FEATURE

Techie's Guide to C Programming Part 12

This month we're going to find some practical uses for all the file access facilities we've been investigating in the previous two installments of this series. We're going to see how a database manager operates.

STEVE RIMMER

The better you understand disk file access, the less mysterious a lot of programs will probably become for you. Several broad classes of applications software rely on forms of file manipulation, and many others seem to do the impossible because their authors thought of clever ways to make files work.

The word processor I'm using to write this article, for example, will edit a file which is larger than the memory in the computer it's running on. It does this by using what has come to be called "virtual memory". It keeps the current part of the file in a small memory buffer and "spills" the rest of it on and off the disk as is required.

It may seem difficult to think of memory and disk space as being similar quantities, but in many respects they are. Using the file management facilities we've discussed thus far, it's possible to seek around in a file in the same way that we might use pointers to access areas of memory. Of course, moving data in and out of a file is a lot slower than equivalent memory operations... software which uses virtual memory techniques must do so with some forethought, lest it slow to a crawl every time it goes to alter its data.

A "stock" PC compatible computer can only address 640 kilobytes of useful system memory. The rest of its one megabyte address space is tied up with various memory mapped oddities like the screen buffer and the system BIOS. When you load DOS and a sizeable application program into that available memory, you might well find that all you have is a few hundred kilobytes left. If the application program in question deals with large amounts of data, it will probably find itself a bit cramped.

Loss of Memory

The picture in Figure 1 is a black and white version of a public domain "GIF" image. GIF images are full colour computer graphics which can be displayed on a VGA monitor. This one is 800 by 600 pixels across. Each pixel requires one byte. If you whip out your pocket calculator you'll discover that this picture requires 480,000 bytes of memory to store it.

A program which was confronted with the task of doing something with this picture might be faced with a problem. Unless the program was very tiny indeed... and hence didn't do very much... it would tie up enough system memory to

make allocating a buffer big enough to store the picture impossible.

The designer of the program in question would thus have several options. One of them would be to use extended or expanded memory, preferably the former, as it's faster. Extended memory allows users of AT and 386 computers to put up to 16 megabytes of additional memory in their computers, memory which is useless for running programs in, but is good for storing data. This is the sort of applications which extended memory is ideal for.

Unfortunately, in writing commercial software which relies on extended memory, one is immediately making one's software inaccessible to users of straight PC compatibles... which can't support extended memory... as well as to AT and 386 users who don't happen to have any extended memory installed in their machines.

There is a second option. Rather than writing the decoded GIF image to memory... of which there isn't enough... we could write it to a big disk file. If the program in question wanted to print the picture to a laser printer, for example, it could then retrieve the lines of the picture one at a time, making the memory requirements for the process pretty tame.

The process for locating the lines would be simple, and you can probably see how it's going to work if you recall last month's discussion of files and seeking. Allowing that the picture is *x* bytes wide and that we want to read line *n* into a buffer, *b*, we would do this. We'll assume that *fp is a handle to the file with the image data in it.*

```
fseek(fp,(long)x*(long)n,SEEK_SET);
fread(b,1,x,fp);
```

The *fread* function is used to read raw data into a buffer.

The reason for casting both the size variables to *long* will be discussed in a moment.

With this arrangement, the only memory requirements of the application software in question would be a buffer six hundred bytes long to hold one line of the image at a time.

This approach has two principal drawbacks. The first is that it assumes that its users will have about half a megabyte of disk space available to hold the big temporary image file. In reality, it assumes there will be this much hard drive space on hand... virtual memory is too slow to think about on floppies. The other drawback is that even with a hard drive this will be a lot slower than using memory would have been.

