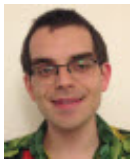
C File I/O and Binary File I/O



By Alex Allain

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);
```

Quiz yourself
Previous: Strings
Next: Typecasting
Back to C Tutorial Index

Related articles

More on working with files in C

C++ file IO