FEATURE

The Techie's Guide to C, Part 14

This month we're going to have a look at some of the more intricate bits of indirection, pointer notation and other C language nasties.

Steve Rimmer

The nature of C, with its local variables and structurally isolated blocks of code, makes for creating programs quickly and with far fewer bugs than would crop up under a language like BASIC. C also lets you keep all your data together in data structures, a facility which makes dealing with complex blocks of data a lot more practical.

The notation for using complex data structures under C is a bit obtuse, perhaps made more so by the responsibility the compiler places on the shoulders of programmers for differentiating between objects and pointers to objects.

This month we're going to try to make sense of the pointer problem, looking at exactly what a pointer is and what it points to.

It's Rude to Point

If you have a line of C code like this

```
char b[64];
```

you will set up a sixty-four byte buffer... or more properly, an array of *char* variables... somewhere in memory. However, C never lets you deal with complex data types, such as array, directly. As a result, *b* does not contain the array itself. It's a pointer to the array. The size of *b* is two or four, rather than sixty-four.

The size of *b*, and of any pointer, is determined by the number of bytes needed to form a pointer under your compiler's current memory model. This will be two bytes for a sixteen bit pointer under the small and tiny models and four bytes for a thirty-two bit pointer under the large and huge models. Don't worry about this if you don't understand it... it's not all that important at the moment. For the most part, C insulates you from having to know how big a pointer actually is.

This bit of C code will print a string which you have subsequently stored in *b*, above. We'll assume that you've actually done so.

```
puts(b);
```

Now, as you'll recall from previous installments of this series, when you do something like this, C actually copies the argument... *b* in this case... onto the stack so the function it's calling, *puts*, can find it. If we were to pass *b* as an object, this would mean copying sixty-four bytes onto the stack. More than this, though, it would mean that the compiler would have to keep track of the size of every object it uses, and there are several sorts of objects which it cannot know about.

For this reason, the compiler stores pointers to large objects. Simple objects, such as *int*s, *double*s and *char*s are passed directly on the stack. Everybody else gets to use pointers.

Because arrays are always dealt with through the use of pointers, C lets you treat them pretty loosely. Data *structs*, which we've discussed previously, are declared as objects and passed through pointers, so C forces you to distinguish between them when you use struct variables.

Keeping this straight is one the biggest headaches for new C programmers.

Let's see how these pointers work. Here's a struct variable type being declared.

```
typedef struct{
char name[40];
double annual_income;
int number_of_dogs;
}HUMAN;
```

This is all the pertinent information about a person for a program which correlates annual income with the number of dogs the person in question owns. The ultimate usefulness of such a piece of software will be left for discussion some time in the far future.

We can begin by declaring a variable of this type.

```
HUMAN h;
```

At this point, *h* is an object. As such, we might set the name of the person in question like this.

```
strcpy(h.name,"Augustus L. Fizzbatt");
```

The expression *h.name* appears as a string pointer in this case, a pointer to the part of the variable *h* where the string called *name* is to be stored.

Now, here's where things let a bit tricky. Let's say that we have written a function which takes a HUMAN variable as input and returns a mutt factor. A mutt factor is a number which relates the income of the person in question to his or her dog count, the exact nature of which is irrelevant to this discussion. Here's the function.

```
mutt_factor(n)
HUMAN *n;
{
int mutt;

/* calculate the mutt factor */
```

```
return(mutt);
}
```

Now, in this function, *n* is not a struct variable of the type HUMAN, but rather a pointer to it. As such, if we wanted to change the contents of the *name* field in the variable from within this function, we would have to use slightly different notation.

```
strcpy(n-name,"Ivor X. Wombdecker");
```

Likewise, if the calculation of the mutt factor involved multiplying the two numeric values together, it would be done with this "arrow" notation.

```
mutt=(int)(n-annual_income/1000)*
n-number_of_dogs;
```

Type Checking

Another of the things which confuses new C programmers is C's propensity for type checking. In the process of compiling your program, the compiler will make sure that you don't attempt to use a pointer to one sort of object to reference another.

