The Techie's Guide to C Programming Part 4

Meeting and getting to know the interrupt.

STEVE RIMMER

If the structure of C programs is a tad daunting when you first come upon it, the structure of the data they use is very nearly overpowering. Whereas just about everything under BASIC is handled with real numbers and occasional strings, C has a plethora of data types. Furthermore, if you don't find a data type you fancy after a casual browse through the manual you can easily add your own – C offers facilities to do this.

Now, you might well be thinking that this obsession with data is some sort of brain problem on the part of bald headed programmers in white lab coats, and nothing you should really be concerned about. To some extent this is true, at least for simple programming, but much of the power of C is in how it deals with data. Once again, we see in C an example of language efficiency at the expense of convenience. By forcing you to think about data types, rather than thinking about them for you, as in the case of BASIC, C creates faster, tighter programs.

It's not the fault of C that you haven't got sixteen fingers.

This month we're going to look at data and how it's handled under C.

*char*s, *int*s and *float*s

There are those who would have said that the best thing to do with an *int* if you happened across one would be to step on it and brush the remains under the television set. This is probably a wise move.

The basic unit of data under C – and, in fact, under any language running on a PC – is a byte. A byte is eight bits wide, and, as such, can represent numbers from zero to two hundred and fifty five. This is not a great range of numbers, and bytes all by themselves aren't much use, except for holding fixed range data like ASCII text.

Under C, a byte is called a *char*, for character... indicating its relationship with text.

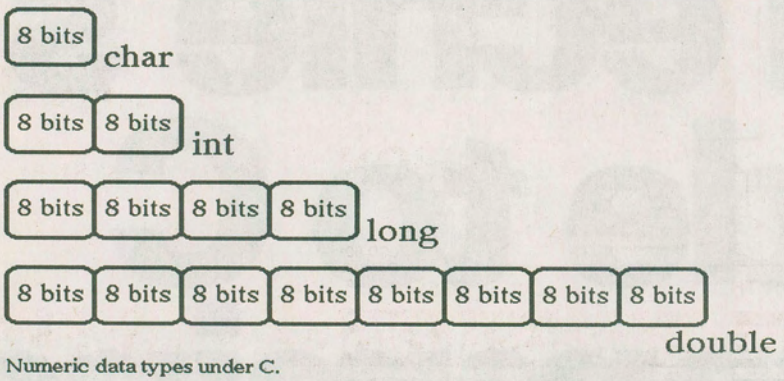
The most usual data type found under C is *int*, for integer. We should qualify this a bit: it's actually a signed integer. This is a 16-bit number – two bytes – and can hold numbers from -32767 to 32768. We can also declare an *int* as being unsigned, which means that it can hold numbers in the range of zero through 65535.

We can move data between *int*s and *char*s freely so long as we're conscious of what we're doing. For example, if we have a *char* which contains the letter "A", which is the ASCII value of 65, and make an *int* equal to this, the value of the *int* will be 65. The lower of the two bytes comprising the *int will be loaded with the value of the char* and the upper byte will be loaded with zero.

If we have an *int* which holds the value of one thousand and we make a *char* equal to this, the value of the char will be two hundred and thirty two. This is a bit of a head scratcher; the two numbers don't really seem to have a lot to so with each other. Actually, they do, but only to a computer.

If you could look at the aforementioned *int* as the computer does, you'd see that it consists of the value three stored in its upper byte and two hundred and thirty two stored in its lower byte. When we try to store it in a *char*, the upper byte gets thrown away.

Moving data between variables of differing types is called "casting". Moving data between *int*s and *char*s is the only case in which C allows us to cast data freely without explicitly telling it what we're up to. Here's a more formal example of casting.



Numeric data types under C.

When 16-bits just aren't enough, and this happens quite frequently when you're dealing with large numbers, C offers another data type called *long*, for long integer. This is a 32-bit number, with your choice of signed or unsigned operation. The unsigned version can hold numbers up to 4,294,967,295, or roughly the number of shoes that Mila Mulroony buys in an average afternoon's shopping.

If you cast an *int* to a *long* just by making one equal to the other, C is not obliged to fill the upper part of the *long* with zeros, so the actual value of the *long* may be undefined. The proper syntax to make sure that the *long* represents the same thing as the *int* is this:

```
int i;
long l;

l = (long)i;
```

There are several more complex situations wherein this becomes still more important. For example, consider that we have two *int*, called *m* and *n*, and that they each hold sort of large numbers, such that if they're multiplied together the result will require a *long* to hold it. This expression

```
long l;
int m,n;

l = (long)m * n;
```

will avail of us of an erroneous result. The two *int*s will be multiplied together as 16-bit numbers, they'll overflow and the resulting 16-bit mistake will be cast to a *long*. What we really wanted to say was this.

```
l = (long)m * (long)n;
```

Up until now, all the numbers we've dealt with have been integers of one sort or another. C also has floating point variables, and the performance of the floating point packages of various C compilers is the subject of much contention and bench marking. Floating point variables are of the types *float* or *double*, depending on their precision. In fact, on most PC compiler packages, all floating point numbers are treated as *double*s. A *double* is a 64-bit number.

You can cast between integers and floating point numbers with the same syntax as you would between *int*s and *long*s, although you should be aware that when you do C is installing some fairly elaborate code to translate between its floating point representation and straight machine level integers.

