# CASE STUDY

## COMPUTER PROGRAMMING

**DONE BY-** Kumari Mahima, Rajdeep Kaur, Gagandeep Kaur

**UID-** 24BCA10111, 24BCA10116, 24BCA10120

**CLASS-** 24BCA2

**GROUP-** B

# FileHandling

## Introduction :

In programming, we may require some specific input data to be generated several numbers of times. Sometimes, it is not enough to only display the data on the console. The data to be displayed may be very large, and only a limited amount of data can be displayed on the console, and since the memory is volatile, it is impossible to recover the programmatically generated data again and again. However, if we need to do so, we may store it onto the local file system which is volatile and can be accessed every time. Here, comes the need of file handling in C.

File handling in C enables us to create, update, read, and delete the files stored on the local file system through our C program. The following operations can be performed on a file.

1. Creation of the new file
2. Opening an existing file
3. Reading from the file
4. Writing to the file
5. Deleting the file

# Why file handling in C ?

There are occasions when a program's output, after it has been compiled and run, does not meet our objectives. In these circumstances, the user would want to verify the program's output several times. It takes a lot of work for any programmer to compile and run the same program repeatedly. This is the precise situation where file handling is helpful.

- **Reusability:** The process of File handling helps the user by allowing him to preserve all the information or the data that is generated after he runs the designed program.
- **Saves Time:** In some cases, the programs need a large amount of user input. Here, file handling allows the user to access part of the code using individual commands easily.
- **Commendable storing capacity:** While storing data in the files, the user can leave behind the stress of storing information in bulk in the program.
- **Portability:** The availability of contents in any file is easily transferable to another without any hassle and data loss in the system. Hence, saving a lot of time and effort along with minimizing the risk of flawed coding.

# Types of Files in a C Program

When referring to file handling, we refer to files in the form of data files. Now, these data files are available in 2 distinct forms in the C language, namely:

- Text Files
- Binary Files

Example : Program to Open a File, Read from it, And Close the FileText Files

A text file contains data in the form of ASCII characters and is generally used to store a stream of characters.

Each line in a text file ends with a new line character (‘\n’).

It can be read or written by any text editor.

They are generally stored with .txt file extension.

Text files can also be used to store the source code.

2. Binary Files

A binary file contains data in binary form (i.e. 0’s and 1’s) instead of ASCII characters. They contain data that is stored in a similar manner to how it is stored in the main memory.

The binary files can be created only from within a program and their contents can only be read by a program.

More secure as they are not easily readable.

They are generally stored with .bin file extension.

**FUNCTION:** There are many functions in the C library to open, read, write, search and close the file. A list of file functions are given below:

| No. | Function | Description |
|---|---|---|
| 1 | fopen() | opens new or existing file |
| 2 | fprintf() | write data into the file |
| 3 | fscanf() | reads data from the file |
| 4 | fputc() | writes a character into the file |
| 5 | fgetc() | reads a character from file |
| 6 | fclose() | closes the file |
| 7 | fseek() | sets the file pointer to given position |
| 8 | fputw() | writes an integer to file |
| 9 | fgetw() | reads an integer from file |
| 10 | ftell() | returns current position |
| 11 | rewind() | sets the file pointer to the beginning of the file |

### File Pointer in C

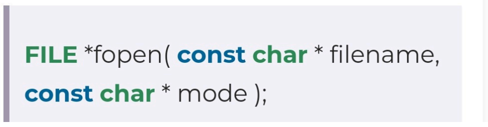
In file handling in C, a file pointer serves as a reference to a specific position within an opened file. It facilitates various file operations in C like reading, writing, closing, and more. The FILE macro is employed to declare a file pointer variable, and it is defined within the <stdio.h> header file.

#### Syntax of File Pointer

## Opening File: fopen()

## FILE* fopen(const char *f_name, const char *access_mode);

We must open a file before it can be read, write, or update. The fopen() function is used to open a file. The syntax of the fopen() is given below.



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

The fopen() function accepts two parameters:

- The file name (string). If the file is stored at some specific location, then we must mention the path at which the file is stored. For example, a file name can be like **"c://some_folder/some_file.ext"**.
- The mode in which the file is to be opened. It is a string.

