuffered library functions. In the above program, if no executable file name is specified (i.e., if `argc` is equal to 1), then the usage of the program is printed, and the program terminates with an error code 1, the statement,

```
old_prn_han = dup(fileno(stdprn));
```

will save the original file handle of the stream `stdprn` in the variable `old_prn_han`. If a program name is specified on the command line, then the file is opened and used as the redirected printer file, else, the default name `OUTFILE.PRN` is used. The file is created in the binary mode using `open`. This handle is now associated with the stream `stdprn`. The statement,

```
dup2(new_prn_han, fileno(stdprn));
```

will attach the stream `stdprn` to the new file opened. Any write to the stream `stdprn` will result in the data going to the file. Thus, the printer stream is redirected to a file. The function `system` will execute the program specified by the user. Due to redirection, any write to the printer from this program will be saved in the file. After the program terminates, the printer handle is reset to its old (original) value and the opened file is closed.

The above program demonstrates:

- The use of `io.h` functions for file I/O
- The use of `dup` and `dup2` for redirection
- A method to access the command line arguments

Other Functions in `io.h`

(i) **dup**

Declaration: `int dup(int handle);`

`dup` duplicates the file handle passed, and creates a new file handle.

(ii) **dup2**

Declaration: `int dup2(int oldhandle, int newhandle);`

`dup2` duplicates a file handle onto an existing file handle, and creates a new handle with the value `newhandle`. The integers `newhandle` and `oldhandle` are file handles obtained from calls to `creat`, `open`, `dup`, or `dup2`. They can also be used with `fopen` which returns the file handle with a `FILE` structure

The new file handle has the following in common with the original file handle:

- The same open file or device
- The same file pointer (changing the file pointer of one changes the other)
- The same access modes: read, write, read/write

Subsequent to the `dup2` call, writing to the file descriptor `newhandle` will write to the file associated with the `oldhandle`, i.e., `newhandle` is redirected to `oldhandle`.

Return value

On success, `dup` returns the new file handle, a non-negative integer; and `dup2` returns 0.
On error, both the functions return -1 and set `errno` to one of the following:

`EMFILE`	too many open files
`EBADF`	bad file number

(iii) **fileno**

Declaration: `int fileno(FILE *stream);`

`fileno` is a macro that returns the file handle for the stream. It is defined in `stdio.h` as:

```
#define fileno(f)        ((f)->fd)
```

If stream has more than one handle, `fileno` returns the handle assigned to the stream when it was first opened.

Return value

`fileno` returns the integer file handle associated with the stream.

(iv) **open and sopen**

Declarations: `int open(const char *path, int access[, unsigned mode]);`

`int sopen(path, access, shflag, mode);`

`open` and `sopen` open the file specified by `path`, and then prepare it for reading and/or writing. `sopen` prepares the `path` for shared reading and/or writing. The parameters used by these functions are described in Table 10.11.

`sopen` is a macro defined as:

```
open(path, (access) | (shflag), mode);
```

On successfully opening the file, both routines set the file pointer (which marks the current position in the file) to the beginning of the file. The maximum number of files opened simultaneously is defined by HANDLE_MAX.

Argument	Description/Meaning
path	File that the function opens and prepares for reading/writing
access	Specifies the file-access mode in which the path has to be opened. File-access symbolic constants are defined in fcntl.h
mode	Specifies the mode in which the newly created path opens (used only if access includes O_CREATE)
shflag	Specifies the type of file-sharing allowed on path. File-sharing symbolic constants are defined in share.h

Table 10.11

Return value

On success, both return the file handle (a non-negative integer) and set the file pointer to the beginning of the file.

On error, both return -1 and set errno to one of the following:

ENOENT	no such file or directory
EMFILE	too many open files
EACCES	permission denied
EINVACC	invalid access code

(v) **close**

Declaration: `int close(int handle);`

This function closes the file associated with handle. It is the low level counterpart of the fclose function. The function close does not write a Ctrl-Z character at the end of the file. The file handle, handle is obtained from a call to creat, creatnew, creattemp, dup, dup2, open.

Return value

On success, this function returns 0.

On error (handle is not the handle of a valid, open file), close returns -1 and sets errno to EBADF (Bad file number).

(vi) **write**

Declaration: `int write(int handle, void *buf, unsigned len);`

write writes data to a file or device. Parameters are described in Table 10.12.

Argument	Description/Meaning
handle	File handle obtained from a create, open, dup, or dup2 call
buf	Points to a data area that the function wrote the bytes from
len	Number of bytes the function attempts to write

Table 10.12

Except when you use `write` to write to a text file, the number of bytes written to the file will not be more than the number requested. If the number of bytes actually written is less than that requested, it is probably an error (most likely a full disk). The maximum number of bytes that `write` can write at a time is 65534 on a 16-bit compiler because, 65,535 (`0xFFFF`) is the same as `-1`, the error return indicator.

