**File**

The last chapter explained the standard input and output devices handled by C programming language. This chapter cover how C programmers can create, open, close text or binary files for their data storage.

**A file represents a sequence of bytes, regardless of it being a text file or a binary file.** C programming language provides access on high level functions as well as low level (OS level) calls to handle file on your storage devices. This chapter will take you through the important calls for file management.

Opening Files

You can use the **fopen( )** function to create a new file or to open an existing file. This call will initialize an object of the type **FILE**, which contains all the information necessary to control the stream. The prototype of this function call is as follows −

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

Here, **filename** is a string literal, which you will use to name your file, and access **mode** can have one of the following values −

|  |  |
| --- | --- |
| **Mode** | **Description** |
| r | Opens an existing text file for reading purpose. |
| w | Opens a text file for writing. If it does not exist, then a new file is created. Here your program will start writing content from the beginning of the file. |
| a | Opens a text file for writing in appending mode. If it does not exist, then a new file is created. Here your program will start appending content in the existing file content. |
| r+ | Opens a text file for both reading and writing. |
| w+ | Opens a text file for both reading and writing. It first truncates the file to zero length if it exists, otherwise creates a file if it does not exist. |
| a+ | Opens a text file for both reading and writing. It creates the file if it does not exist. The reading will start from the beginning but writing can only be appended. |

If you are going to handle binary files, then you will use following access modes instead of the above mentioned ones −

"rb", "wb", "ab", "rb+", "r+b", "wb+", "w+b", "ab+", "a+b"

Closing a File

To close a file, use the fclose( ) function. The prototype of this function is −

int fclose( FILE \*fp );

The **fclose(-)** function returns zero on success, or **EOF** if there is an error in closing the file. This function actually flushes any data still pending in the buffer to the file, closes the file, and releases any memory used for the file. The EOF is a constant defined in the header file **stdio.h**.

There are various functions provided by C standard library to read and write a file, character by character, or in the form of a fixed length string.

Writing a File

Following is the simplest function to write individual characters to a stream −

int fputc( int c, FILE \*fp );

The function **fputc()** writes the character value of the argument c to the output stream referenced by fp. It returns the written character written on success otherwise **EOF** if there is an error. You can use the following functions to write a null-terminated string to a stream −

int fputs( const char \*s, FILE \*fp );

The function **fputs()** writes the string **s** to the output stream referenced by fp. It returns a non-negative value on success, otherwise **EOF** is returned in case of any error. You can use **int fprintf(FILE \*fp,const char \*format, ...)** function as well to write a string into a file. Try the following example.

Make sure you have **/tmp** directory available. If it is not, then before proceeding, you must create this directory on your machine.

#include <stdio.h>

main() {

FILE \*fp;

fp = fopen("/tmp/test.txt", "w+");

fprintf(fp, "This is testing for fprintf...\n");

fputs("This is testing for fputs...\n", fp);

fclose(fp);

}

When the above code is compiled and executed, it creates a new file **test.txt** in /tmp directory and writes two lines using two different functions. Let us read this file in the next section.

Reading a File

Given below is the simplest function to read a single character from a file −

int fgetc( FILE \* fp );

The **fgetc()** function reads a character from the input file referenced by fp. The return value is the character read, or in case of any error, it returns **EOF**. The following function allows to read a string from a stream −

char \*fgets( char \*buf, int n, FILE \*fp );

The functions **fgets()** reads up to n-1 characters from the input stream referenced by fp. It copies the read string into the buffer **buf**, appending a **null** character to terminate the string.

If this function encounters a newline character '\n' or the end of the file EOF before they have read the maximum number of characters, then it returns only the characters read up to that point including the new line character. You can also use **int fscanf(FILE \*fp, const char \*format, ...)** function to read strings from a file, but it stops reading after encountering the first space character.

#include <stdio.h>

main() {

FILE \*fp;

char buff[255];

fp = fopen("/tmp/test.txt", "r");

fscanf(fp, "%s", buff);

printf("1 : %s\n", buff );

fgets(buff, 255, (FILE\*)fp);

printf("2: %s\n", buff );

fgets(buff, 255, (FILE\*)fp);

printf("3: %s\n", buff );

fclose(fp);

}

When the above code is compiled and executed, it reads the file created in the previous section and produces the following result −

1 : This

2: is testing for fprintf...

3: This is testing for fputs...

Let's see a little more in detail about what happened here. First, **fscanf()** read just **This** because after that, it encountered a space, second call is for **fgets()** which reads the remaining line till it encountered end of line. Finally, the last call **fgets()** reads the second line completely.

