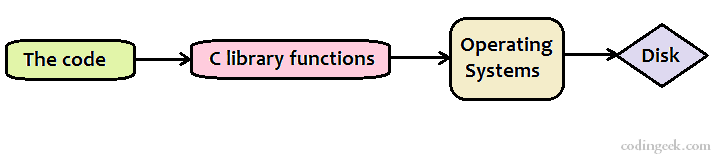
**Files in C Programming Language: Introduction and Various File Modes**

*A****File****is an object that stores data, information, settings or commands. These files can be of different types which are generally used for different purposes. C Programming Language allows****creating, accessing, reading, writing and other file operations that can control the input and output data****.*

The data from the File is stored in the disk space available on the system in the form of binary data. The storage of this data varies from OS to OS. Also, there is a list of library functions that enables us to handle all the file operations.



### **Why do we need Files?**

Here are some reasons why we create files:

* Consider the situation where the program handles a large quantity of data. Now the memory is limited to the amount that can be displayed on the screen. To overcome this limitation, we make use of **Files** that allows us to handle a large quantity of data without any complexities.
* The data displayed on the screen is stored in the memory for only a very short period of time. As the program terminates the data is lost. With the help of **Files**, we can avoid this. The data can be stored for any period of time without losing it i.e. permanent storage.
* Also,**Files** enable a user to transfer data from one system to another without any ambiguity.

### **FileTypes**

Text and Binary are the file types that you should have a basic idea before moving further into the topic.

#### Text Files –

A **text file** contains textual information in the form of alphabets, digits and special characters or symbols.  
This file can be created using a text editor and can be read and understood by a human being as they are in plain language.

#### Binary Files –

A **binary file** contains bytes or a compiled version of a text file i.e. data in the form of 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**

The list below shows the file operations that can be carried out on a file.

* ***Creation of a new file***
* ***Opening or accessing an existing file***
* ***Reading from a file***
* ***Writing to a file***
* ***Seeking in a file or moving to a specific location***
* ***Closing a file***

### **The file pointer**

Before moving on to the file operations it is necessary to understand **file pointers**. The **file pointer** points to the structure that contains details of the file like name, the size of the file, position, read/write status etc. The file pointer is declared as follows:

**FILE \*fp;**

The **FILE** is the structure which stores the details and has been declared in the [header file](https://www.codingeek.com/tutorials/c-programming/inbuilt-and-user-defined-header-files-in-c-language/) “stdio.h“(Standard input/output header file).

### **File Modes for opening a file**

Here are some **file modes** that we shall be using in our code to access the file that is to be operated. Note that all the file modes are strings and must be enclosed in double quotes. The function fopen() is used to open the files. The function is used in the following way:

**fp=fopen(“file\_name”,”mode”);**

Now have a look at the modes which are used by the function fopen().

| **File Mode** | **Description** |
| --- | --- |
| **r** | This file mode opens the file for “*reading*“. If the file doesn’t exist then the function *fopen()* returns NULL. |
| **rb** | This file mode opens the file for “*reading in binary mode*“. If the file doesn’t exist then*fopen()* returns NULL. |
| **w** | Opens the file for “*writing*“. If the file exists the content is overwritten else a new file is created. |
| **wb** | Opens the file for “*writing in binary mode*“. If the file exists the content is overwritten else a new file is created. |
| **a** | Opens the file for “*appending*“, i.e., writing data at the end of the file. If the file doesn’t exist then a new file is created. |
| **ab** | Opens the file for “*appending in binary mode*“. If the file doesn’t exist then a new file is created. |
| **r+** | Opens the file for both “*reading and writing*“. If the file doesn’t exist then the function*fopen()* returns NULL. |
| **rb+** | Opens the file for both “*reading and writing in binary mode*“. If the file doesn’t exist then the function *fopen()* returns NULL. |
| **w+** | Opens the file for both “*reading and writing*“. If the file exists the content is overwritten else a new file is created. |
| **wb+** | Opens the file for both “*reading and writing in binary mode*“. If the file exists the content is overwritten else a new file is created. |
| **a+** | Opens the file for both “*reading and appending*“.  If the file doesn’t exist then a new file is created. |
| **ab+** | Opens the file for both “*reading and appending in binary mode*“.  If the file doesn’t exist then a new file is created. |

# File Operations in C Programming Language

The various file operations are:

* ***Creation of a new file***
* ***Opening or accessing an existing file***
* ***Reading from a file***
* ***Writing to a file***
* ***Seeking in a file or moving to a specific location***
* ***Closing a file***

Before we start with reading or writing to a **file** we must first learn about the *buffer*.