Carriage Returns and Line Feeds

On text files, when `write` is used to output an `LF` (linefeed) character, it outputs a `CR/LF` (carriage-return/linefeed) pair. The written value from a write to a text file does not count, generated carriage returns.

Disk Files and Devices

With disks or disk files, writing always proceeds from the current file pointer. With devices, these functions send bytes directly to the device. For files opened with the `O_APPEND` option, `write` positions the file pointer to `EOF` before writing the data.

Return value

On success, `write` returns the number of bytes written.
On error, `write` returns `-1` and sets `errno` to `EACCES` or `EBADF`

10.10 COMMAND LINE ARGUMENTS IN DOS AND UNIX

The prototype of the function `main` when command line parameters are required, is as follows:

```
void main(int argc, char*argv[])
```

`argc` is the counter of the number of arguments on the command line. `argv` is a pointer to an array of character strings. By convention, `argv[0]` is the name by which the program is invoked, hence, `argc` is always atleast `1` and `argv[argc]` is the `NULL` pointer.

Example 10.7

If a program, whose name is `prog, is` called as,

```
prog data.in data.out
```

from the command line, then the variable `argc` will be equal to 3 (2 arguments + 1 program name) and `argv[0] = "prog", argv[1] = "data.in", argv[2] = "data.out"`, and `argv[3] = NULL` pointer.

Thus, we can use these two variables to access the command line variables. In DOS, the command line arguments are passed directly to the program(with no modification), and the program has to process the filename with its wildcards, but in UNIX, the wildcards are expanded by the shell and then all filenames that match the wildcard are passed to the program.

Example 10.8

```
prog *.txt
```

In DOS, `argc = 2, argv[1] = "*.txt", argv[2] = NULL` pointer.

In UNIX, `argc = 1` + the number of files that match `*.txt`, and `argv` points to the array of strings having all the filenames that match `*.txt`. If `a.txt, b.txt` and `c.txt` are the only matching filenames, then:

```
argc = 4, argv[0] = "prog", argv[1] = "a.txt", argv[2] = "b.txt", argv[3]
= "c.txt", and argv[4] = NULL.
```

10.11 DIRECTORY FUNCTIONS

Example 10.9

The following program displays the contents of the directory specified on the command line. We can also include wildcards in the name. This program is specific to DOS.

```
/* dir.c: program to list files and directories on command line*/
#include <dir.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
   struct ffblk ffb;
   int done, lines = 1;
   /* If argument absent, print usage. */
   if( argc == 1)
   {
      printf("Usage:\n\t%s <directory_name>|<pathname>\n");
      printf("The pathname can contain wildcards.");
      return 1;
   }
   /* Argument found, hence give the listing of the directory */
   printf("Directory listing of %s\n", argv[1]);
   done = findfirst(argv[1], &ffb, 0);
   while (!done)
   {
      /* 'lines' keeps a count of the number of lines displayed */
      /* It is initialized to 1 as we have already printed one line
         outside this loop*/
      lines++;
      printf("%s\n", ffb.ff_name);
      done = findnext(&ffb);
      if( lines > 22 )
      {
         /*  Screen full, hence wait for key press.  */
         printf("\nPress Enter to continue . . .");
         getchar();
         fflush(stdin);
         /*  Reset 'lines' to 0(Note: not to 1)  */
         lines = 0;
         printf("\n");
      }
   }
   return 0;
}
```

The structure `ffblk` is the DOS file control block structure. It is given below.

```
struct ffblk
{
   char ff_reserved[21]; /* reserved by DOS */
   char ff_attrib;       /* attribute found */
   int  ff_ftime;        /* file time */
```

```
    int  ff_fdate;        /* file date */
    long ff_fsize;        /* file size */
    char ff_name[13];     /* found file name */
};
```

`ff_ftime` and `ff_fdate` are 16-bit structures divided into bit fields to refer to the current date and time. The structures of these fields are established by DOS.

(i) **`findfirst`**

Declaration: `int findfirst(const char *pathname, struct ffblk *ffblk, int attrib);`

`findfirst` finds the first matches and fills up the `ffblk` structure. This structure has to be passed in subsequent calls to `findnext`.

(ii) **`findnext`**

Declaration: `int findnext(struct ffblk *ffblk);`

`findnext` finds subsequent files that match the pathname argument of `findfirst`.
`pathname` is a pathname string with an optional drive specifier; it is the file name to be found. The file name can contain wildcard characters (? and *).
`ffblk` points to the `ffblk` structure that is filled with the file-directory information if `findfirst` or `findnext` finds a file that matches `pathname`.
`attrib` is a DOS file-attribute byte that `findfirst` uses to select eligible files for the search . For each call to `findnext` or `findnext`, one file name will be returned until no more files are found in the directory specified in `pathname`.