We can use one of the following modes in the fopen() function.

| No. | Function | Description |
|---|---|---|
| 1 | fopen() | opens new or existing file |
| 2 | fprintf() | write data into the file |
| 3 | fscanf() | reads data from the file |
| 4 | fputc() | writes a character into the file |
| 5 | fgetc() | reads a character from file |
| 6 | fclose() | closes the file |
| 7 | fseek() | sets the file pointer to given position |
| 8 | fputw() | writes an integer to file |
| 9 | fgetw() | reads an integer from file |
| 10 | ftell() | returns current position |
| 11 | rewind() | sets the file pointer to the beginning of the file |

The fopen function works in the following way.

- Firstly, It searches the file to be opened.
- Then, it loads the file from the disk and place it into the buffer. The buffer is used to provide efficiency for the read operations.
- It sets up a character pointer which points to the first character of the file.

Consider the following example which opens a file in write mode.

```
FILE *fp ;
char ch ;
fp = fopen("file_handle.c","r") ;
while ( 1 )
{
ch = fgetc ( fp ) ;
if ( ch == EOF )
break ;
printf("%c",ch) ;
}
fclose (fp ) ;
}
```

## Output

The content of the file will be printed.

```
{
FILE *fp; // file pointer
char ch;
fp =
fopen("file_handle.c","r");
while ( 1 )
{
ch = fgetc ( fp ); //Each
character of the file is read
and stored in the character
file.
if ( ch == EOF )
break;
printf("%c",ch);
}
fclose (fp );
}
```

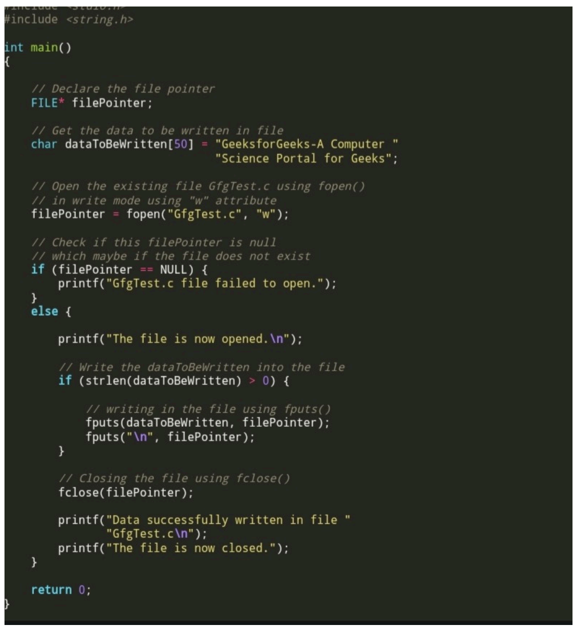
## Closing File: fclose()

The fclose() function is used to close a file. The file must be closed after performing all the operations on it. The syntax of fclose() function is given below:

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

Program to Open a File, Read from

it, And Close the File.



```c
#include <string.h>

int main()
{

    // Declare the file pointer
    FILE* filePointer;

    // Get the data to be written in file
    char dataToBeWritten[50] = "GeeksforGeeks-A Computer "
                               "Science Portal for Geeks";

    // Open the existing file GfgTest.c using fopen()
    // in write mode using "w" attribute
    filePointer = fopen("GfgTest.c", "w");

    // Check if this filePointer is null
    // which maybe if the file does not exist
    if (filePointer == NULL) {
        printf("GfgTest.c file failed to open.");
    }
    else {

        printf("The file is now opened.\n");

        // Write the dataToBeWritten into the file
        if (strlen(dataToBeWritten) > 0) {

            // writing in the file using fputs()
            fputs(dataToBeWritten, filePointer);
            fputs("\n", filePointer);
        }

        // Closing the file using fclose()
        fclose(filePointer);

        printf("Data successfully written in file "
               "GfgTest.c\n");
        printf("The file is now closed.");
    }

    return 0;
}
```

**Output :**

```
The file is now opened.

GeeksforGeeks-A Computer Science Portal for Geeks D

ata successfully read from file GfgTest.cThe file is now closed.
```

# fprintf Function

## Writing File: fprintf() function

The fprintf() function is used to write set of character into file. It sends formatted output to a stream.

## Syntax:

```
int fprint(FILE *STREAM, const char *format [, argument, ...]
```

## Example:

```
#include <stdio.h>
Int main() {
  FILE *fp;
  Fp = fopen("file.txt", "w");//opening file
  Fprintf(fp, "Hello file by fprintf...\n");//writing data into file
Fclose(fp);//closing file
```

## fscanf  Function

### Reading File: fsanf() function

The fscanf() function is used to read set of characters from file. It reads a word from the file and returns EOF at the end of file.

