C File I/O and Binary File I/O

In this tutorial, you'll learn how to do file IO, text and binary, in C, using **fopen**, **fwrite**, and **fread**, **fprintf**, **fscanf**, **fgetc** and **fputc**.

FILE *

For C File I/O you need to use a FILE pointer, which will let the program keep track of the file being accessed. (You can think of it as the memory address of the file or the location of the file).

For example:
```
FILE *fp;
```

fopen

To open a file you need to use the fopen function, which returns a FILE pointer. Once you've opened a file, you can use the FILE pointer to let the compiler perform input and output functions on the file.
```
FILE *fopen(const char *filename, const char *mode);
```
In the filename, if you use a string literal as the argument, you need to remember to use double backslashes rather than a single backslash as you otherwise risk an escape character such as \t. Using double backslashes \\ escapes the \ key, so the string works as it is expected. Your users, of course, do not need to do this! It's just the way quoted strings are handled in C and C++.

fopen modes

The allowed modes for fopen are as follows:
```
r  - open for reading
w  - open for writing (file need not exist)
a  - open for appending (file need not exist)
r+ - open for reading and writing, start at beginning
w+ - open for reading and writing (overwrite file)
a+ - open for reading and writing (append if file exists)
```
Note that it's possible for fopen to fail even if your program is perfectly correct: you might try to open a file specified by the user, and that file might not exist (or it might be write-protected). In those cases, fopen will return 0, the NULL pointer.

Here's a simple example of using fopen:
```
FILE *fp;
fp=fopen("c:\\test.txt", "r");
```
This code will open test.txt for reading in text mode. To open a file in a binary mode you must add a b to the end of the mode string; for example, "rb" (for the reading and writing modes, you can add the b either after the plus sign - "r+b" - or before - "rb+")

fclose

When you're done working with a file, you should close it using the function
`int fclose(FILE *a_file);`
fclose returns zero if the file is closed successfully.

An example of fclose is
`fclose(fp);`

Reading and writing with fprintf, fscanf fputc, and fgetc

To work with text input and output, you use fprintf and fscanf, both of which are similar to their friends printf and scanf except that you must pass the FILE pointer as first argument. For example:
```
FILE *fp;
fp=fopen("c:\\test.txt", "w");
fprintf(fp, "Testing...\n");
```
It is also possible to read (or write) a single character at a time--this can be useful if you wish to perform character-by-character input (for instance, if you need to keep track of every piece of punctuation in a file it would make more sense to read in a single character than to read in a string at a time.) The fgetc function, which takes a file pointer, and returns an int, will let you read a single character from a file:
`int fgetc (FILE *fp);`
Notice that fgetc returns an int. What this actually means is that when it reads a normal character in the file, it will return a value suitable for storing in an unsigned char (basically, a number in the range 0 to 255). On the other hand, when you're at the very end of the file, you can't get a character value--in this case, fgetc will return "EOF", which is a constant that indicates that you've reached the end of the file. To see a full example using fgetc in practice, take a look at the example here.

The fputc function allows you to write a character at a time--you might find this useful if you wanted to copy a file character by character. It looks like this:
`int fputc( int c, FILE *fp );`
Note that the first argument should be in the range of an unsigned char so that it is a valid character. The second argument is the file to write to. On success, fputc will return the value c, and on failure, it will return EOF.

Binary file I/O - fread and fwrite

For binary File I/O you use fread and fwrite.

The declarations for each are similar:
```
size_t fread(void *ptr, size_t size_of_elements, size_t number_of_elements,
FILE *a_file);

size_t fwrite(const void *ptr, size_t size_of_elements, size_t
number_of_elements, FILE *a_file);
```
Both of these functions deal with blocks of memories - usually arrays. Because they accept pointers, you can also use these functions with other data structures; you can even write structs to a file or a read struct into memory.

Let's look at one function to see how the notation works.

fread takes four arguments. Don't be confused by the declaration of a void *ptr; void means that it is a pointer that can be used for any type variable. The first argument is the name of the array or the address of the structure you want to write to the file. The second argument is the size of each element of the array; it is in bytes. For example, if you have an array of characters, you would want to read it in one byte chunks, so size_of_elements is one. You can use the sizeof operator to get the size of the various datatypes; for example, if you have a variable int x; you can get the size of x with sizeof(x);. This usage works even for structs or arrays. E.g., if you have a variable of a struct type with the name a_struct, you can use sizeof(a_struct) to find out how much memory it is taking up.

e.g.,
```
sizeof(int);
```

The third argument is simply how many elements you want to read or write; for example, if you pass a 100 element array, you want to read no more than 100 elements, so you pass in 100.

The final argument is simply the file pointer we've been using. When fread is used, after being passed an array, fread will read from the file until it has filled the array, and it will return the number of elements actually read. If the file, for example, is only 30 bytes, but you try to read 100 bytes, it will return that it read 30 bytes. To check to ensure the end of file was reached, use the feof function, which accepts a FILE pointer and returns true if the end of the file has been reached.

fwrite is similar in usage, except instead of reading into the memory you write from memory into a file.

For example,
```
FILE *fp;
fp=fopen("c:\\test.bin", "wb");
char x[10]="ABCDEFGHIJ";
fwrite(x, sizeof(x[0]), sizeof(x)/sizeof(x[0]), fp);
```

A file represents a sequence of bytes, does not matter if it is a text file or 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 important calls for the 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. Following is the prototype of this function call:

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

Here, **filename** is 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 reading and writing both.
w+	Opens a text file for reading and writing both. It first truncate the file to zero length if it exists otherwise create the file if it does not exist.
a+	Opens a text file for reading and writing both. 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 below mentioned access modes instead of the above mentioned:

```
"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 provide by C standard library to read and write a file character by character or in the form of a fixed length string. Let us see few of the in the next section.

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 next section.

Reading a File

Following 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 functions allow you 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 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 the first space character encounters.

```
#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 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 detail about what happened here. First **fscanf()** method read just **This** because after that it encountered a space, second call is for **fgets()** which read the remaining line till it encountered end of line. Finally last call **fgets()** read second line completely.

Binary I/O Functions

There are following two functions, which can be used for binary input and output:

```
size_t fread(void *ptr, size_t size_of_elements,
             size_t number_of_elements, FILE *a_file);

size_t fwrite(const void *ptr, size_t size_of_elements,
             size_t number_of_elements, FILE *a_file);
```

Both of these functions should be used to read or write blocks of memories - usually arrays or structures.

http://www.tutorialspoint.com/cprogramming/c_file_io.htm