Return value

On success, (a match is found) these functions return 0.
On error (no more files can be found, or there is some error in the file name), `findfirst` and `findnext` return `-1` and set `errno` to either `ENOENT` or `EMFILE` (no more files).

10.12 DOS AND BIOS FILE/DISK I\O

`dos.h` and `bios.h` header files are generally provided with C compilers for DOS. These header files contain prototypes of the file/disk I/O functions that are specific to DOS. Using these functions, we can access the disk at the sector level. We can access the file allocation table, directory entries, etc., of the disk. These functions are specific to DOS and are equivalent to the services provided by `INT 21H`, `INT 25H`, `INT 26H` (DOS interrupts) and `INT 13H` (BIOS interrupt). The functions provided in this library vary from compiler to compiler.

10.13 PROGRAMMING EXAMPLES

Example 10.10

The following program accepts three filenames on the command line. The contents of the first file followed by those of the second are put in the third file. Note that in `main`, the first two files are opened in the read mode, and the third one in the write mode. If a file whose name is same as the third one already exists, it is overwritten.

```
/* merg.c: program to merge two files into a third file */
#include <stdio.h>
#include <stdlib.h>
FILE *open_file( const char *filename, const char *mode )
```

```
{
   FILE *stream = fopen( filename, mode );
   if( !stream )
   {
      printf( "Cant open %s: %s\n", filename, strerror( errno ) );
      exit( 1 );
   }
   return stream;
}
/* The merge function puts the contents of f1 followed by those of f2
into the file represented by f3.*/
void merge( FILE *f1, FILE *f2, FILE *f3 )
{
   int input_char;
   /* First copy f1 to f3.*/
   while( (input_char = fgetc( f1 )) != EOF )
      fputc( input_char, f3 );
   /*Next copy f2 to f3.*/
   while( (input_char = fgetc( f2 )) != EOF )
      fputc( input_char, f3 );
}
void main( int argc, char *argv[] )
{
   FILE *f1, *f2, *f3;
   if( argc != 4 )
   {
      printf( "\nUsage:\n" );
      printf( "%s <source1> <source2> <destination>\n\n", argv[ 0 ] );
      exit( 1 );
   }
   f1 = open_file( argv[ 1 ], "r" );
   f2 = open_file( argv[ 2 ], "r" );
   f3 = open_file( argv[ 3 ], "w" );
   merge( f1, f2, f3 );
   fclose( f1 );
   fclose( f2 );
   fclose( f3 );
}
```

Run:

Assume that the file `one.txt` **has contents,**

```
To iterate is human,
to recurse divine.
```

and `two.txt` **has:**

```
So says Peter Deutch.
```

The above file is compiled and linked into an executable code called `merge.exe`**: the command**

```
merge one.txt two.txt three.txt
```

will result in the file `three.txt` **being created with the contents**

```
To iterate is human,
to recurse    divine.
So says Peter Deutch.
```

Example 10.11

The following program accepts two filenames as command line parameters and appends the second file to the first. Note that in `main`, the first file is opened in the append mode, while the second one in read mode.

```
/* concat.c:  program to concatenate two files */
#include <stdio.h>
#include <stdlib.h>
FILE *open_file( const char *filename, const char *mode )
{
   FILE *stream = fopen( filename, mode );
   if( !stream )
   {
      printf( "Cant open %s: %s\n", filename, strerror( errno ) );
      exit( 1 );
   }
   return stream;
}
/* Stores contents of f1 followed by f2 into file represented by f3.*/
void concat( FILE *f1, FILE *f2 )
{
   int input_char;
   while( (input_char = fgetc( f2 )) != EOF ) /*First copy f1 to f3.*/
      fputc( input_char, f1 );
}
void main( int argc, char *argv[] )
{
   FILE *f1, *f2;
   if( argc != 3 )
   {
      printf( "\nUsage:\n" );
      printf( "%s <source1> <source2>\n\n", argv[ 0 ] );
      exit( 1 );
   }
   f1 = open_file( argv[ 1 ], "a" );
   f2 = open_file( argv[ 2 ], "r" );
   concat( f1, f2 );
   fclose( f1 );
   fclose( f2 );
}
```

Run:

If the contents of the files `one.txt` and `two.txt` are specified as in the previous example, and the above file is compiled and linked into an executable code called `concat.exe`, the command

```
concat one.txt two.txt
```

will modify the file `one.txt` to read:

```
To iterate is human,
to recurse   divine.
So says Peter Deutch.
```