CHAPTER 15

Files

LEARNING OBJECTIVE

In this chapter, we will learn about files which are very important for storing information permanently. The stored information can then be processed using programs written in the C language.

15.1 INTRODUCTION TO FILES

A *file* is a collection of data stored on a secondary storage device such as a hard disk. Till now, we had been processing data that was entered through the computer's keyboard. However, this task can become very tedious especially when there is a huge amount of data to be processed. A better solution, therefore, is to combine all the input data into a file and then design a C program to read this data from the file whenever required.

Broadly speaking, a file is basically used because real-life applications involve large amounts of data and in such applications the console-oriented I/O operations pose two major problems:

- First, it becomes cumbersome and time-consuming to handle huge amounts of data through terminals.
- Second, when performing I/O operations using terminals, the entire data is lost when either the program is terminated or the computer is turned off. Therefore, it becomes necessary to store data on a permanent storage device (the disks) and read it whenever necessary, without destroying the data.

In order to use files, we have to learn file input and output operations, i.e., how data is read or written to a file. Although file I/O operations are almost same as terminal I/O, the only difference is that when doing file I/O, the user must specify the name of the file from which data should be read/written.

15.1.1 Streams in C

In C, the standard streams are termed as pre-connected input and output channels between a text terminal and the program (when it begins execution). Therefore, a stream is a logical interface to the devices that are connected to the computer.

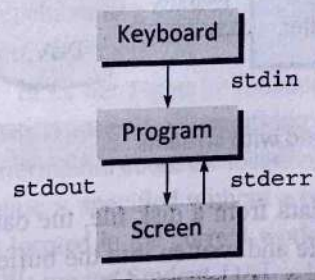


Figure 15.1 Standard streams

Streams are widely used as a logical interface to a file where a file can refer to a disk file, the computer screen, keyboard, etc. Although files may differ in the form and capabilities, all streams are the same. The three standard streams (Figure 15.1) in the C language are as follows:

- standard input (`stdin`)
- standard output (`stdout`)
- standard error (`stderr`)

Standard input (stdin) Standard input is the stream from which the program receives its data. The program requests transfer of data using the `read` operation. However, not all programs require input. Generally, unless redirected, input for a program is expected from the keyboard.

Standard output (stdout) Standard output is the stream where a program writes its output data. The program requests data transfer using the `write` operation. However, not all programs generate output.

Standard error (stderr) Standard error is basically an output stream used by programs to report error messages or diagnostics. It is a stream independent of standard output and can be redirected separately. No doubt, the standard output and standard error can also be directed to the same destination.

A stream is linked to a file using an `open` operation and disassociated from a file using a `close` operation.

15.1.2 Buffer Associated with File Stream

When a stream linked to a disk file is created, a buffer is automatically created and associated with the stream. A buffer is nothing but a block of memory that is used for temporary storage of data that has to be read from or written to a file.

Buffers are needed because disk drives are block-oriented devices as they can operate efficiently when data has to be read/written in blocks of certain size. An ideal buffer size is hardware-dependent.

The buffer acts as an interface between the stream (which is character-oriented) and the disk hardware (which is block-oriented). When the program has to write data to the stream, it is saved in the buffer till it is full. Then the entire contents of the buffer are written to the disk as a block. This is shown in Figure 15.2.

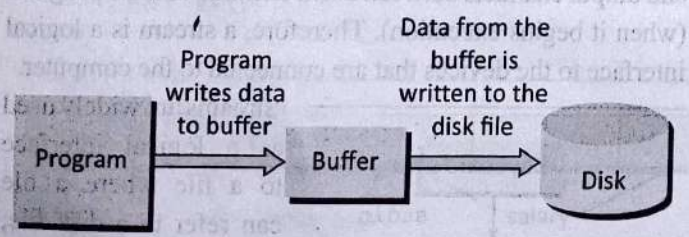


Figure 15.2 Buffers associated with streams

Similarly, when reading data from a disk file, the data is read as a block from the file and written into the buffer. The program reads data from the buffer. The creation and operation of the buffer is automatically handled by the operating system. However, C provides some functions for buffer manipulation. The data resides in the buffer until the buffer is flushed or written to a file.

15.1.3 Types of Files

In C, the types of files used can be broadly classified into two categories—ASCII text files and binary files.

