1 File I/O: Introduction

So far, we have written programs, but all of our input has been from standard input (normally the keyboard) and all our output has been to the standard output (normally the screen). Now it is time for us to learn how to read and write to files. In this way, we can do more interesting things such as accessing disk files, and storing our output.

2 Streams

The file input and output operations in C are done through logical constructs that are called *streams*. A stream is a unified logical description of any computer resource than can input and/or output a series of bytes. In the most common form, a stream will be an interface to a disk file, but the "file" could be the screen, the keyboard and the like just as well.

There are two kinds of streams: Binary and text. Binary streams are raw streams that can be used with any kind of data. On the other hand, some character conversions may happen in a text stream. The most notable conversion happens on DOS/Windows text files, where a newline character is converted to a carriage-return/linefeed pair on output.

The C language also keeps a reference, called the *current location* with any open stream. It stores basically where the next input or output operation access in the file. So, in a file 100 bytes long, if the current location is 50, the next read operation will return the 50th byte, and advance the location by one.

3 The `FILE *` Type

There is no way you can create a file object on your own. You have to call a library function and ask it to open a file for you. The function will return you a pointer to the newly opened file. To store that pointer, you need to declare a pointer that can point to a file object. This type is `FILE` which is defined in `stdio.h`. Thus, if you want to call your pointer `myfile` you can declare it as below:

```
FILE *myfile;
```

4 Opening a File

The function you will use to open a file is `fopen()`, the prototype of which is given below:

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

The first argument is the pathname of the file you wish to open. If you specify a bare filename, it will be assumed to be in the current working directory. Or, you can specify a full pathname.

The `mode` argument specifies in what "mode" you wish to open the file. There are 12 related possibilities:

Mode	Meaning
r	Open a text file for reading
w	Create a text file for writing
a	Append to a text file
rb	Open a binary file for reading
wb	Create a binary file for writing
ab	Append to a binary file
r+	Open a text file for read and write
w+	Create a text file for read and write
a+	Append or create a text file for read and write
r+b	Open a binary file for read and write
w+b	Create a binary file for read and write
a+b	Append or create a binary file for read and write

For all "w" modes, the file is created if it does not exist, and is truncated to zero length otherwise. For all "a" modes, the file will be created if it does not exist and the current location is set to the end of the file.

This function returns a pointer to the opened file upon success. If the call fails, it returns a `NULL` pointer.

5 Closing a File

Once you are done with a file, you should close it. This is done using the following function:

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

6 Checking the Status of a File

First, you can check if the file has reached the end-of-file. This is done with the following function:

```
int feof(FILE *stream);
```

The function `feof()` tests the end-of-file indicator for the stream pointed to by `stream`, returning non-zero if it is set.

Second, you can check whether the stream has encountered any errors. This is done with the following function:

```
int ferror(FILE *stream);
```

The function `ferror` tests the error indicator for the stream pointed to by `stream`, returning non-zero if it is set.

7 Reading Single Characters

Once a file is opened, you can use the following function to read single characters from it:

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

The function `fgetc()` reads the next character from stream and returns it as an `unsigned char` cast to an `int`, or `EOF` on end of file or error.

8 Writing Single Characters

The following function can be used to write a single character to a file:

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

The function `fputc()` writes the character `c`, cast to an `unsigned char`, to `stream`. It returns the character written as an `unsigned char` cast to an `int` or `EOF` on error.

9 Reading and Writing Text

9.1 `fputs()` and `fgets()`

These two functions are used to read and write strings.

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

The function `fputs()` writes the string `s` to `stream`.

```
char *fgets(char *s, int size, FILE *stream);
```

The function `fgets()` reads in at most one less than `size` characters from `stream` and stores them into the buffer pointed to by `s`. Reading stops after an `EOF` or a newline. If a newline is read, it is stored into the buffer. A `'\0'` is stored after the last character in the buffer.

9.2 `fprintf()` and `fscanf()`

These two functions are identical to their counterparts that operate on the console and keyboard (`printf()` and `scanf()`) and differ only in that they operate on files instead. They are as follows:

```
int fprintf(FILE *stream, const char *format, ...);
int fscanf( FILE *stream, const char *format, ...);
```

10 Reading and Writing Binary Data

It is possible to write binary data to a file, and read binary data from a file as well. There are two functions that accomplish this:

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

The function `fread` reads `nmemb` elements of data, each `size` bytes long, from the stream pointed to by `stream`, storing them at the location given by `ptr`

The function `fwrite` writes `nmemb` elements of data, each `size` bytes long, to the stream pointed to by `stream`, obtaining them from the location given by `ptr`.

11 Moving Around a File

It is also possible to move the "current location" pointer, or the "mark" of the stream. There are three essential functions that can be used for this purpose:

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

The `fseek` function sets the file position indicator for the stream pointed to by `stream`. The new position, measured in bytes, is obtained by adding `offset` bytes to the position specified by `whence`. If whence is set to `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`, the offset is relative to the start of the file, the current position indicator, or end-of-file, respectively.

The return value is 0 upon successful completion, and -1 if an error occurred.

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

The `rewind` function sets the file position indicator for the stream pointed to by `stream` to the beginning of the file. It is equivalent to:

```
fseek(stream, 0L, SEEK_SET)
```

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

The ftell function obtains the current value of the file position indicator for the stream pointed to by stream, and returns it as its return value.