**There are a large number of functions to handle file I/O (Input Output) in C. In this tutorial, you will learn to handle standard I/O in C using fprintf(), fscanf(), fread(), fwrite(), fseek.and more.**

In C programming, file is a place on your physical disk where information is stored.

## Why files are needed?

* When a program is terminated, the entire data is lost. Storing in a file will preserve your data even if the program terminates.
* If you have to enter a large number of data, it will take a lot of time to enter them all.  
  However, if you have a file containing all the data, you can easily access the contents of the file using few commands in C.
* You can easily move your data from one computer to another without any changes.

## Types of Files

When dealing with files, there are two types of files you should know about:

1. Text files
2. Binary files

### 1. Text files

Text files are the normal .txt files that you can easily create using Notepad or any simple text editors.

When you open those files, you'll see all the contents within the file as plain text. You can easily edit or delete the contents.

They take minimum effort to maintain, are easily readable, and provide least security and takes bigger storage space.

### 2. Binary files

Binary files are mostly the .bin files in your computer.

Instead of storing data in plain text, they store it in the binary form (0's and 1's).

They can hold higher amount of data, are not readable easily and provides a better security than text files.

## File Operations

In C, you can perform four major operations on the file, either text or binary:

1. Creating a new file
2. Opening an existing file
3. Closing a file
4. Reading from and writing information to a file

## Working with files

When working with files, you need to declare a pointer of type file. This declaration is needed for communication between the file and program.

FILE \*fptr;

## Opening a file - for creation and edit

Opening a file is performed using the [library function](https://www.programiz.com/c-programming/library-function) in the **"stdio.h"** header file: fopen().

The syntax for opening a file in standard I/O is:

ptr = fopen("fileopen","mode")

For Example:

fopen("E:\\cprogram\\newprogram.txt","w");

fopen("E:\\cprogram\\oldprogram.bin","rb");

* Let's suppose the file newprogram.txt doesn't exist in the location E:\cprogram. The first function creates a new file named newprogram.txt and opens it for writing as per the mode 'w'.  
  The writing mode allows you to create and edit (overwrite) the contents of the file.
* Now let's suppose the second binary file oldprogram.bin exists in the location E:\cprogram. The second function opens the existing file for reading in binary mode 'rb'.  
  The reading mode only allows you to read the file, you cannot write into the file.

| Opening Modes in Standard I/O | | |
| --- | --- | --- |
| File Mode | Meaning of Mode | During Inexistence of file |
| r | Open for reading. | If the file does not exist, fopen() returns NULL. |
| rb | Open for reading in binary mode. | If the file does not exist, fopen() returns NULL. |
| w | Open for writing. | If the file exists, its contents are overwritten. If the file does not exist, it will be created. |
| wb | Open for writing in binary mode. | If the file exists, its contents are overwritten. If the file does not exist, it will be created. |
| a | Open for append. i.e, Data is added to end of file. | If the file does not exists, it will be created. |
| ab | Open for append in binary mode. i.e, Data is added to end of file. | If the file does not exists, it will be created. |
| r+ | Open for both reading and writing. | If the file does not exist, fopen() returns NULL. |
| rb+ | Open for both reading and writing in binary mode. | If the file does not exist, fopen() returns NULL. |
| w+ | Open for both reading and writing. | If the file exists, its contents are overwritten. If the file does not exist, it will be created. |
| wb+ | Open for both reading and writing in binary mode. | If the file exists, its contents are overwritten. If the file does not exist, it will be created. |
| a+ | Open for both reading and appending. | If the file does not exists, it will be created. |
| ab+ | Open for both reading and appending in binary mode. | If the file does not exists, it will be created. |

## Closing a File

The file (both text and binary) should be closed after reading/writing.

Closing a file is performed using library function fclose().

fclose(fptr); //fptr is the file pointer associated with file to be closed.

## Reading and writing to a text file

For reading and writing to a text file, we use the functions fprintf() and fscanf().

They are just the file versions of printf() and scanf(). The only difference is that, fprint and fscanf expects a pointer to the structure FILE.

### Writing to a text file

**Example 1: Write to a text file using fprintf()**

#include <stdio.h>

int main()

{

int num;

FILE \*fptr;

fptr = fopen("C:\\program.txt","w");

if(fptr == NULL)

{

printf("Error!");

exit(1);

}

printf("Enter num: ");

scanf("%d",&num);

fprintf(fptr,"%d",num);

fclose(fptr);

return 0;

}

This program takes a number from user and stores in the file program.txt.

After you compile and run this program, you can see a text file program.txt created in C drive of your computer. When you open the file, you can see the integer you entered.