ASCII Text files

A *text file* is a stream of characters that can be sequentially processed by a computer in forward direction. For this reason, a text file is usually opened for only one kind of operation (reading, writing, or appending) at any given time. Because text files only process characters, they can only read or write data one character at a time. In C, a text stream is treated as a special kind of file.

Depending on the requirements of the operating system and on the operation that has to be performed (read/write operation) on the file, newline characters may be converted to or from carriage return/line feed combinations. Besides this, other character conversions may also be done to satisfy the storage requirements of the operating system. However, these conversions occur transparently to process a text file.

In a text file, each line contains zero or more characters and ends with one or more characters that specify the end of line. Each line in a text file can have maximum of 255 characters. A line in a text file is not a C string, so it is not terminated by a null character. When data is written to a text file, each newline character is converted to a carriage return/line feed character. Similarly, when data is read from a text file, each carriage return/line feed character is converted into newline character.

Another important thing is that when a text file is used, there are actually two representations of data—internal or external. For example, an `int` value will be represented as 2 or 4 bytes of memory internally, but externally this value will be represented as a string of characters representing its decimal or hexadecimal value. To convert internal representation into external, we can use the `printf` and `fprintf` functions. Similarly, to convert an external representation into internal, `scanf` and `fscanf` can be used. We will read more about three functions in the coming sections.

Programming Tip:
The contents of a binary file are not human-readable. If you want the data stored in the file to be read, then store the data in a text file.

Note: In a text file, each line of data ends with a newline character. Each file ends with a special character called the end-of-file (EOF) marker.

Binary files

A binary file may contain any type of data, encoded in binary form for computer storage and processing purposes.

Like a text file, a binary file is a collection of bytes. In C, a byte and a character are equivalent. Therefore, a binary file is also referred to as a character stream with the following two essential differences:

- A binary file does not require any special processing of the data and each byte of data is transferred to or from the disk unprocessed.
- C places no constructs on the file, and it may be read from, or written to, in any manner the programmer wants.

While text files can be processed sequentially, binary files, on the other hand, can be either processed sequentially or randomly depending on the needs of the application. In C, to process a file randomly, the programmer must move the current file position to an appropriate place in the file before reading or writing data. For example, if a file is used to store records (using structures) of students, then to update a particular record, the programmer must first locate the appropriate record, read the record into memory, update it, and finally write the record back to the disk at its appropriate location in the file.

Note Binary files store data in the internal representation format. Therefore, an `int` value will be stored in binary form as a 2-or 4-byte value. The same format is used to store data in memory as well as in file. Similar to text file, a binary file also ends with an EOF marker.

In a text file, an integer value 123 will be stored as a sequence of three characters—1, 2, and 3. So, each character will take 1 byte and therefore, to store the integer value 123, we need 3 bytes. However, in a binary file, 123 will be stored in 2 bytes in the binary form. This clearly indicates that binary files take less space to store the same piece of data and eliminates conversion between internal and external representations and are thus more efficient than the text files.

15.2 USING FILES IN C

To use files in C, we must use the following steps:

- declare a file pointer variable
- open the file
- process the file
- close the file

In this section, we will discuss these steps in detail.

15.2.1 Declaring a File Pointer Variable

There can be a number of files on the disk. In order to access a particular file, you must specify the name of the file that has to be used. This is accomplished by using a file pointer variable that points to a structure `FILE` (defined in `stdio.h`). The file pointer will then be used in all subsequent operations in the file. The syntax for declaring a file pointer is,

Programming Tip: An error will be generated if you use the filename to access a file rather than the file pointer.

```
FILE *file_pointer_name;
```

For example, if we write

```
FILE *fp;
```

then, `fp` is declared as a file pointer.

15.2.2 Opening a File

A file must first be opened before data can be read from or written to it. In order to open a file and associate it with a stream, the `fopen()` function is used. The prototype of `fopen()` can be given as,

```
FILE *fopen(const char *file_name, const char
   *mode);
```

Using this above prototype, the file whose pathname is the string pointed to by `file_name` is opened in the mode specified using the mode. If successful, `fopen()` returns a pointer-to-structure and if it fails, it returns `NULL`.

Programming Tip: A file must be opened before any operation can be performed on it.