### Syntax:

```
Int fscanf(FILE  *stream, const char *format [, argument, ...])
```

### Example:

```
#include <stdio.h>
Int main() {
    FILE *fp;
    char buff[255];//creating char array to store data of file
    fp= fopen("file.txt","r");
    while(fsanf(fp, "%s",buff)!=EOF){
    printf("%s", buff);
}
fclose(fp);
}
```

## C File Example: Storing employee information

Let’s see a file handling example to store employee information as entered  by user from console. We are going to store id, name and salary of the employee.

```
#include <stdio.h>
Void main()
{
   FILE *fptr;
   int  id;
   char name[30];
   float salary;
   fptr = fopen(“emp.txt”, “w+”);/* open for writing */
   if (fptr == NULL)
   {
       Printf(“File does not exist \n”);
       Return;
}
   printf(“Enter the id\n”);
   Scanf(“%d”,&id);
   fprintf(fptr, “Id= %d\n”, id);
   printf(“Enter the name\n”);
   Scanf(“%s”, name);
   fprintf(fptr, “Name= %s\n”, name);
   printf(“Enter the salary\n”);
   scanf(“%f”, &salary);
   fprintf(“%f”, &salary= %.2f\n”, salary);
   Fclose(fptr);
}
```

**Output:**

```
Enter the id
1

Enter the name
Mahima

Enter the salary
6000000
```

Now open file from current directory. For windows operating system, go to TC\bin directory, you will see emp.txt file. It will have following information.

**emp.txt**

```
Id = 1
Name= Mahima
Salary= 6000000
```

# C fputs() and fgets()

The fputs() and fgets() in C programming are used to write and read string from stream. Let's see examples of writing and reading file using fgets() and fgets() functions.

## Writing File : fputs() function

The fputs() function writes a line of characters into file. It outputs string to a stream.

### Syntax:

```
int fputs(const char *s, FILE *stream)
```

### Example:

```
#include<stdio.h>
#include<conio.h>
void main() {
FILE *fp;
clrscr();

fp=fopen("myfile2.txt","w");
fputs("hello c programming",fp);

fclose(fp);
getch();
}
```

### myfile2.txt

```
hello c programming
```

## Reading File : fgets() function

The fgets() function reads a line of characters from file. It gets string from a stream.

### Syntax:

```
char* fgets(char *s, int n, FILE *stream)
```

### Example:

```
#include<stdio.h>
#include<conio.h>
void main(){
FILE *fp;
char text[300];
clrscr();

fp=fopen("myfile2.txt","r");
printf("%s",fgets(text,200,fp));

fclose(fp);
getch();
}
```

### Output:

```
hello c programming
```

# fputc() function

## Writing File : fputc() function

The fputc() function is used to write a single character into file. It outputs a character to a stream.

### Syntax:

```
int fputc(int c, FILE *stream)
```

### Example:

```
#include <stdio.h>
main(){
   FILE *fp;
   fp = fopen("file1.txt", "w");//opening file
   fputc('a',fp);//writing single character into file
   fclose(fp);//closing file
}
```

### file1.txt

```
a
```

# fgetc() function

## Reading File : fgetc() function

The fgetc() function returns a single character from the file. It gets a character from the stream. It returns EOF at the end of file.

### Syntax:

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

### Example:

```
#include<stdio.h>
#include<conio.h>
void main(){
FILE *fp;
char c;
clrscr();
fp=fopen("myfile.txt","r");

while((c=fgetc(fp))!=EOF){
printf("%c",c);
}
fclose(fp);
getch();
}
```

## C fseek() function

The fseek() function is used to set the file pointer to the specified offset. It is used to write data into file at desired location.

### Syntax:

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

There are 3 constants used in the fseek() function for whence: SEEK_SET, SEEK_CUR and SEEK_END.

### Example:

```
#include <stdio.h>
void main() {
  FILE *fp;

  fp = fopen("myfile.txt","w+");
  fputs("This is c", fp);

  fseek( fp, 7, SEEK_SET );
  fputs("Rajdeep Kaur", fp);
  fclose(fp);
}
```

### myfile.txt

```
This is Rajdeep Kaur
```

### Understanding the fseek() Parameters:

Let's understand the *input parameters* for the *fseek() function* is essential for complete use:

## Example 1:

Consider the following text in a file called ***"data.txt"***:

```
Line 1
Line 2
Line 3
Line 4
Line 5
```

## Code:

```c
#include <stdio.h>

int main() {
FILE *file = fopen("data.txt", "r");
if (file == NULL) {
printf("Unable to open the file.\n");
return 1;
}

char line[100];
fseek(file, 4, SEEK_SET);  // Move to the beginning of the fourth line
fgets(line, sizeof(line), file);
printf("Fourth line: %s", line);

fclose(file);
return 0;
}
```

The *fseek() function* in the C language allows programmers to set the file pointer to a given offset inside a *file*, making it an essential tool for *file manipulation*. Developers may perform various tasks, including *random access, adding data, editing existing material,* and *reading particular file sections*, skillfully utilizing the *fseek() function*.