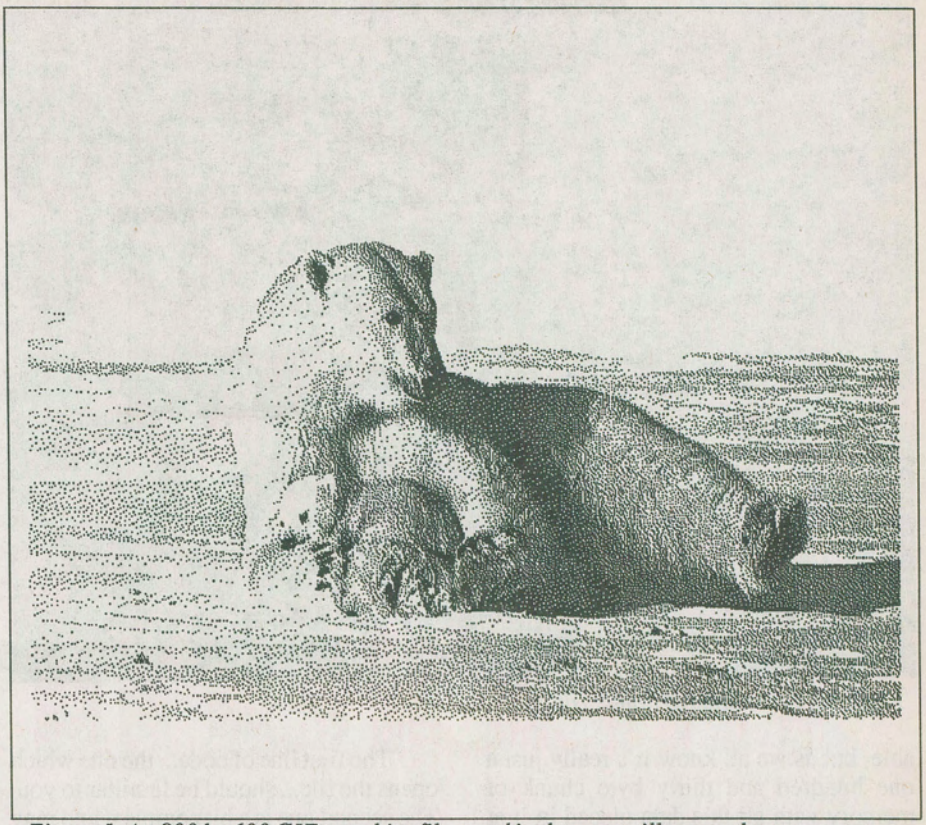

Figure 1. An 800 by 600 GIF graphics file, used in the text to illustrate how memory copes with extra-large files.

On the other hand, it's better than not working at all.

Commercial software which uses lots of memory usually has a multiple option approach to it. If a program needs a big buffer, it might start by checking for sufficient regular DOS memory. If that fails, it will check for the presence of extended memory, using that if it's available. Finally, it will try using a disk file. Programs such as Microsoft Windows do a lot of virtual memory operations to create the illusion of having unlimited amounts of memory for applications.

Shift That Data

A database manager is a very simple sort of virtual memory task, one which is easy to understand. A database is simply a collection of structs, in C terms, written to a file. However, the file can be of any size so the database manager cannot assume that it can simply read it into memory and work on it there. It must deal with its data as virtual memory.

A mailing list is a good example of a typical data base. Each name and address on the list can be thought of as one *record*. Each record contains several *fields*. We might represent a record in C language terms like this.

```
typedef struct{
char first_name[25];
char last_name[25];
char street_address[37];
char appt[16];
char city[16];
char province[3];
char postcode[8];
}MY_RECORD;
```

This struct occupies one hundred and thirty bytes. You can write this into a C language program very conveniently with the expression

```
sizeof(MY_RECORD);
```

This expression would return 130.

We could declare a variable of the type MY_RECORD and fill it like this.

```
MY_RECORD r;
strcpy(r.first_name,"Augustus");
strcpy(r.last_name,"Robes");
strcpy(r.street_address,"14 Dead Pharaoh
Parkway");
strcpy(r.appt,"6 1/4");
strcpy(r.city,"Bentfoot");
strcpy(r.province,"ON");
strcpy(r.postcode,"L4T 4A5");
```

Having done this, the variable *r* is loaded up with all this data. C would like us to treat this as a MY_RECORD vari-

