FEATURE

Techie's Guide to C Programming Part 10

File operations under C look extremely complex. This is exacerbated by there being at least three ways to do everything. In this installment we'll look at sorting out some of the confusion.

STEVE RIMMER

If you're new to C programming... and especially if you've tried similar tasks in BASIC... you'll know that writing programs which involve disk file operations is a bit tricky. Actually, the disk file stuff is dead easy... it's just the way that most programming languages make you access them that can get a bit hairy.

Unfortunately, the disk file operations available to a C programmer are pretty numerous, and it's by no means obvious which functions are intended to be used in which applications. If you've perused your compiler's reference manual, you've probably come away with a head full of file pointers, file numbers, long offsets and relative seeking.

It's almost enough to send one back to BASIC. Almost.

The first thing that's important to realize about file handling in C is that you can ignore at least two thirds of the available facilities offered by your compiler. For reasons we'll get to, there's a lot of redundancy in there. If we start with the remaining file operations, things will start to work a bit more seamlessly.

Handling the Handles

The profusion of file services under C stems from two causes... or possibly three. The first one is historical. Early microcomputer C compilers were not always able to implement higher level file handling, even though the definition of C called for it. As such, they provided oftentimes pretty crude low level, block oriented file functions. Later compilers, wishing to remain compatible with their forebears, maintained some semblance of these functions along with the newer, more useful functions. As such, multiple levels of file functions came to be.

The more prominent cause, however, has to do with the way DOS on the PC likes to handle files. In a sense, it's quite powerful at this, especially in relation to earlier operating systems. C provides us with varying levels of access to this power, depending on what we're up to at the moment.

The simplest form of file access is *sector* access. One saw this sort of thing happen a lot under CP/M, the eight bit precursor to MS-DOS. A disk file is really made up of what are called "allocation blocks", or, in a still simpler sense, disk sectors. As such, using sector based file access, one is able to read or write specific sectors of a file. A sector was one hundred and twenty eight bytes under CP/M... it's of varying size under MS-DOS, but no DOS based C compiler with any pride at all would ever expect its users to work with files this way.

Using sector access, if you wanted to read a specific byte from a file, you would figure out which sector that byte resided in like this

```
sector_number=byte_number/sector_size;
```

and then read that sector into a buffer. Next, you'd figure out how many bytes into the sector buffer the one you wanted was.

```
offset=byte_number % sector_size;
```

The byte you wanted would be

```
the_byte=sector_buffer[offset];
```

As I said, you won't have to do this on a PC.

What C does offer us on a PC... somewhat akin to this but with rather more flexibility... is *block* access. In this access procedure, you really get to deal with DOS as DOS deals with its files. It allows you to read or write from any point in a file, and to deal with blocks of any size. This is often referred to as ''low level'' file access.

Low level file access is handled by the following functions under C.

open to establish a link to the file in question
read to read in blocks of data
write to write out blocks of data
lseek to position the file pointer
close to close the file

Some of this might not be completely clear just yet... don't worry about it for the moment.

Handling files in this way is great if you want to read and write large blocks of data, because low level file access is fast. For example, this function will save the entire text screen of a PC into a file called SCREEN.TXT. The file will be four thousand bytes long.

```
save_screen()
{
char *p;
int handle;

if((hand-
le=open("SCREEN.TXT",O_CREAT))
!=-1)){
p=MK_FP(0xb800,0);
write(handle,p,4000);
close(handle);
}else puts("Error creating file");
}
```

In this example, the *MK_FP* function is used to synthesize a pointer which points to the base of the screen buffer... don't worry about how this works for the moment. We'll assume that the video card in question is a CGA card and that the text screen happens to be on page zero.

The *open* function returns a number called a *file handle*, or —1 if the file couldn't be created for some reason. The argument O_CREAT tells *open* to create the file in question. In fact, these arguments are quite complex... we'll look at them properly later on in this series.

A file handle is actually a number which serves as an index into a series of file pointers. We'll see what a file pointer is shortly.

The *write* function writes the contents of the screen buffer to the file whose handle is passed to it. The *close* function tidies up the file and *releases* the handle, that is, it severs the link with the file and makes the handle available for future use.

We're actually going to get into this level of file access in greater detail in a coming issue, because it's not all that useful for most of the things you're likely to want to write immediately. Its drawback is that it's great for dealing with large blocks, but quite sloppy if you want to read or write a single byte. You can do it... just read in a block that's only a byte long... but the efficiency of low level file access kind of up and vanishes when you do this.

Streams and Rivers

The other options for file access under C is ''high level'' file handling, which is the really powerful way to handle files. This allows you to deal with them in the way that most programs want to, that is, on a single byte level. However, high level file access is flexible... you can use both byte by byte access and block access in the same file.

The functions which do high level file access all have the letter ''f'' in front of them. The equivalents of the low level functions, above, are as follows.

fopen to establish a link to the file in question
fread to read in blocks of data
fwrite to write out blocks of data
fseek to position the file pointer
fclose to close the file

There are also two others we'll look at here.

fputc to write a single byte to the file in question
fgetc to get a single byte from the file in question

In order to deal with a file, we have to start by opening it. This does several things. If the file is to be read, opening it makes sure that it actually exists. If it's to be written to, opening it creates the file. If it's to be appended... its current contents read and possibly altered... opening it makes sure that you're allowed to do this, that is, that the file is not read only.