For *fseek()* to operate properly, it is crucial to comprehend its syntax, which comprises the *FILE pointer, offset,* and *whence arguments*. The reference location from which the offset is computed is determined by the three variables *SEEK_SET, SEEK_CUR*, and *SEEK_END*, enabling precise control over the movement of the file pointer.

Programmers may manage *big files*, access data structures in binary files, and modify metadata sections with the help of *fseek()*. Additionally, the function has error-handling features that make it possible to identify and fix problems like looking outside of file boundaries or using closed files.

When using *fseek()*, it's crucial to consider the compromise between ease and speed. Performance can be impacted by excessive random-access operations,

**Metadata updating:**

In some circumstances, *metadata* may be present in places in files. You may access these metadata sections using *fseek()* and make the necessary changes. It is typical when working with file formats that include *headers* or other descriptive information.

**Support for huge Files:**

The *fseek()* function is compatible with *huge files* when given a *file offset* of type *long int*. You may now work with files that are larger than what is allowed by standard file operations.

**Consideration for performance:**

Although *fseek()* function offers flexibility, frequent use of random-access operations can have an adverse effect on speed, particularly with big files. Consider the trade-off between convenience and performance when considering whether to use *fseek()* for file manipulation.

Here are several examples of how to use the C *fseek() function* and their accompanying results:

**A file position indicator update:** The *fseek() method* automatically updates the file position indication to reflect the changed location of the file pointer. By doing this, you may be confident that the file will always be operated on from the right angle.

Regular Use Cases of fseek()

There are several regular use cases of fseek() function. Some use cases are as follows:

**Random Access:** The *fseek() function* allows you random access to a file, enabling you to *read* or *write data* at any point you want inside the file. It is especially beneficial for huge files or databases when sequential data access is not the most effective method.

**Modifying Existing Data:** Without rewriting the whole file, *fseek()* lets you edit data at specified locations inside a file. You can replace or insert *new data* while keeping the present contents by shifting the file pointer to the appropriate spot.

**Data Appending:** You may relocate the file pointer to the end of the file and add additional data by using the *fseek() function* with the *SEEK_END constant*. It is quite helpful when working with log files or adding entries to an existing file.

**Reading Specific Portions:** You can *read only* the information you need by using the *fseek() function* to move to a *specific location* inside a file. Working with huge files or structured data can improve file reading procedures.

File*: The *FILE *stream* argument is a reference to the file's associated *FILE structure*. By utilizing the *fopen() method*, it is acquired.

Long int offset: The offset parameter specifies how many bytes will relocated to the whence parameter's location. It may be *zero, positive*, or *negative*.

***Whence*** is an integer that indicates the reference place from which the offset is determined. Any one of the following three constants may be used:

**SEEK_SET:** It establishes the offset with respect to the file's start.

**SEEK_CUR:** It modifies the *file pointer's offset* to its current location.

**SEEK_END:** It establishes the offset with respect to the file's end.

Detailed Understanding of the fseek() Function

The three main tasks of the fseek() function are used to move the file pointer, write data to the specified place, and update the file position indication.

**The file pointer is being moved:** You can shift the file pointer to a specified area of the file using the *fseek() method*. You can change the reference point for the offset by altering the *'whence' parameter* to one of the values *SEEK_SET*, *SEEK_CUR*, or *SEEK_END*.

**Putting Data in the Right Place:** Using operations like *fprintf()*, *fputs()*, *fwrite()*, etc., you may write data after shifting the *file reference* to the correct spot. The data will be written beginning at the file pointer's new location.

especially when dealing with *big files*. Therefore, it is essential to analyze how to optimize file handling and reduce pointless carefully seeks.

Developers may improve their ability to manipulate files by understanding the *fseek() method*, which will make their programs more *effective*, *adaptable*, and *resilient*. Programmers can easily handle sophisticated file operations due to *fseek()*, which enables file navigation, data modification, and content adding.