Techie's Guide to C Programming, Part 12

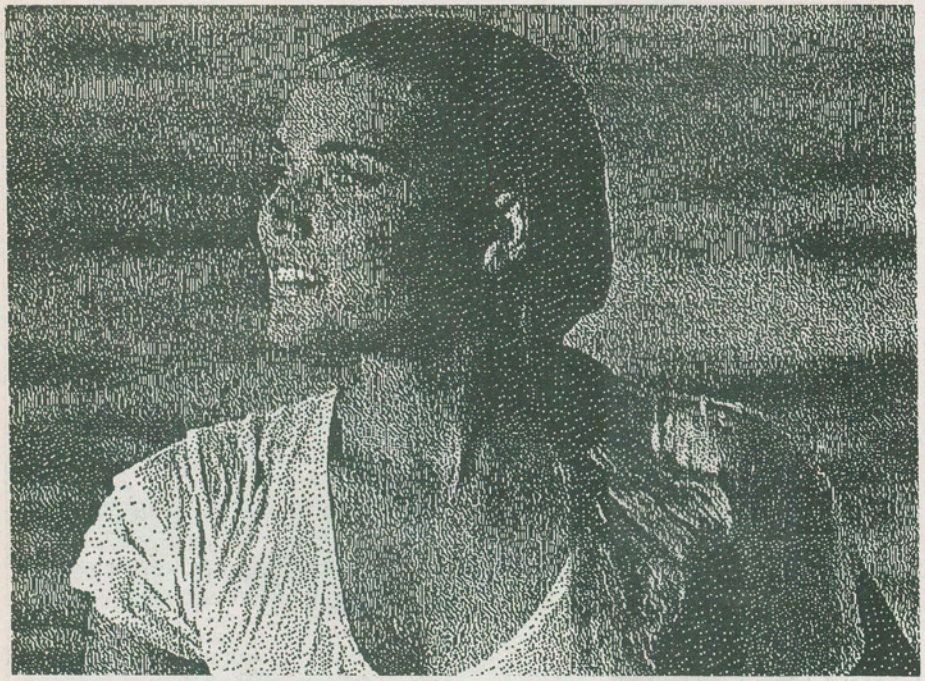

Figure 2. Yet another graphics file.

able, but as we all know it's really just a one hundred and thirty byte chunk of memory with all this data placed in it at strategic locations.

Under C, as you will recall, variables of one type may be *cast* to variables of another providing that their base types are somewhat compatible. What this really means is that in this one particular case you'll be telling your C compiler that you want to break the type checking rules it has set up but that you're doing it explicitly and you take the responsibility for any consequences which might result from your action. We may, for example, cast a variable of the type MY_RECORD to a *char* buffer because we know full well that that's all *r* really is.

This rather obtuse expression illustrates the syntax for this cast.

```
(char*)&r;
```

Here is a simple bit of C code which will create a database file called MYFILE.DAT and write the above record into it as the first record. We'll assume that *r*, above, has been declared and filled with data.

```
FILE fp;
if((fp=fopen("MYFILE.DAT","wb"))!=
NULL){
fwrite((char
*)&r,1,sizeof(MY_RECORD),fp);
fclose(fp);
}else puts("Can't create the file");
```

There's a lot of strange stuff going on in here.

The first line of code... the one which opens the file... should be familiar to you. The second one is a bit complex, and may not be.

The *fwrite* function, like the *fread* function mentioned above, is used to deal with blocks of raw data which move in and out of files. It's usually used with files which have been opened in binary mode, as this one has. Notice the *"wb"* argument to *fopen*.

There are four arguments to *fwrite*. The first one is a pointer to the *char* buffer which is to be written from. In this case, we don't have real a char buffer... we have a synthetic one cast from a MY_RECORD variable, as discussed a moment ago. You will note that the casting syntax is the same.

We could have handled this a bit more readably by doing something like this.

```
char *p;
p=(char*)&r;
fwrite(p,1,sizeof(MY_RECORD),fp);
```

In this case we have explicitly cast *r* to the *char* variable *p* and then written the data from the pointer rather from the buffer which it points to. The results are exactly the same... this approach might be a bit easier to understand, although it uses one more line of code than is really required.

The second argument to *fwrite* represents the size of the objects being written in bytes. The third represents the number of objects to write. In this case we have told *fwrite* that the objects are one byte long and there are one hundred and thirty of them, this being the size of a MY_RECORD struct. We could have reversed these arguments, that is, we could have told *fwrite* to write one object which is one hundred and thirty bytes long instead of one hundred and thirty objects which are one byte long.

The last argument to *fwrite* is the file pointer for the file being written to.

If the write is successful, *fwrite* will return the number of objects which have been written, the value one hundred and thirty in this case. If something goes wrong, such as the disk proving to be full before the entire struct could be written to it, *fwrite* will return the actual number of bytes which made it into the file.

