

C File Handling

Objective : Students will learn about file handling in C using standard I/O functions

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 a 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. You can easily create text files using any simple text editors such as Gedit/vim/Emacs.

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

They take minimum effort to maintain, are easily readable, and provide the 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 a higher amount of data, are not readable easily, and provides better security than text files.

File Operations

In C, you can perform four major operations on files, 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 the program.

```
FILE *fptr;
```

Opening a file - for creation and edit

Opening a file is performed using the `fopen()` function defined in the `stdio.h` header file.

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

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

For example,

```
fopen("./newprogram.txt", "w");
```

```
fopen("./oldprogram.bin", "rb");
```

- Let's suppose the file `newprogram.txt` doesn't exist in the present working directory. 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 present working directory. 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

Mode	Meaning of Mode	During Inexistence of file
r	Open for reading.	If the file does not exist, <code>fopen()</code> returns NULL.
rb	Open for reading in binary mode.	If the file does not exist, <code>fopen()</code> 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. Data is added to the end of the file.	If the file does not exist, it will be created.
ab	Open for append in binary mode. Data is added to the end of the file.	If the file does not exist, it will be created.
r+	Open for both reading and writing.	If the file does not exist, <code>fopen()</code> returns NULL.
rb+	Open for both reading and writing in binary mode.	If the file does not exist, <code>fopen()</code> 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.

Mode	Meaning of Mode	During Inexistence of file
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 exist, it will be created.
ab+	Open for both reading and appending in binary mode.	If the file does not exist, 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 the `fclose()` function.

```
fclose(fp);
```

Here, `fp` is a file pointer associated with the 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_file.c : This program takes a number from the user and stores in the file `my_program.txt`. After you compile and run this program, you can see a text file `my_program.txt` created in present working directory. When you open the file, you can see the integer you entered.

reading_from_a_file.c : This program reads the integer present in the `my_program.txt` file and prints it onto the screen. If you successfully created the file by executing `writing_to_a_file.c`, you will get you the integer you entered.

Other functions like `fgetchar()`, `fputc()` etc. can be used in a 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 `fwrite()` function. The functions take four arguments:

1. address of data to be written in the disk
2. size of data to be written in the disk
3. number of such type of data
4. pointer to the file where you want to write.

```
fwrite(addressData, sizeData, numbersData, pointerToFile);
```

writing_a_binary_file.c : In this program, we create a new file `program.bin` in the present working directory.

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 the `fwrite()` function as above.

```
fread(addressData, sizeData, numbersData, pointerToFile);
```

reading_binary_file.c : 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`.

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
SEEK_SET	Starts the offset from the beginning of the file.
SEEK_END	Starts the offset from the end of the file.
SEEK_CUR	Starts the offset from the current location of the cursor in the file.

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