Chapter 12: Files and I/O

Notes

- Probably should put this explanation of files in part 1.
- Escape characters.
- Detecting failure with 'fail'.

Introduction

Files are blobs, one or more bytes, of permanent storage in a computer. They are usually stored on some form of disk, albeit hard or floppy, and labeled with *attributes* that can be specific to the file system[1] and/or operating system they exist for. The most common attribute is the *filename*, or name for the file.

Files are stored in directories, or folders, which may be nested. That is, a directory can have *child* directories, which can then have more child directories, and so on and so forth. The hierarchy specifying all the folders that must be accessed in order to access the file is known as the *file path*. Sometimes this file path will also include a device name. A filename name including the file path is known as a *full name*.

Sometimes a filename can be identified using a partial name, also known as a relative path. That is, the path specified in the filename does not start from the top-level and is relative to the *current directory*. The current directory is a "default directory" from which relative paths and short filenames (ones without paths) are evaluated.

The process of tell a device which file you wish use is known as *opening a file*, even if you are going to create one that doesn't exist. If you have an actual, physical file drawer then to access one of the files you must pull it out and *open* it. To add a new file to the drawer, you'll get a new folder, but you'll still have to open that to put stuff in it. In computers, you do not "pull out" the file, it remains in the same place, but you must "open" it in order to use it. When opening a file you must also tell the device, and operating system, what you plan to do with the file: look at it, change it, add to it, or all of these. Once you have opened the file, you are allowed only to do the things you initially opened the file for.

Files are very similar to the character strings or character arrays. Since the 'char' data type is equivalent to a byte, files are equivalent to a string of bytes or 'char's. The difference is that files can contain any value, including zero, in a single character. You must determine the size of files differently than the length of strings. But file data, the bytes that a file is made of, exists on another device and cannot be accessed directly. In

[1] A disk is just a device that stores a certain number of bytes, so a file system is a specification of how to organize the bytes meaningfully; it is how files are stored. A common file system is FAT or FAT32.

order for you to use the data of a file, it must be copied from the device it exists on to local memory that your program has access to. This is known as reading from a file. Changing data by replacing or adding is known as writing to a file. When you are adding to the end, it is known as appending to the file. But appending is still a write operation.

When finished with a file, you must *close* it. You cannot do something other than you are allowed, so sometimes you'll close a file, and re-open it with different intentions. Say you created a new file by writing to it and now you want to read back that data. You'd have to close the file you were writing to and re-open it with the intention to read from it.

This whole process is hidden from you and made simple by *file streaming* objects.

File Streaming

File streaming objects are instances of classes that you create yourself in order to access a file. These objects do not already exist for you, like 'cout' and 'cin', which means you have to name them. The class you use for file streaming, reading and writing to files, is 'fstream':

```
fstream file;
```

To open a file with an 'fstream' object, use the 'open' member function. This function has no return value and takes two necessary parameters: the filename and *access mask*:

```
void fstream::open(const char *filename, int access);
```

The access mask is your specified intentions for the file: read, write, create, append, etc. You specify it by members of an enumeration that exist in the 'ios' class using the scope operator. The two members of this we'll use for now are 'in' and 'out' which, using the scope operator, are written as 'ios::in' and 'ios::out' respectively. So, to open a file for reading:

```
file.open("somefile", ios::in);
```

Not so difficult is it? The 'ios::in' member signifies the intent to only *read* from the file, so you can't do any writing to a file opened with this designated access. To *write* to a file, use 'ios::out':

```
file.open("somefile", ios::out);
```

This will not only open the file for writing, it will clear the file if it already exists and create it if it doesn't. From that point you can start merrily pumping data into the file. When you're finished with a file, don't forget to *close* it with the 'close' member function:

```
void fstream::close(void)
```

Since it returns no value and takes no parameters, so it's a piece of cake to use:

```
file.close();
```

Besides 'fstream' there are also two other classes you can use which are *completely* equivalent to 'fstream'. The only difference is that you don't have to specify an access mask using them, because they have parameter defaults. The first class is 'ifstream' which, when 'open' is called without an access mask, opens files for reading. The second is 'ofstream' which opens files for writing:

```
ifstream inpfile;
inpfile.open("somefile");
inpfile.close();

ofstream outfile;
outfile.open("somefile");
outfile.close();
```

Author's Preference: Because they don't really provide any additional usefulness, I typically avoid using 'ifstream' or 'ofstream'.