If you've done any BASIC programming you might well look at all this and wonder why anyone would want to pound his or her head against such a hard, poorly mortared wall, juggling all these variable types, when you can just let a language like BASIC take care of them for you. The reason is fairly simple. Under BASIC, variables default to floating point, with integer numbers optional. Floating point numbers are hundreds of times slower to work with than are integers. Long integers are slower than short, 16-bit, ones, and short integers are correspondingly slower than are *char*s.

If you let the language choose your variables for you, it will have to choose the ones which can handle the most complex numbers possible, as it cannot know the use to which you intend to put the numbers. You, hopefully, can figure this out in advance, and in forcing you to chose the precise data type for every variable in your program, C allows you to optimize things in a way that it cannot. This contributes greatly to the ultimate speed and size of your final program.

Pointers from Hell

So far, all we've looked at have been simple numbers. Anyone can cope with numbers. Under C, however, we also have pointers. Pointers are dreadful, horrible, ugly, repulsive, demonic things which will crawl into your head and wrap their slime encrusted tails about your brain, muttering insanely into your ear until you go mad. No foolin'.

Regrettably, you can't really get into C without them.

Under C, there is no data type which can hold strings. Instead, C forces us to treat strings as what they really are, that is, a collection of bytes. Hence, a string under C is defined as an array of *char*s.

This is how we define a string.

```
char s[65];
```

Having done this, the variable *s* is a sixty five byte string. Under C, all strings are terminated by a zero, so for practical purposes, we must be sure not to try to store more than sixty four bytes of text in this string, to leave room for the null at the end.

Here's a typical application of this string.

```
strcpy(s,"Wombats in love");
```

This will copy the second string into the first. We are passing two strings to the function *strcpy*... almost.

When we pass integers to a function, we really pass the actual numbers. The mechanism for doing this is to push the numbers up on the stack, call the function and then pop them back off the stack again. The function peeks at the stack to find the numbers it was passed. Don't worry if you aren't really into stacks just yet – the internal workings are not all that important just yet.

The meaningful bit to consider, though, is that to pass a sixty five byte string in the same way as we'd pass an integer, we'd have to save an awful lot of data somewhere before we called the function it's being passed to. This would be very, very inefficient, and C won't let you do it just on principal. As such, when we talk about passing a string, what we really mean is that we pass a pointer to a string.

A pointer is simply a number which represents the location in memory where a

The Techie's Guide to C Programming

thing lives, as opposed to the thing itself. If the string "Wombats in love" lived at location one thousand, then we would, in effect, be passing the number one thousand as a pointer to it. A function which expects a pointer to something will know to look at where the pointer points to get at whatever it is being passed.

Since strings are *always* passed around as pointers, it's quite painless to deal with them as such. Wait a sec... we'll get into pointers to other things shortly.

If *p* is a pointer to the string "Wombats in love", then *p[0]* will be a char of the value eighty seven... the ASCII value for "W". If you recall our discussion of this notation a few months back, the value of **p* will also be 87 — the two representations are equivalent in this case.

The important thing to note about passing values versus passing pointers is this. If you pass an *int* to a function and the function changes it, it will not affect the value of the *int* in the function which called the function that did the changing. When you pass a variable to a function, you are passing a copy of that variable. If you pass a pointer to a variable, however, the called function has access to the actual number in the calling function, and it can affect the value of it.

As such, consider this function.

```
strupr(s)
char *s;
{
int i,j;

i = strlen(s);
for(j=0;j++) s[j]=toupper(s[j]);
}
```

This bit of C code will translate any string passed to it into all upper case. The function it calls, *toupper*, is a library function which returns the upper case version of any alphabetic character passed to it. The important part to observe about this function, however, is that it doesn't *return* an upper case string; it changes the actual string passed to it.

Now, to finish things off, we're going to look at a slightly more obtuse bit of pointer notation... pointers to *int*s.

Writing a function to exchange the values of two integers is a classic C language problem. Here's how it *isn't* done.

```
swap(i,j)
int i,j;
{
int t;

t=i;
i=j;
j=t;
}
```

This doesn't work because the *i* and *j* that this function gets to work with are *copies* if the *i* and *j* in the function which calls it, and those copies get thrown away when this function is finished. This is the correct function

```
swap(i,j)
int *i,*j;
{
int t;

t=*i;
*i=j;
*j=t;
}
```

We must call this a bit differently too.

```
int i,j;

swap (&i,&j);
```

The & operator tells C to pass the address of the thing it's in front of, rather than the thing itself. Declaring *i* and *j* as being pointers to *int*s rather than the *int*s themselves in the *swap* function allows us to deal with the actual contents of the numbers. Just as we said that the notation **p* got at the first byte in our string, above, so too does **i* get at the value of the integer pointed to by *i*.

Pointing out

If this all seems a bit obtuse, don't let it bother you too much at this stage. Because it forces you to deal with the real world in its handling of data, C makes very clever use of its numbers but it also requires that you think a lot more about what you *really* want to do.

Data type errors are amongst the most common problems in writing C programs, and, as such, the latest generation of compilers are very good at spotting them. If you try to do anything untoward the compiler probably will help you avert it. This is extremely useful while you're getting your mind around just how all this peculiar notation goes together.

Next month we're going to look at complex data. You probably thought that all this was quite complex enough. Wait 'til you find out about *structs* — they make pointers look almost civilized. ■