Opening a file using high level access also establishes a pointer to a structure of the type FILE. The structure itself is put somewhere by the file management routines of C... you will never create FILE variables in your programs, just pointers to them. This is what's actually in a FILE variable.

```
typedef struct {
short    level; /* fill/empty level of buffer */
unsigned   flags; /* File status flags */
char     fd; /* File descriptor */
unsigned char hold; /* Ungetc char if no
buffer */
short    bsize; /* Buffer size */
unsigned char *buffer; /* Data transfer
buffer */
unsigned char *curp; /* Current active
pointer */
unsigned   istemp; /* Temporary file in-
dicator */
short    token; /* Used for validity check-
ing */
} FILE;
```

None of this need mean a thing to you. Not only will you never have to create one of these variable... you'll also never have to use any of the members that a FILE pointer points to. We simply pass these things around, oblivious to what they really do. The high level file routines do all the work.

Here is an example of the use of the *fopen* function. In this case, we are going to open a text file to be read.

```
FILE *fp;

fp=fopen("WOMBAT.TXT","ra");
if(fp==NULL) puts("File not found");
```

Actually, the correct... convoluted... way of writing this would be

```
if((fp=fopen("WOMBAT.TXT","ra"))!=
NULL){
/* do something */
}else puts("File not found");
```

Both versions do the same thing.

The *fopen* function returns a pointer to a FILE structure which C has created behind our backs. If the pointer points to NULL... location zero... the file didn't open for some reason. When *fopen* successfully opens a file, the FILE variable will have its fields filled in by *fopen*, although as we saw, we never deal with them and needn't care what gets put in them.

The first argument to *fopen* is the name of the file. This could be a complete DOS path if we wanted it to be, as in

```
fp=fopen("D:\TEXTFILE\WOM-
BAT.TXT","ra");
```

The second argument is a bit more involved. It tells *fopen* what the file mode will be.

The first character of the argument can be ''r'' if the file is to be opened for reading. In this case, any attempt to write

to the file would fail. It could be "w", in which case you could write to the file, but not read from it. If *fopen* is asked to open a file for writing, it destroys the existing contents of the file if there's already a file with the name in question on your disk. Finally, the first character could be "a", for appending, in which case you could both read from and write to the file.

The second character can either be "a" or "b". The "b" option is called *binary mode*, and it means that data moves in and out of the file unchanged. The "a" option is often referred to as *cooked mode*. It's especially designed for text applications. It means that carriage returns and line feeds are handled in a way which is convenient and easy to work with under C.

There are more options available for *fopen*, but we'll look at them a little later on.

Having established a pointer to a file structure with *fopen*, we can proceed to use the file. This code would read in the contents of a text file and display them on the screen.

```
while((putchar(fgetc(fp))!=EOF);
```

You'll want to look at this carefully for a moment. What it's really saying is this.

```
int c=0;

while(c!=EOF){
c=fgetc(fp);
putchar(c);
}
```

Left to its own devices, the *fgetc* function simply keeps returning bytes from the opened file until the file has all been read, at which time it returns the constant EOF.

It's important to understand that *fgetc* returns an *int*, even though it's really only getting *chars* from the file. The upper byte of the *int* will usually be empty. However, when the end of the file is reached, the *int* will contain —1, that is, 0xffff. This is how we differentiate between an EOF and a legitimate byte which contains 0xff.

This can cause a lot of problems if you aren't careful. For example, you might look at the above bit of code without thinking and decide that the variable "c" really only needs to be a *char*. If you do this, *c* will never equal EOF, since EOF is a sixteen bit value and *c* can only hold eight, being a *char*. As such, this code will never know when it's reached the end of the file, and it will loop forever.

When we're done with a file, we must close it, as before. This is done by saying

```
fclose(fp);
```

Closing a file which has been opened for high level file access does a number of things. It frees up the FILE structure for future use. It also *flushes* DOS's internal file buffers.

Flushing a file buffer really only matters if you've been writing to the file in question. However, it illustrates how high level file access really works. If you write one byte to a file with *fputc*, your program does not send that byte directly to the disk. It would be extremely inefficient to have to wake up the disk drive, seek around on the disk for a while, find the appropriate sector, read it in, place the byte in question and write it back out every time you call *fputc*.

Instead, the byte is added to a buffer. When the buffer gets full, a whole block of data is written to the disk. The buffer is then emptied, and subsequent calls to *fputc* continue to flow into the buffer.

If you close the file, any unwritten bytes in the buffer are written out to the disk. If you fail to do this, everything since the last automatic block write will be lost.

In a large program, it's important to remember to close your files when you're done with them. This flushes the file buffers in question. It also frees up FILE structures. If you try to open too many files, *fopen* will start returning NULL pointers, even though the files you've asked to be opened are valid and should present no problems.

The Circular File

We'll be continuing to look at file access under C next month. However, you might want to have a look at this practical application of the file functions we've seen today. This program converts WordStar files to plain text files using high level file functions. Actually, the only difference between a WordStar file and a text file is that the former has some of its characters stored with their high bits set. If we AND all the characters in the file with 0x7f... recall the discussion of bit manipulation from last month... the result will be clean text.

```
main(argc,argv)
int argc;
char *argv[];
{
FILE *source,*dest;
int c;

if((source=fopen(argv[1],"rb"))!=NULL)
{
if((dest=fopen(argv[2],"wb"))!=NULL){
while((c=fgetc(source))!=EOF)
fputc(c&0x7f);
fclose(dest);
}else printf("Can't open %s as destination\n",argv[2]);
fclose(source);
}else printf("Can't open %s as source\n",argv[1]);
}
```

This program uses command line arguments, which we haven't formally looked at yet. However, you can probably see how they work. Assuming that it was called UNWS.C... leaving you with UNWS.EXE after it is compiled... you would convert WOMBAT.WS into WOMBAT.TXT like this.

```
UNWS WOMBAT.WS WOMBAT.TXT
```

In practice, we could make this into a better WordStar converter with a few more lines of code, but this serves to illustrate just how flexible high level file access can be.

Next month we'll see some of the other things it can balance on its nose. ■