File I/O

Purpose of files in C:

- Files are used to store content hence **reducing the program's size**.
- We can **read or access data** from files.
- The data in files remain stored even after the program's execution is **terminated**.

Files are stored in **non-volatile memory**. To understand what a non-volatile memory is, we have to see the difference between volatile and non-volatile memory.

VOLATILE MEMORY	NON- VOLATILE MEMORY
The data can only remain in it while the computer’s power is on.	The data will also be present in it while the computer’s power is off.
Can only hold information when having a constant power supply	Can also hold information, in case of inconstant power supply
Will hold data for a short period.	Will hold data for a long term.
RAM is an example	Hard Disk is an example.

So due to the above differences, we can conclude the reasons we need files.

Types of Files:

There are two types of files:

- Binary Files
- Text Files

Binary Files:

- Binary files stores data in 01 i.e., **binary format**.
- They are not directly readable.
- An application or software is required to read binary files.
- An example is a **.doc** file.

Text Files:

- They store data in simple **text format**.
- They are directly readable.
- No software is required to access them.
- An example is a **.txt** file.

Operations on files:

By using C language, we can perform four different tasks related to files. We will see them theoretically in this tutorial, and in the next one's, we will see their practical implementation.

Creating a File:

We can create a file using C language, in any directory, without even leaving our compiler. We can select the name or type we want our file to have, along with its location.

Opening a File:

We can open an existing file and create a new file and open it using our program. We can perform different operations on a file after it has been opened.

Closing a File:

When we are done with the file, meaning that we have performed whatever we want to perform on our file, we can close the file using the close function.

Read/Write to a file:

After opening a file, we can access its contents and read, write, or update them.

Modes:

Before discussing Files' functions, we have to learn about different modes used along with these functions as a parameter. The following are the modes:

- **r**: opens a file for reading.
- **w**: opens a file for writing. It can also create a new file.
- **a**: opens a file for appending.
- **r+**: opens a file for both reading and writing but cannot create a new file.
- **w+**: opens a file for both reading and writing.

Note: there are many other modes, but these are the basic and most used ones.

Opening a File:

We use the fopen() function for opening files in C.

Syntax:

```
ptr = fopen(“file_location”,”mode”);
```

Example:

```
ptr = fopen(“D:\\file.txt”,”w”);
```

Closing a File:

Closing an open file is one of the most crucial steps while dealing with C. Files does not automatically get closed after working with them. We have to close them manually. To close a file, we have to use the **fclose()** function. The syntax is straightforward because we just have to send the pointer as a parameter to the function.

Syntax:

```
fclose(fptr);
```

Reading a File:

To read a file in C, we use a function fscanf(). This function is a file version of **scanf()**. Like scanf() used to get input from the keyboard, it gets its input from a file and prints it onto the screen. We have to send the file pointer as an argument for the program to be able to read it. The file has to be opened in r mode, i.e., read mode, to work properly for fsanf().

Example:

```
ptr = fopen(“D:\\file.txt”,”r”);
char str[128] = "Welcome to code with Harry";
fscanf(ptr, "%s", str);
printf("%s",str );
```

Writing to a file:

In order to write to a file, we use the function **fprintf()**. The function is a file version of printf(). Same as we used to print text onto the screen using printf(), we use fprintf() to print text inside the file. We have to send the file pointer as an argument for the program to be able to print it into the file. The file has to be opened in w mode, i.e. write mode, to work properly for fsanf().

Example:

```
char str[128] = "Empty";
ptr = fopen(“D:\\file.txt”,”w”);
fprintf(ptr, "%s", str);
```

We have already covered **file pointer**, opening/closing, reading/writing to a file, and the different modes of opening a file. In this tutorial, we will see a few more built-in functions related to files in C. By using these functions, we can read or write data from or to the files, character by character, or in the form of a full string.

fputc():

fputc() is used to write **characters** to the file. The C in the name of the function stands for character. The function takes two parameters as input. The first one is the single character that we want to input to the file. The second parameter is the pointer to the file. On successful implementation, it returns the character on to the screen. If it couldn't do so in case of any other issue, it would display an EOF exception. EOF stands for End of File. You will see a lot of this exception while working with files.

Syntax:

```
type fputc(‘character’,file_pointer);
```

Example:

```
Int fputc(‘a’,ptr);
```

fputs():

fputs() is used to write a null-terminated string to the file. The S in the name of the function stands for string.it also takes two parameters, the same as fputc(). One is the variable storing the string and the second one is the pointer to the file. A **null-terminated string** is a character string that can be terminated by a null character i.e., \0. You do not have to bother much about null-terminated strings as our computer automatically converts character strings to null-terminated strings.

Syntax:

```
fputs(“string”,file_pointer);
```

Example:

```
fputs("code with harry",ptr);
```

fgetc():

fgetc() works exactly the opposite of fputc(). It reads the character from the file. It reads only **one character at a time**. We can print it as many times as we want to get the next character and so on. Its syntax is straightforward, as we have to send the file pointer as a parameter. We can store the character into another character to display it onto the screen.

Syntax:

```
type = fgetc(file_pointer);
```

Example:

```
c = fgetc(ptr);
```

fgets():

fgets() is used to read a string from a file. It takes **three parameters** as input and stores them in a null terminated array. Now talking about the parameters, the first one is the storage array we want our string to store. The third one is the file pointer, and the second one is the count of variables we want to get from the files. For example, we want to take the first four characters from the string, so we will input the second parameter equals to 5. The purpose of doing that is that the null character will hold the fifth place.

Note: Blank space is also considered as a character.

Syntax:

```
int fgets(const char *s, int n, file_pointer);
```

Example:

```
fgets(str, 5, ptr);
```