### **Buffer memory**

In order to read or write to a**file**, we must first load the file to a ***buffer*** from the disk. This happens automatically once a file is opened in read or write mode. Now, the question that arises here is why do we need a buffer memory between the disk and the program to read or write or perform other operations to a file? The answer lies in the inefficiency of the process of accessing the disk every single time a character is written to a disk or read from the disk. It takes time for the system to adjust the read/write head correctly every time the file is accessed. So, we use a buffer memory which copies the file from the physical disk and then all the read/write operations are performed on the copy of the file inside the buffer. When the file is closed the changes made in the file are transferred to the file on the disk.

It also prevents us from the scenario (if in case the file is not copied to the buffer and the operations are performed on physical memory) where multiple read and write operations are performed on the same file at the same time. In this case, it might change the data for the other user dynamically and can cause data inconsistency and data corruption.

### **Creating a file**

The file modes play a very important part in creating or opening a file. In order to create a new file, we use the file mode ***“w”***. The syntax for creating a new file is:

FILE \*fp;

fp=fopen("NewDoc.txt","w");

As we already know that if a file with the same name already exists then the data is overwritten otherwise a new file is created.

### **Opening a File**

In order to read or write to a file, we must first open the file using the function fopen(). We open the file in ***“r”*** or read mode which allows us to read the contents of the file. When we open the file in ‘read mode’, the file is first searched for in the disk and then loaded into the buffer. A char pointer is set up which shall point to the first character of the buffer. The syntax for opening a file in ‘read mode’ is:

FILE \*fp;

fp=fopen("Doc.txt","r");

A file can be opened in other modes as well depending upon the programmer’s requirement

### **Reading a File**

When we open a file we assign a pointer variable to the **File** which points to the first character. The file is opened in the ‘read mode’ as shown in the above example. Now, to read the contents of the file we have another function fgetc() which fetches or reads the first character from the current position of the pointer and stores it in another variable and shifts the pointer to the next character.

In this way, the characters in the file are read one by one. The function fgetc() is used inside a while loop which runs indefinitely.But, running a loop indefinitely is not feasible so we have the **EOF** or End of file macro defined in the [header file](https://www.codingeek.com/tutorials/c-programming/inbuilt-and-user-defined-header-files-in-c-language/) ‘stdio.h’. The function fgetc() returns an **EOF** macro once all the characters inside the file are read. We have another function

We have another function getc() which produces the same result as fgetc(). Here is the syntax of how we read a file using the fgetc() function:

**while**(1) {

ch=fgetc(fp);

**if**(ch==EOF)

**break**;

//print ch. }

**Note:** Once we open the file, we don’t need to refer to it by its name. We use the file pointer for that purpose.

### **Writing to a file**

In order to write to a file, we need to open the file in the write mode, i.e., using ***“w”*** mode.

FILE \*fw;

fw=fopen("Doc1.txt","w");

Now that we have opened the file in write mode we use the function fputc() to write to the file. One can write to the file using a while loop and once all the characters are exhausted the loop exits. The fputc() function is used in the following manner:

**char** ch = 'a'; //ch is the variable that contains the character to be written to the file.

fputc(ch,fw);

### **Closing a File**

Once we are done with the reading and writing or other file operation it is important to close the file. We use the function fclose() to close the file. The same file pointer is used to close the file and not the file name.

fclose(fp);//closes the file which was opened in read mode.

fclose(fw);//closes the file that was opened in write mode.

**Note :-** Once we close the file, the function getc() can no longer read the contents of the file unless the file is reopened. Also, the data written to the file will be written to the file on the disk and the buffer will be eliminated from the memory. Consider a situation where the buffer gets full before closing the file and we still have data to write to the file. In this case, the buffer will copy all the data to the file present on the disk once it becomes full. This occurs automatically with the help of library functions.

### **Program to Read the contents of a File and Write a string into another file**

#include <stdio.h>

int main() {

FILE \*fp, \*fw; // we create a file read pointer and a file write pointer

fp = fopen("E:\\Doc1.txt", "r"); // opens the txt file present in E:// drive in read mode

char ch;

/\*We read the contents of the file Doc1.txt and print it\*/

while (1) {

ch = fgetc(fp); // reading the content of the file

if (ch == EOF) // checks whether the loop reached the end of file

break;

printf("%c", ch);

}

fw = fopen("E:\\Doc2.txt","w"); // opens the file in write mode. Since there's no file with the name Doc2.txt, a new file is created.

char ch1[20] = "Codingeek..!!";

int i = 0;

/\*copying the string in the array ch1 to the file Doc2\*/

while (ch1[i] != '\0') {

fputc(ch1[i], fw);

i++;

}

fclose(fp); // close the file Doc1.txt

fclose(fw); // close the file Doc2.txt

return 0;

}

**Output:-**

Hello....!!!