For example, if you create a pointer and a struct like this,

```
char *p;
HUMAN h;
```

the compiler will not allow you to do this without spitting out a few warnings.

```
p=h;
```

In reality, all pointers are the same under the skin, but the foregoing use of *p* is almost guaranteed to cause some problems later on.

There are certainly occasions in which it's desirable to interchange pointers, but in these cases C wants your assurance that you're doing so explicitly, and that you know what you're up to. This process is called *casting*.

This is a legal cast.

```
p=(char *)&h;
```

There are all sorts of perfectly respectable reasons for doing this sort of thing. For example, if you wanted to zero all the fields in the struct variable *h*, you could do it this way.

```
memset((char *)&h,0,sizeof(HUMAN));
```

The *memset* function expects to have a *char* pointer as its first argument.

A struct variable can have any combination of objects as its component fields. This includes other structs. For example, you might define a new struct variable like this...

```
typedef struct{
char breed[32];
int age;
int brain_cell_count;
}DOG;
```

We can now redefine our HUMAN struct to include this new information.

```
typedef struct{
char name[40];
double annual_income;
int number_of_dogs;
DOG the_dog;
}HUMAN;
```

This struct now allows you to store information about one dog. You would access the DOG variable just like any other variable, but the notation for getting at *its* component parts may be a bit obtuse.

In order to change the breed of the dog in the HUMAN variable *h*, you would do this.

```
strcpy(h.the_dog.breed,"Dead poodle");
```

You can nest structures like this for as many levels as you like.

Structs can be placed in arrays, and you can have arrays of structs. This declaration creates an array of sixteen HUMAN variables.

```
HUMAN h[16];
```

Having done this, you would access the eighth entry in this array as follows.

```
strcpy(h[7].the_dog.breed,"Dead poodle");
```

Note that under C, arrays start with the zero'th element. If you want to access the eighth element, you would specify element seven.

The problem with this declaration for the HUMAN variable is that it doesn't allow for more than one dog per owner. It's hard to understand why anyone would want more than one dog... or even one dog at all, come to think of it... but you might want to change the declaration to allow for this possibility anyway. This involves having an array of DOG variables in the declaration for the HUMAN variable.

```
typedef struct{
char name[40];
double annual_income;
int number_of_dogs;
DOG the_dog[4];
}HUMAN;
```

This allows for up to four dogs per owner, which seems like it should take care of even the most masochistic possibilities.

This is how you would change the breeds for all four dogs in a HUMAN variable.

```
strcpy(h.the_dog[0].breed,"Dead poodle");
strcpy(h.the_dog[1].breed,"Brainless set-
ter");
strcpy(h.the_dog[2].breed,"Overfed mas-
tiff");
strcpy(h.the_dog[3].breed,"Plastic
spaniel");
```

On Bounds and Arrays

Under BASIC, if you declare an array with twenty-five elements and you attempt to access the twenty-sixth, BASIC will complain and stop the program. C does not do this. If you have an array of HUMAN variables with twenty-five humans in it and you attempt to write to something beyond this, C will go ahead and let you. If there's some useful data after the declared space for the array it will be trashed. If there's some code there, your program will probably detonate.

C does not check array bounds because it rarely knows what they are. For example, let's say that you wanted to store an array of two thousand HUMAN variables. You could do this.

```
HUMAN h[2000];
```

You compiler would probably reply with this.

```
Too much static data allocated.
```

Even if it didn't, this is a very inefficient way to handle a big array. It would be much better to use the *malloc* function to allocate a big buffer, put your HUMAN variables in it and then blow the buffer away when you no longer need it. However, even though this buffer can be made to behave just like an explicitly allocated array, C can't know how big you've made the buffer and, hence, it can't know the bounds of the array.

It's exceedingly important, in dealing with arrays and complex variables, that you keep in mind that under C you are responsible for keeping you data within the bounds you've set up for it.