File Name

Every file on the disk has a name known as the file name. The naming convention of a file varies from one operating system to another. For example, in DOS, the file name can have one to eight characters optionally followed by a period and an extension that has one to three characters. However, Windows and UNIX permit filenames having maximum of 256 characters. Windows also lays some restrictions on usage of certain characters in the filenames, i.e, characters such as /, \, :, *, ?, ", <, >, and ! cannot be part of a file name.

In C, the `fopen()` function may contain the path information instead of specifying the filename. The path gives information about the location of the file on the disk. If a filename is specified without a path, it is assumed that the file is located in the current working directory. For example, if a file named Student.DAT is located on D drive in directory BCA, then the path of the file can be specified by writing,

```
D:\BCA\Student.DAT
```

In C, a backslash character has a special meaning with respect to escape sequences when placed in a string. So, in order to represent a backslash character in a C program,

you must precede it with another backslash. Hence, this path will be specified as follows in a C program.

```
D:\\BCA\\Student.DAT
```

File Mode

The second argument in the `fopen()` is the `mode`. Mode conveys to C, the type of processing that will be done with the file. The different modes in which a file can be opened for processing are given in Table 15.1.

Table 15.1 File modes

Mode	Description
r	Open a text file for reading. If the stream (file) does not exist, then an error will be reported.
w	Open a text file for writing. If the stream does not exist, then it is created. If the file already exists, then its contents would be deleted.
a	Append to a text file. If the file does not exist, it is created.
rb	Open a binary file for reading. b indicates binary. By default this will be a sequential file in Media 4 format.
wb	Open a binary file for writing.
ab	Append to a binary file.
r+	Open a text file for both reading and writing. The stream will be positioned at the beginning of the file. When you specify 'r+', you indicate that you want to read the file before you write to it. Thus, the file must already exist.
w+	Open a text file for both reading and writing. The stream will be created if it does not exist, and will be truncated if it exists.
a+	Open a text file for both reading and writing. The stream will be positioned at the end of the file content.
r+b/rb+	Open a binary file for read/write.
w+b/ wb+	Create a binary file for read/write.
a+b/ab+	Append a binary file for read/write.

The following code opens a file using the `fopen()` function.

```
FILE *fp;
fp = fopen("Student.DAT", "r");
if(fp == NULL)
{
   printf("\n The file could not be opened");
   exit(1);
}
```

Or

```
char filename[30];
FILE *fp;
gets(filename);
fp = fopen(filename, "r+");
if(fp == NULL)
{
   printf("\n The file could not be opened");
   exit(1);
}
```

Programming Tip: An error will be generated if you try to open a file that does not exist.

We have already discussed that `fopen()` returns a pointer to `FILE` structure if successful and a `NULL` otherwise. So it is recommended to check whether the file was successfully opened before actually using the file. The `fopen()` function can fail to open the specified file under certain conditions that are listed as follows:

- opening a file that is not ready for use
- opening a file that is specified to be on a non-existent directory/drive
- opening a non-existent file for reading
- opening a file to which access is not permitted

15.2.3 Closing a file

To close an open file, the `fclose()` function is used which disconnects a file pointer from a file. After `fclose()` has disconnected the file pointer from the file, the pointer can be used to access a different file or the same file but in a different mode. The `fclose()` function not only closes the file, but also flushes all the buffers that are maintained for that file.

Programming Tip: It is always recommended to close all the opened files when they are not going to be used further in the program.

If you do not close a file after using it, the system closes it automatically when the program exits. However, since there is a limit on the number of files which can be open simultaneously; the programmer must close a file when it has been used. The prototype of the `fclose()` function can be given as

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

Here, fp is a file pointer which points to the file that has to be closed. The function returns an integer value which indicates whether the fclose() was successful or not. A zero is returned if the function was successful; and a non-zero value is returned if an error occurred.

Note When fclose() is executed, any unwritten buffered data for the stream will be written to the file and any unread buffered data will be discarded.

In addition to fclose(), there is an fcloseall() function which closes all the streams that are currently open except the standard streams (such as stdin, stdout, and stderr). The prototype of fcloseall() can be given as,

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

fcloseall() also flushes any stream buffers and returns the number of streams closed.

