The Techie's Guide to C Programming Part 11

This month we'll continue to delve into the mysteries of high level file access and, in the fullness of time, to wholly appreciate how weird and powerful it can be. Time has a lot of fullness.

STEVE RIMMER

Last month we got into the basics of file handling under C. It might have seemed a bit nasty then, but high level file access is actually very easy to use once you get used to it. Most of the difficulties that programmers new to C have with it stems from their prior involvement with BASIC's file routines.

BASIC conditions one to believe that nothing associated with disk files can really be easy.

High level file access under C has a number of very powerful features, as we'll see. It can make accessing disk files no more complex than accessing your keyboard and screen.

In fact, as far as C is concerned, all these functions are basically the same.

Illusion and Reality

In an ideal computer, everything external to the processor would either be a data source or a data sink, or both. The screen would be a data sink — data goes to it. The keyboard would be a data source, that is, data occasionally comes from it. Garbage occasionally comes from it too — it's a bit hardware dependant in this respect. Consider well that you are the hardware it's dependant on.

Disk files can be both data sources and data sinks.

Under C, we have the option of treating data in this way if we want to by using "streamed" data handling. The standard file input and output channels of DOS have been designed to fit nicely into this model. Not surprisingly, most of the applications which Microsoft, the creators of MS-DOS, write, are done in C.

Last month we looked at file handling using file pointers. This month we're going to expand on that and see how the concepts of file handling can work with most forms of data which moves in and out of the computer.

When a C language program starts up, it automatically opens five files. These are given names, and are available to any programmer who chooses to use them. This is how they're defined under C.

```
stdin
stdout
stderr
stdaux
stdprn
```

Now, the use of these things might not be completely obvious. The problem with them is that they're so elegantly simple that they defy immediate understanding.

It's so zen-like as to make your karma green.

Consider the following program. This will read in a disk file called WOMBAT.DOC and display it on the screen as it's read.

```
FILE *fp;

if((fp=fopen("WOMBAT.DOC","ra")) !=
NULL){
while(putch(fgetc(fp)) != EOF);
fclose(fp);
} else puts("WOMBAT.DOC ain't there");
```

The principal behind this program should be fairly clear if the cat didn't eat your copy of last month's ETT before you read it.

Each of the five mystery words previously mentioned acts just like the file pointer in the above example, except that each one is tied to a particular device, rather than to a disk file. Consider the following program.

```
if((fp=fopen("WOMBAT.DOC","ra")) !=
NULL){
while(fputc(fgetc(fp),stdout) != EOF);
fclose(fp);
} else puts("WOMBAT.DOC ain't there");
```

What this is really doing is to copy the file WOMBAT.DOC into the mystery file pointer called *stdout*. For reasons too warped for mere mortals to get their heads around, everything written to the file pointer *stdout* will appear on the screen.

Here's a variation on this program.

```
if((fp=fopen("WOMBAT.DOC","ra")) !=
NULL){
while(fputc(fgetc(fp),stdprn) != EOF);
fclose(fp);
} else puts("WOMBAT.DOC ain't there");
```

The only thing that's changed here is the name of the destination of the data read in from WOMBAT.DOC. Instead of going to *stdout*, the screen, it's going to *stdprn*, which is usually connected to the printer port LPT1. This little program, then, will print the contents of WOMBAT.DOC.

Consider this third variation.

```
FILE *fp,*destination;

puts("Hit P to print WOMBAT.DOC, any
other");
puts("key to view it");
if(getch()=='P') destination=stdprn;
else destination=stdout;

if((fp=fopen("WOMBAT.DOC","ra"))!=
NULL){
while(fputc(fgetc(fp),destination)!=EOF);
fclose(fp);
} else puts("WOMBAT.DOC ain't there");
```

In this example, we've created a file pointer and sent the data from the file to it. The file pointer can be the printer or the screen, depending on whether we want to view or print the file. Alternately, the file pointer could really point to an open file. If you have a program which prints to the printer through *stdprn*, this is an easy way to implement a print to disk function — just assign the file pointer the value of *stdprn* instead of that of an open disk file with *fopen*.