*fseek()* is a useful tool for manipulating file pointers and accessing data at precise points in the world of file management in C. By embracing and comprehending the potential of fseek(), developers may handle files with a new degree of precision and control, resulting in more complex and useful programs.

# C rewind() function

The rewind() function sets the file pointer at the beginning of the stream. It is useful if you have to use stream many times.

The rewind() function sets the file pointer at the beginning of the stream. It is useful if you have to use stream many times.

## Syntax:

```
void rewind(FILE *stream)
```

Example:

File: file.txt

```
this is a simple text
```

Test it Now

File: rewind.c

```
#include<stdio.h>

  #include<conio.h>

void main()

{  FILE *fp;

char c;

clrscr();

 fp=fopen("file.txt","r");
 while((c=fgetc(fp))!=EOF)

{

printf("%c",c);

  }

rewind(fp);

//moves the file pointer at beginning of the file

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

  printf("%c",c);

}

  fclose(fp);

  getch();

}
```

## Output:

```
this is a simple textthis is a simple text
```

# C ftell() function

The ftell() function returns the current file position of the specified stream. We can use ftell() function to get the total size of a file after moving file pointer at the end of file. We can use SEEK_END constant to move the file pointer at the end of file.

Syntax:

```
long int ftell(FILE *stream)
```

File: ftell.c

```
#include <stdio.h>

#include <conio.h>

void main (){

 FILE *fp;

 int length;

 clrscr();

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

 fseek(fp, 0, SEEK_END);



 length = ftell(fp);



 fclose(fp);

 printf("Size of file: %d bytes", length);



 getch();

}
```

**Output:**

```
Size of file: 21 bytes
```

**explanation:**

Using ***fseek(fp, 0, SEEK_END),*** the program accesses the file ***"file.txt"*** in ***read mode*** and advances the file pointer to the end. Using ***ftell(fp)***, it obtains the file location, which represents the ***file size***. Finally, the program uses the ***printf() function*** to output the file's size in bytes. The result shows that ***"file.txt"*** has ***21 bytes*** in it. You should take note of the program's use of uncommon functions like ***clrscr()*** and ***getch()***.

Knowing the file's position and size is crucial for various tasks when working with files in C programming. As the preceding example shows, the ***ftell() function*** is essential in giving this information. Let's look more closely at some additional crucial file operations and ideas since file management encompasses more than simply determining the file size.

**Modes for Opening Files:** In the example, we opened the file in ***"r" mode***. This mode indicates ***read-only access***. Additional access modes are available, including ***"w"*** for writing (creates a new file or overwrites an existing one), ***"a"*** for append (writes at the end of the file), ***"rb"*** for binary read, ***"wb"*** for binary write, etc. It is essential to comprehend and use the proper file modes to guarantee that the correct file actions are carried out.

**Error Handling:** It's critical to manage potential failures when working with file operations properly. If the ***fopen() method*** cannot open the file, it returns ***NULL***. Therefore, looking at the return value and responding appropriately is crucial if the file opening doesn't succeed. ***Error management*** helps you find and fix possible problems and prevent program crashes.

**File Reading and Writing:** Although the function ***fte*****ile Reading and Writing:** Although the function ***ftell()*** is generally used to determine the size of a file, file handling frequently includes ***reading data*** from and ***writing data*** to files. For these uses, C has several methods, including ***fread(), fwrite(), fgets(), and fputs()***. You may efficiently modify the content of files by becoming familiar with and using these functions.

**File Seeking:** The ***fseek() function*** is used in conjunction with ***ftell()*** to adjust the file position indication to a particular point inside the file. It is helpful when you want to read or write data from a specific location in the file. You can traverse a file and perform the necessary operations with the ***fseek() function***.

**Text vs. Binary Mode:** Both ***binary mode*** and ***text mode*** are available for opening files. While ***text mode*** is used for ***text files***, binary mode is appropriate for non-text files. Different platforms (such as ***Windows*** and Unix-like systems) may treat newline characters differently when opening a file in text mode. In binary mode, there are no such transformations and data is ***read***, and written just as it is.

## Conclusion:

In conclusion, the C ***ftell() method*** is a helpful resource for locating the current file location and retrieving the file size. Programmers may easily search through files and do actions by combining them with the ***fseek() function***. Successful file handling in C depends on the understanding and use of file ***opening modes, error management***, and many file-related functions. Furthermore, understanding the differences between binary and text modes guarantees precise data handling. C programmers who are skilled in file handling can read, write, and modify files efficiently, enabling the creation of various programs with exact file operations.