# IO in Files – Examples and explanation of fgets, fputs, fprintf, fscanf, fread, fwrite

If we want to store a complete string at one go what should we do? Or what if we want to write numbers in the file or a combination of numbers and characters? Or write a record in a file using structures?*Using*fputc()*will not be very feasible. So we are now going to see other file library functions that allow us to solve these problems.*

### **String I/O in Files**

Using a while loop and the function fputc() to store a string or a character array in a **file** will make the program more complex and a bit inefficient as well. Therefore, to avoid these we have another function fputs() which directly copies the whole string to the file without the use of any loop.Here is the syntax of the fputs() function:

fputs(string\_variable,file\_pointer);

where the *file pointer* points to the file where the string is to be added.

Similarly, when reading a String from a file we can simply use the function*fgets()* to read it from the file instead of using the while loop to retrieve each character from the file one by one.Here is the syntax of the*fgets()* function:

fgets(string\_name, length of the string, file pointer);

Unlike the *fputs()* function the *fgets()* function takes three arguments: the name of the variable that will store the string, the length of the string and the file pointer which points to the file from where we have to read the data.

The Program below copies the string in file **Doc1.txt** to another file **Doc2.txt** using the functions *fgets()* and *fputs()*:

#include <stdio.h>

int main() {

FILE \*fp, \*fw; // file pointers for the respective read and write mode.

char s[30];

fp = fopen("E://Doc1.txt", "r"); // The file that contains the String

fw = fopen("E://Doc2.txt", "w"); // The file from which the string is being copied

/\*while runs until it gets NULL\*/

while (fgets(s, 30, fp) != NULL) { // reading the string from the file Doc1.txt

fputs(s, fw); // copying to the file Doc2.txt

printf("%s", s);

}

fclose(fp);

fclose(fw);

return 0;

}

**Output:-**

Codingeek..!!

Hello people...!!

### **I/O of Records in Files using Structures**

In the previous section, we have been dealing with string data only. What if we have a record which contains string data, number data as well as float data? We have already seen that we can easily handle records using [**structures**.](https://www.codingeek.com/tutorials/c-programming/structures-in-c-programming-languagebasics-declaration-and-accessing/) Now, the problem arises as to how we can add the numeric data to the file without worrying about the type conversions? This problem can be easily solved using the function fprintf().

fprintf() is very similar to the function printf() and it allows the user to write any type of data into.the file. The only difference being the file pointer present in the arguments.

Similarly, to read any type of data from the file we can use the function fscanf(). fscanf() is similar to scanf() with the only difference being the file pointer present in the arguments. Here is a program that reads the records from one file **Doc1.txt** and copies it to another file **Doc2.txt**:

#include <stdio.h>

#include <stdlib.h>

int main() {

FILE \*fp, \*fw;

/\*creating a structure to store the particulars of the students from the file Doc1.txt\*/

struct student {

char name[20];

int id;

float marks;

};

struct student s;

fp = fopen("E://Doc1.txt", "r"); // The file from which data is copied

fw = fopen("E://Doc2.txt", "w"); // the new file into which the data is copied

/\*copying the student record from Doc1.txt to Doc2.txt. The file runs until end of file is reached\*/

while (fscanf(fp, "%s%d%f", &s.name, &s.id, &s.marks) != EOF) {

fprintf(fw, "%s %d %f", s.name, s.id, s.marks);

fprintf(fw, "\n"); // To add new line after every record that is read from the File

}

fclose(fp);

fclose(fw);

return 0;

}

### **Using fwrite() and fread() to read and write data in files**

Now, in case we have a lot of data elements or fields in the structure then writing the data or reading the data using the function fprintf() or fscanf() won’t be efficient. Hence we have another function fwrite() to write the structure data into the file and fread() to read the structure data from the file.

The function fwrite() takes four arguments, i.e.,

1. The address of the structure,
2. The size of the structure in bytes,
3. The number of structure that we want to write to the file at a time and
4. The file pointer.

The program below takes student details from the user and copies it to the file using the function fwrite().