You would check for errors in *fwrite* with the following code.

```
if(fwrite((char*)&r
,1,sizeof(MY_RECORD),
fp)==sizeof(MY_RECORD)){
puts("Written ok");
}else puts("Write error");
```

Bigger Files

Let's now suppose that MYFILE.DAT has acquired the entire mailing list of Publisher's Clearinghouse. It's many hundreds of thousand of records long, teeming with people who don't want to be there even though they may already have won millions of dollars, cars, trips and subscriptions to trashy magazines. Each of these hapless souls is represented by one record structured as a MY_RECORD variable.

The folks at Publisher's Clearinghouse have discovered that Augustus Robes, famed Egyptologist and sample datum, sought to fool them by giving them the wrong post code. For the past four years all his junk mail has been winding up in a home for retired cat skinners in Thunder Bay. Swearing under their breath, they find Mr. Robes' record number from one of his old mailing labels and set about changing his post code.

One may assume that Publisher's Clearinghouse has a very sophisticated database manager to do this sort of thing, but the process would work pretty much like this. We'll allow that the record which stores Augustus Robes' information is record 10967.

To begin with, we need a function to read records from the file. All of the information about Augustus Robes is correct except for his post code, and we do not wish to have to enter everything anew. As such, we would use this function.

```
read_record(fp,record_number,r)
FILE *fp;
unsigned int record_number;
MY_RECORD *r;
{
fseek(fp,
(long)sizeof(MY_RECORD)*
(long)record_number,SEEK_SET);
fread((char
*)r,1,sizeof(MY_RECORD),fp);
}
```

There are a few important things to say about this function. First off, it allows for no error checking, which it would do in a real world application. Second, you will notice that the syntax for the cast in *fread* is a bit different. You would fetch the record in question by using this function as follows.

```
MY_RECORD r;
read_record(fp,10967,&r);
```

As you will recall from several months back, structs cannot be passed by value, only by location. We pass a pointer to the struct *r* rather than *r* itself. As such, having used the *&* operator in passing the thing, we need not use it in the cast within the function. The variable *r* withing the function is not a MY_RECORD variable but a MY_RECORD pointer.

Finally, note that we cast each of the numbers being multiplied together to *long* before performing the calculation. This is a subtle but very important point. If you multiply 10967 by 130 as long integers, you will get the correct result, 1,425,710. This is the position in the file where Augustus Robes' record begins. However, if you multiply them together as straight *int*s and then cast the result to long, the result will be 49,475. This happens because the result of the calculation will be limited to sixteen bits.

In the real world, one might assume that Publisher's Clearinghouse has so many records in its database that it must use long integers to express their record numbers.

Having executed *read_record*, above, the *r* variable will have all of the information about Augustus Robes, including his erroneous post code. We would change this one item,

```
strcpy(r.post_code,"L9G 1Q4");
```

and then write the record back to the file. The function to do this looks pretty much like the one to read it.

```
write_record(fp,record_number,r)
FILE *fp;
unsigned int record_number;
MY_RECORD *r;
{
fseek(fp,
(long)sizeof(MY_RECORD)*
(long)record_number,SEEK_SET);
fwrite((char
*)r,1,sizeof(MY_RECORD),fp);
}
```

It would be called in much the same way too.

```
write_record(fp,10967,&r);
```

These two functions are the basis of any fixed field database manager, that is, of any program which treats its data records as hard wired structs. You can apply them to all sorts of database applications. For example, suppose that the Publisher's Clearinghouse people found out that Augustus Robes had fooled them but they did not know which record his data was kept in. They could do something like this to find him. We'll allow that the variable *max_record* holds the number of the last record in the file.

```
MY_RECORD r;
int i;
for(i=0;i<max_record;++i){
read_record(fp,i,&r);
if(strcmp(r.first_name,"Augustus")==0
&&
strcmp(r.last_name,"Robes")==0){
printf("The record number of %s %s is
%d\n",
r.first_name,r.last_name,i);
break;
}
}
```

Record Time

Obviously, the examples in this article have been a bit simplistic. If you go to write a database manager of your own using them you will want to add some error trapping and, more important, a user interface. In the real world one would not write a custom C program every time there was a need to locate a specific record.

The database manager illustrates a fundamental use of disk files as a way to handle large amounts of structured data. A database manager is really just a slightly more complex version of the picture file we discussed at the beginning of this feature. However, it points up a simple rule of data files. As long as the data is structured in some predictable way, you can handle it as a series of fixed records and, as such, access it fairly quickly.