Using this model for data handling, it might be a bit easier to understand how to deal with streamed file input and output. Allowing that *fp* is a file pointer to an open file and that *stdout* is the pseudo-pointer for the screen, you will observe that writing a character to the screen is handled like this:

```
fputc('A',stdout);
```

while writing a character to the file is handled like this:

```
fputc('A',fp);
```

The screen and the file behave more or less the same way. In both cases, the newly written byte will appear immediately after the most recently written byte. This will be at the cursor position on the screen and at the end of the file as it has been written to date for the disk file. If you think of a disk file as being simply a series of bytes, as the screen is, you should have no problem understanding how C deals with files.

More File Modes

Files can do things which the screen cannot, and the analogy does start to fall apart after a while. For example, You can "seek" in a file. The seeking mechanism under C allows you to define where in the file the next byte, or bytes, will be written to or read from. Allowing that we have a file of one kilobyte in length. If we open it to read and begin to read in bytes with the *fgetc* function, the bytes will be drawn from the beginning of the file. If we wanted to read from the five hundredth byte on, we would have to read in and throw away a lot of data, which is inefficient and very slow.

The *fseek* function allows us to position the file position pointer, that is, where the bytes will be read from, anywhere in the file at any time. If the opened file has a pointer *fp*, we can position its file position pointer to the five hundredth byte like this:

```
fseek(fp,500L,SEEK_SET);
```

The first argument to this function is obviously the file pointer. The second one is the number of bytes in from the head of the file — note, however, that we must make it a long value. Since files can be bigger than sixty-four kilobytes, anything which specifies a location within a file must do so in long numbers. Finally, the last argument must tell *fseek* whether 500L reflects the position relative to the start of the file, the end of the file or the current file position. Three constants are defined to reflect this — SEEK_SET tells *fseek* to position the file's position pointer relative to the start of the file. We'll deal with the other modes at a later time.

If we read a byte from the file with *fgetc* after executing the above command, we should see the five hundredth byte returned. If we write to the file, assuming that the file has been opened for both reading and writing, the first byte we write will overwrite byte five hundred.

If you have a file which has been opened for both reading and writing you must do an *fseek* if you switch between these two functions.

There's another function which works nicely with *fseek*. The *ftell* function returns a long integer which specifies how far into the file next byte will be read from or written to. In other words, it returns the file position pointer. You can use this for all sorts of things. For example, this bit of code will return the size of the file which has been opened with the file pointer *fp*.

```
fseek(fp,0L,SEEK_END);
size=ftell(fp);
```

The variable *size* must be a long integer. The constant SEEK_END tells fseek to seek to the end of the file, or to as many bytes from the end as are specified by the second argument to *fseek*.

Having seeked to the end of the file, of course, the file position pointer might not be where you want it to be if you had plans to read data from the file. You could, of course, use *fseek* to return to the start of the file, but C also provides a shorthand version of this function. You can just

```
rewind(fp);
```

This returns the file position pointer of *fp* to the beginning of the file.

Cooking with Rabbits

When you open a file using *fopen*, the second argument specifies the mode, as we discussed last month. This can be either binary mode, which means that bytes are read as bytes, or "cooked" mode, in which case they mostly are. The cooked mode is used when you're want to read ASCII files as ASCII.

When C wants to represent the end of a line, it uses the '\n' characters, that is, the "newline" character. When you are using the streamed file functions with a file opened in the cooked mode, any instances of carriage return line feed pairs will be converted to the newline character in the way in, and newlines will be converted to carriage returns and line feeds on the way out. In the binary mode, no conversion is performed.

In order to open a file for reading in cooked mode, you would do this:

```
fopen("FROGFILE.DOC","ra");
```

To open a file in binary mode you would do this:

```
fopen("FROGFILE.BIN","rb");
```

There are several consequences of using cooked mode. For one thing, the numbers returned by *ftell* may not really reflect the file as you see it from your program, as newline characters are one byte which a carriage return line feed pair is two characters long. Likewise, using *fseek* might not place the file position pointer where you think. Cooked mode is usually used with text files, wherein these two functions don't apply for the most part.

Despite the added need for caution in using cooked mode, it can make using text files with C a great deal easier. We'll learn a bit more about how it works in the next installment of this series.