If a file's buffer has to be flushed without closing it then use fflush() or flushall() to flush the buffers of all open streams.

15.3 READING DATA FROM FILES

C provides the following set of functions to read data from a file.

- fscanf()
- fgetc()
- fgets()
- fread()

In this section, we will read about these functions.

15.3.1 fscanf()

The fscanf() function is used to read formatted data from the stream. Its syntax can be given as

```
int fscanf(FILE *stream, const char *format,...);
```

The fscanf() function is used to read data from the stream and store them according to the parameter format into the locations pointed by the additional arguments. However, these additional arguments must point to the objects that have already occupied memory. These objects are of the type specified by their corresponding format tag within the format string.

Similar to the format specifiers used in scanf(), in fscanf() also, the format specifiers is a C string that begins with an initial percentage sign (%). The format specifier is used to specify the type and format of the data that has to be obtained from the stream and stored in the memory locations pointed by the additional arguments. The prototype of a format specifier can be given as,

[=%[*][width][modifiers]type=] where '*' is an optional argument that suppresses assignment of the input field. It indicates that data should be read from the stream and ignored (not stored in the memory location). width specifies the maximum number of characters to be read. However, fewer characters will be read if the fscanf() function encounters a white space or an inconvertible character. modifiers can be h, l, or L for the data pointed to by the corresponding additional arguments. Modifier h is used for short int or unsigned short int, l is used for long int, unsigned long int, or double values. Finally, L is used for long double data values. type specifies the type of data that has to be read. It also indicates how this data is expected to be read from the user.

The type specifiers for fscanf() function are given in Table 15.2.

Table 15.2 Type specifiers

Type	Qualifying input
C	for single character
D	for decimal values
e, E, f, g, G	for floating point numbers
O	for octal number
S	for a sequence of (string of) characters
U	for unsigned decimal value
x, X	for hexadecimal value

The fscanf() function has some additional arguments. Each of this additional argument must point to an object of the type specified by their corresponding % tag within the format string, in the same order.

Note The fscanf() function is similar to the scanf() function, except that the first argument of fscanf() specifies a stream from which to read, whereas scanf() can only read from standard input.

Let us look at an example which illustrates the use of fscanf(). Here, we will not give the complete program but just a partial program to demonstrate the use of fscanf().

```
#include <stdio.h>
main()
{
   FILE *fp;
   char name[80];
```

```
    int roll_no;
    fp = fopen("Student.DAT", "r");
    if(fp==NULL)
    {
       printf("\n The file could not be opened");
       exit(1);
    }
    printf("\n Enter the name and roll number
    of the student: ");
    // READ FROM KEYBOARD
    fscanf(stdin, "%s %d", name, &roll_no);
    /* read from keyboard */
    printf("\n NAME: %s \t ROLL NUMBER =
    %d", name, roll_no);
    // READ FROM FILE Student.DAT
    fscanf(fp, "%s %d", name, &roll_no);
    printf("\n NAME: %s \t ROLL NUMBER =
    %d", name, roll_no);
    fclose(fp);
    return 0;
}
```

Output

```
Enter the name and roll number of the
   student: 01 Zubin
NAME: Zubin   ROLL NUMBER = 01
NAME: Goransh ROLL NUMBER = 03
```

Note: If you want to use `fprintf()` to write on the screen, then specify stdout instead of specifying any other file pointer.

15.3.2 fgets()

The function `fgets()` stands for *file get string*. This function is used to get a string from a stream. The syntax of `fgets()` can be given as,

```
char *fgets(char *str, int size, FILE *stream);
```

The `fgets()` function reads atmost one less than the number of characters specified by size (gets size – 1 characters) from the given stream and stores them in the string `str`. The `fgets()` terminates as soon as it encounters either a newline character, EOF, or any other error. However, if a newline character is encountered it is retained. When all the characters are read without any error, a `'\0'` character is appended to the end of the string.

The `gets()` and `fgets()` functions are almost same except that `gets()` has an infinite size and a stream of `stdin`. Another difference is that when `gets()` encounters a newline character, it does not retain it, i.e., the newline character (if any) is not stored in the string.