### Reading from a text file

**Example 2: Read from a text file using fscanf()**

#include <stdio.h>

int main()

{

int num;

FILE \*fptr;

if ((fptr = fopen("C:\\program.txt","r")) == NULL){

printf("Error! opening file");

// Program exits if the file pointer returns NULL.

exit(1);

}

fscanf(fptr,"%d", &num);

printf("Value of n=%d", num);

fclose(fptr);

return 0;

}

This program reads the integer present in the program.txt file and prints it onto the screen.

If you succesfully created the file from **Example 1**, running this program will get you the integer you entered.

Other functions like fgetchar(), fputc() etc. can be used in similar way.

## Reading and writing to a binary file

Functions fread() and fwrite() are used for reading from and writing to a file on the disk respectively in case of binary files.

### Writing to a binary file

To write into a binary file, you need to use the function fwrite(). The functions takes four arguments: Address of data to be written in disk, Size of data to be written in disk, number of such type of data and pointer to the file where you want to write.

fwrite(address\_data,size\_data,numbers\_data,pointer\_to\_file);

**Example 3: Writing to a binary file using fwrite()**

#include <stdio.h>

struct threeNum

{

int n1, n2, n3;

};

int main()

{

int n;

struct threeNum num;

FILE \*fptr;

if ((fptr = fopen("C:\\program.bin","wb")) == NULL){

printf("Error! opening file");

// Program exits if the file pointer returns NULL.

exit(1);

}

for(n = 1; n < 5; ++n)

{

num.n1 = n;

num.n2 = 5n;

num.n3 = 5n + 1;

fwrite(&num, sizeof(struct threeNum), 1, fptr);

}

fclose(fptr);

return 0;

}

In this program, you create a new file program.bin in the C drive.

We declare a structure threeNum with three numbers - n1, n2 and n3, and define it in the main function as num.

Now, inside the for loop, we store the value into the file using fwrite.

The first parameter takes the address of num and the second parameter takes the size of the structure threeNum.

Since, we're only inserting one instance of num, the third parameter is 1. And, the last parameter \*fptr points to the file we're storing the data.

Finally, we close the file.

### Reading from a binary file

Function fread() also take 4 arguments similar to fwrite() function as above.

fread(address\_data,size\_data,numbers\_data,pointer\_to\_file);

**Example 4: Reading from a binary file using fread()**

#include <stdio.h>

struct threeNum

{

int n1, n2, n3;

};

int main()

{

int n;

struct threeNum num;

FILE \*fptr;

if ((fptr = fopen("C:\\program.bin","rb")) == NULL){

printf("Error! opening file");

// Program exits if the file pointer returns NULL.

exit(1);

}

for(n = 1; n < 5; ++n)

{

fread(&num, sizeof(struct threeNum), 1, fptr);

printf("n1: %d\tn2: %d\tn3: %d", num.n1, num.n2, num.n3);

}

fclose(fptr);

return 0;

}

In this program, you read the same file program.bin and loop through the records one by one.

In simple terms, you read one threeNum record of threeNum size from the file pointed by \*fptr into the structure num.

You'll get the same records you inserted in Example 3.

## Getting data using fseek()

If you have many records inside a file and need to access a record at a specific position, you need to loop through all the records before it to get the record.

This will waste a lot of memory and operation time. An easier way to get to the required data can be achieved using fseek().

As the name suggests, fseek() seeks the cursor to the given record in the file.

### Syntax of fseek()

fseek(FILE \* stream, long int offset, int whence)

The first parameter stream is the pointer to the file. The second parameter is the position of the record to be found, and the third parameter specifies the location where the offset starts.

| Different Whence in fseek | |
| --- | --- |
| Whence | Meaning |
| SEKK\_SET | Starts the offset from the beginning of the file. |
| SEKK\_END | Starts the offset from the end of the file. |
| SEKK\_CUR | Starts the offset from the current location of the cursor in the file. |

### Example of fseek()

#include <stdio.h>

struct threeNum

{

int n1, n2, n3;

};

int main()

{

int n;

struct threeNum num;

FILE \*fptr;

if ((fptr = fopen("C:\\program.bin","rb")) == NULL){

printf("Error! opening file");

// Program exits if the file pointer returns NULL.

exit(1);

}

// Moves the cursor to the end of the file

fseek(fptr, sizeof(struct threeNum), SEEK\_END);

for(n = 1; n < 5; ++n)

{

fread(&num, sizeof(struct threeNum), 1, fptr);

printf("n1: %d\tn2: %d\tn3: %d", num.n1, num.n2, num.n3);

}

fclose(fptr);

return 0;

}

This program will start reading the records from the file program.bin in the reverse order (last to first) and prints it.