#include <stdlib.h>

int main() {

FILE \*fw;

/\* adding three records to the file. It can be any desired value or can be dependent on user input. It is hard coded only for the sake of simplicity, in real scenarios these are not hardcoded values. \*/

int count = 3;

/\*creating a structure to store the particulars of the students from the file Doc1.txt\*/

struct student {

char name[20];

int id;

int age;

float marks;

};

struct student s;

fw = fopen("E://Doc2.dat", "w"); // the new file into which the data is copied

while (count != 0) {

printf("Enter the name,id,age and marks of a student:\n");

scanf("%s %d %d %f", &s.name, &s.id, &s.age, &s.marks);

fwrite(&s, sizeof(s), 1, fw); // writes the data stored in the function to the file

fflush(stdin); // to flush out the data remaining in the buffer

count--;

}

fclose(fw);

return 0;

}

The function fread() takes the same number of arguments, i.e. the address of the structure, the size of the structure, the number of structures whose data is to be read and the file pointer. The program below reads the data that we entered using the fwrite() function from Doc2.dat and prints it:

### int main() {

### FILE \*fp;

### /\*creating a structure to store the particulars of the students from the file Doc1.txt\*/

### struct student {

### char name[20];

### int id;

### int age;

### float marks;

### };

### struct student s;

### fp = fopen("E://Doc2.dat", "rb"); // the new file from which data is read

### if (fp == NULL) {

### printf("Cannot open file");

### exit(1);

### }

### /\*reading the data from the file using fread()\*/

### while (fread(&s, sizeof(s), 1, fp) == 1) {

### printf("%s %d %d %f ", &s.name, &s.id, &s.age, &s.marks);

### printf("\n");

### }

### fclose(fp);

### return 0;

### }

### **Library function fflush()**

In the code, we come across a new function called fflush(stdin). This function flushes out all the data that is present in the buffer to the disk. We use this function because of the ambiguity of the scanf() function which stores the enter or new line data in the buffer only unless the file ends.  stdin stands for standard input device or the keyboard in this context.

**Note:** Also, we see that the file that we created is not a text file but are actually .dat files. The reason being the fact that the text file would occupy more bytes because the characters in a text file are stored in the form of a string. And if we open this file **Doc.dat** using notepad we will see unidentified symbols.

# Text Files vs Binary Files in C Programming Language

*The major difference between these two is that a****text file****contains textual information in the form of alphabets, digits and special characters or symbols. On the other hand, a****binary file****contains bytes or a compiled version of a text file.*

## 1. Differences between Text and Binary file

* A **text file** stores data in the form of alphabets, digits and other special symbols by storing their ASCII values and are in a human readable format. For example, any file with a .txt, .c, etc extension. Whereas, a **binary file** contains a sequence or a collection of bytes which are not in a human readable format. For example, files with .exe, .mp3, etc extension. It represents custom data.
* A small error in a **textual file** can be recognized and eliminated when seen. Whereas, a small error in a **binary file** corrupts the file and is not easy to detect.
* Since the data is not human readable it also adds to the security of the content as one might not be able to get data if the structure is not known.
* Now, when it comes to programming there are three major differences between the two, i.e., **Handling of newlines, storage of numbers and representation of EOF(End of File)**. Let’s look into these differences in detail:

## 2. Handling of Newlines

***Newline*** is the end of the line or line ending or line break. It is usually a special character which signifies the end of the line. A **newline** character in a text file is first converted into a carriage return-linefeed combination and then written to the disk. Similarly, when read by a **text file** the carriage return-linefeed combination is converted into a **newline**. However, in a **binary file**, no such conversions take place.

## 3. Storage of Numbers

In the**text mode**, the function fprintf() is used to store numerical data in the disk. The texts and the characters are stored one character per byte as it should be (char occupies 1 byte in the memory) and as expected the integers should occupy 4 bytes(depends on the compiler) per number. But this is not the case. For example, we have a number 567392. According to integer storage convention, it should occupy 4 bytes in the disk but it does not. It occupies 6 bytes,i.e., 1 byte for every digit in the number. Also, the number 56.9057 will occupy 7 bytes in the disk. Thus, we see that each digit in the file is treated as a character in itself and occupies more space than necessary. So, if we have a lot of numerical data then using a text file will not be very memory efficient( but still the syntax used depends on our usage i.e. if we have uses in which a human has to read the file then we can never choose the binary type).

This problem can be solved by using **binary files**. We should open the file in binary mode(using **“wb”** or **“rb”** for write and read mode respectively). The, using the function fread() or fwrite() we can easily store the data in the binary form which shall use only 4 bytes for storing the integer data.

## 4. Representation of EOF

Another way the **text mode** and the **binary mode** can be distinguished is on the basis of the representation of the **end-of-file(EOF)**. In the **text mode**, a special character with the ASCII code 26 is inserted at the end of the file. This character when encountered returns the EOF signal to the program.

This is not the case in **binary mode**. In the **binary mode**, we do not have any special character to signify the EOF. It keeps track with the help of the number of characters present in the directory entry of the file.