On successful completion, `fgets()` will return `str`. However, if the stream is at EOF, the EOF indicator for the stream will be set and `fgets()` will return a null pointer. In case, `fgets()` encounters any error while reading, the error indicator for the stream will be set and null pointer will be returned. Look at the following program code which demonstrates the use of `fgets()`.

```
#include <stdio.h>
main()
{
   FILE *fp;
   char str[80];
   fp = fopen("ABC.DAT", "r");
   if(fp==NULL)
   {
      printf("\n The file could not be opened");
      exit(1);
   }
   /* the file will read 79 characters
   during each iteration and prints them on
   the screen */
   while (fgets(str, 80, fp) != NULL)
      printf("\n %s", str);
   printf("\n\n File Read. Now closing the file");
   fclose(fp);
   return 0;
}
```

Output

```
Abdceweeferrttet gfejjherroiew tjketjer
   fddfgdfgfd
File Read. Now closing the file
```

15.3.3 fgetc()

The `fgetc()` function returns the next character from stream, and EOF if the end of file is reached, or if there is an error. The syntax of `fgetc()` can be given as,

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

`fgetc()` returns the character read as an `int` or return EOF to indicate an error or end of file.

`fgetc()` reads a single character from the current position of a file (file associated with the stream). After reading the character, the function increments the associated file pointer (if defined) to point to the next character. However, if the stream has already reached the end of file, the EOF indicator for the stream is set. Look at the following program code which demonstrates the use of `fgets()` function.

```
#include <stdio.h>
main()
{
   FILE *fp;
   char str[80];
   int i, ch;
```

```
fp = fopen("Program.C", "r");
if(fp==NULL)
{
   printf("\n The file could not be
     opened");
   exit(1);
}
// Read 79 characters and store them in str
ch = fgetc(fp);
for(i=0; (i<79) && (feof(fp)==0); i++)
{
   str[i] = (char)ch;
   ch = fgetc(stream);
   // reads character by character
}
str[i] = '\0';
// append the string with a null character
printf("\n %s", str);
fclose(fp);
}
```

Output

```
#include <stdio.h>
main()....
displays either first 79 characters or less
   characters if the file contains less than
   79 characters
```

The `feof()` function is used to detect the end of file. We will read more on this function later in this chapter.

15.3.4 fread()

The `fread()` function is used to read data from a file. Its syntax can be given as,

```
int fread(void *str, size_t size, size_t num,
   FILE *stream);
```

The `fread()` function reads `num` number of objects (where each object is `size` bytes) and places them into the array pointed to by `str`. The data is read from the given input stream.

Upon successful completion, `fread()` returns the number of bytes successfully read. The number of objects will be less than `num` if a read error or EOF is encountered. If `size` or `num` is 0, `fread()` will return 0 and the contents of `str` and the state of the stream remain unchanged. In case of error, the error indicator for the stream will be set.

The `fread()` function advances the file position indicator for the stream by the number of bytes read.

> **Note** The `fread()` function does not distinguish between EOF and error. The programmer must use `feof()` and `ferror()` to determine which of the two has occurred.

The following program illustrates the use of `fread()`.

```
#include <stdio.h>
main()
{
   FILE *fp;
   char str[11];
   fp = fopen("Letter.TXT", "r+");
   if(fp==NULL)
   {
      printf("\n The file could not be opened");
      exit(1);
   }
   fread(str, 1, 10, fp);
   /* In the str 10 objects of 1 byte are
   read from the file pointd by fp */
   str[10]= '\0';
   printf("\n First 9 characters of the file
   are: %s",str);
   fclose(fp);
}
```

Output

```
First 9 characters of the file are: Hello
   how
```

Since `fread()` returns the number of bytes successfully read, we can also modify the above program to print the number of bytes read. This would be helpful to know how many characters were read.

```
#include <stdio.h>
main()
{
   FILE *fp;
   char str[80];
   size_t bytes_read;
   fp = fopen("Letter.TXT", "r+");
   if(fp==NULL)
   {
      printf("\n The file could not be opened");
      exit(1);
   }
   bytes_read = fread(str, 1, 79, fp);
   str[bytes_read+1]= '\0';
   // explicitly store null character at the
   end of str
   printf("\n First %d characters of the
   file are: %s", bytes_read, str);
   fclose(fp);
}
```

Output

```
Output will depend on the contents of the
   file. Assuming 14 characters were read,
   the output can be given as
```