Writing to Files

When you open a file to write, it is wiped clean as if it was newly created. And if you open a file for writing that didn't previously exist, it is created. At this point you can begin sending bytes to the files to be added to it. Since it is blank, the data is added at the very beginning.

Writing can be done a single byte at a time using 'put' or blocks at a time using 'write'. The 'put' member function accepts a single byte parameter. There are three versions of 'put', each which takes a different type of 'char': 'char', 'signed char', or 'unsigned char'. These all equate to writing the same byte, but it prevents you from having to cast the value to a specific type. You can also, as normal, pass in numeric or character literals:

```
char c = 'a';
signed char sc = 'b';
unsigned char uc = 'B';

file.put(c);
file.put(sc);
file.put(uc);
file.put('C');
file.put(95);
```

The 'put' member function could be used to write entire strings by itself. Consider:

```
char str[] = "Hello World";
char *pt = str;

while (*pt)
{
    file.put(*pt);
    pt++;
}
```

The above code would write out all of the characters of "Hello World" to the file, one character at a time. This is fairly time consuming though and if you were to have to put this in several places in your program it would get old. The alternative to writing a single character at a time is to write whole blocks of bytes using 'write'.

The 'write' member function has three different versions like 'put', but the parameters are all 'const' because the function takes a 'char' pointer to a block of data:

```
fstream::write(const char *data, int length);
fstream::write(const signed char *data, int length);
fstream::write(const unsigned char *data, int length);
```

I'm deliberately avoiding the return values of these functions for now, so just don't worry about it. The 'write' member function can be passed any old string or array as long as you specify how many bytes to write with the second parameter, 'length', and as long as you use a pointer to a 'char' type:

```
char str[] = "Goodbye World";
char *p = str;
file.write("Hello World", 11);
file.write(str, strlen(str));
file.write(p, strlen(p));
```

Because all pointers contain the same thing, a memory address, regardless of their "type", you can set a 'char' pointer to the address of any variable or data block. You just have to cast the address of the data block to a 'char' pointer type:

```
int x = 5;
char *p = reinterpret_cast<char*>(&x);
file.write(p, sizeof(x));
```

The above code would write out the actual binary data that represents 'x'. It writes the number of bytes that 'x' uses for memory along with whatever its memory contains. I'll cover this more later, but I'm going to avoid using anything other than characters for now.

Blarg.

Reading from Files

Blarg.

Appending to Files

Blarg.

Text versus Binary

Blarg.

old stuff …

Streaming

The enigmatic ‘cout’ and ‘cin’ are far from magical beasts. As you might have guessed, they are objects, instances of some class. File input/output is closely tied to these objects in a concept known as streaming.

A stream is, in concept, a flow of data that it possibly unending. Data is accessed by extracting it from input streams and inserting it into output streams. Some streams are bi-directional which means you can insert data into them as well as extract it. It sounds pretty abstract and it is, it is only a concept. The insertion (<<) and extraction operators (>>) work with streaming by inserting or extracting data.

Inserting data into “console output”, the ‘cout’ object, causes it to appear onto the screen. Likewise, extracting data from “console input”, the ‘cin’ object, draws it from the keyboard. But these are merely sources and destinations for the streams themselves. File i/o works by inserting data into a stream that pours into a file or extracting data from a stream that is coming from a file. Thus, file writing and reading is achieved.