FEATURE

The Techie's Guide to C Programming

Part two, in which Doctor Max O'Blivion locates an ancient map which promises of vast treasure and many parenthetical digressions. Danger lurks everywhere.

STEVE RIMMER

A lot of very zen-like things have been said about programming in C language. I doubt that the greater part of them were original thoughts at the time – zen-like things rarely are. They're simply adapted from previously uttered zen-like things said about related subjects. This, to a large extent, further enhances the zen-like nature of their uttering, somehow suggesting the universality of all human experiences. After all, if you can say roughly the same thing about C language programming and ox gelding there has to be some sort of intelligent design to the universe.

This is how quite a few of those clever sounding slogans for coffee mugs and tote bags come to pass, methinks.

My favourite of these zen-like utterances, which I think I mentioned last month, is the all but immortal "real men don't use Pascal". Add to this that no one over the age of four programs in BASIC and that only people who like pain enjoy assembler and there's little left. We'll discount the yuppie languages like Forth, Ada, COBOL, SnoBOL, Fortran, BLINK, Yawn, gROUCH and Queeg for the moment.

As the gods have clearly indicated that C is *their* preferred way of dealing with computers, one can do little else but get on with it. As such, in this feature we're going to look at a few new aspects of the language and learn a bit more about compilers.

How To Kill Cats Quickly

The structure of C programs can be a bit elusive until your brain attaches itself to the nature of the language. It's helpful to remember that everything under C ultimately represents some sort of value, with the largest part of what you find in the average program representing an integer.

This program will echo everything you type on your keyboard to the screen. Let's see how it works.

```
main( )
{
while(putch(getch( )));
}
```

Anyone with an ounce of common sense, or a millilitre of common sense if you've caved in and gone metric, will see that this works by black magic, and that there's little sense in any further discussion. Fortunately, few people willing to invest several thousand dollars in a television set that only displays letters can be said to be that sensible, so we'll press on.

It will be helpful to start by knowing that *putch*, as we saw last month, will take the character passed to it and print it to the screen of your computer. Thus, if we said *putch(65)*, the letter *A* would appear on the screen, the number sixty five being the ASCII code for the letter *A*. Under C, by the way, we could also say *putch('A')*. A single character in single quotes will be interpreted by C as working out to a numeric value, rather than as a string.

Note that we can only pass *putch* numeric values – ASCII codes – not strings. It prints one character at a time.

The *getch* function is another thing which is built into C. When you call *getch* your program will wait until a key is pressed on your keyboard and it will "return" the ASCII code when *getch* returns. This may be a bit confusing.

This program calls *getch*.

```
main( )
{
getch( );
}
```

In this case, *getch* waits for a key press

and the program ends and returns to DOS. The actual value of the key hit gets thrown away, as there's nothing to catch it.

In this case, something more happens.

```
main( )
{
int i;

i=getch( );
}
```

This program assigns the value returned by *getch* to the integer variable *i*. However, nothing happens to *i* in this program either.

In theory, every function under C returns something. Much of the time this returned value is of no use to us, and we ignore it. In the case of *getch*, we may or may not use it for anything, depending on whether we're using *getch* to wait for a key press or actually to see which key has been belted.

Let's expand the program a bit further.

```
main( )
{
int i;

i=getch( );
putch(i);
}
```

This program will get a keyboard character and print it to the screen. It waits for a key press with *getch*, assigns the keyboard character to *i* and then uses *putch* to send it to the tube.

We can write this another way.

```
main( )
{
putch(getch( ));
}
```

Daring souls who've been awake for a few hours might well ask where the integer went. It's still there, sort of. It's just hiding in the depths of C.

In order to properly understand this latest example, we have to know before hand that presented with a set of nested functions like this, C will always resolve the innermost one first. In this case, it will execute *getch* first, derive its returned value... a character code... and then execute *putch* with the returned value of *getch* passed to it. The program itself will create a place to store the returned value of *getch* in the interim. Variables that appear and disappear for these sorts of purposes are called "temporaries", and, as we'll see a bit later in this series, we can often use them to make C do very clever things.

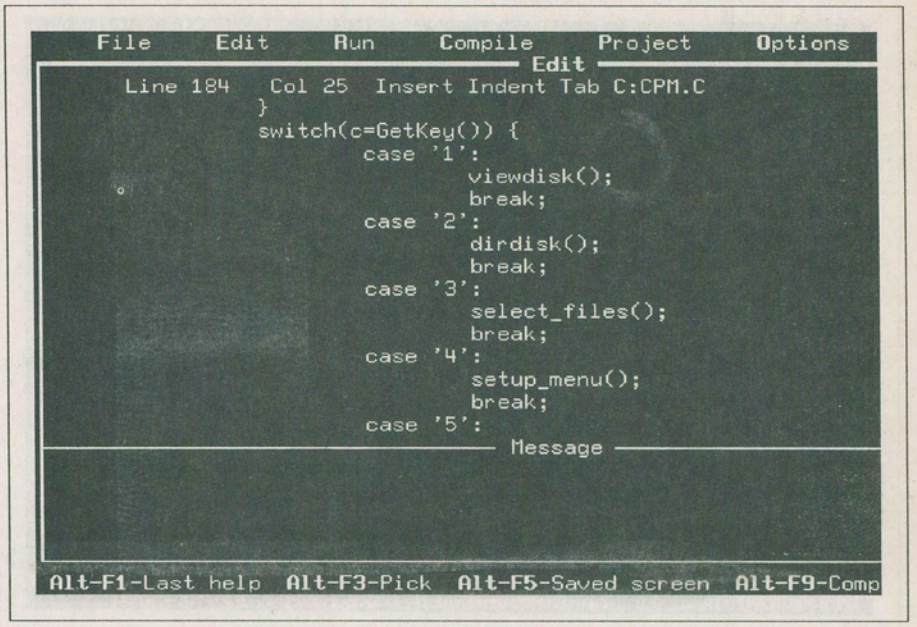

Figure 1. This is an example of a switch statement... or part of one... in a real C program being developed in Turbo C. The GetKey( ) function is an expanded version of getch, discussed in this article.

As we noted last month, the *while* statement under C will execute something repeatedly until the condition it's testing becomes false, that is, until it is zero. Thus, this line will execute forever

```
while(1);
```

and should probably be avoided unless you just happen to like rebooting your computer.

This line... as seen in the first example we looked at, actually... will also execute forever.

```
while(putch(getch( )));
```

but at least it'll do something. This will cause *getch* and *putch* to be called repeatedly, echoing what you type to your tube. Actually, you can break out of this loop on a PC by hitting control break.

As it turns out, this isn't really a perfect dumb typewriter program. For example, if you hit *Enter*... a carriage return... the cursor will go back to the beginning of the current line, not down to the start of the next one, as is usually the case when one hits *Enter*. The reason for this is that C... and your PC... interpret a carriage return literally. The cursor simply returns to the start of the current line. We're used to hitting *Enter* and having DOS or whatever we're typing into issue both a carriage return and a line feed.

Alien Swamp Monsters

Let's change our program a bit. This is how little, elegantly simple programs quickly swell into huge, multi-headed alien swamp monsters that engulf whole mid-Western American towns. What fun...

```
main( )
{
int i;

while((i=getch( ))) {
putch(i);
if(i==13) putch(10);
}
}
```

There's a lot of new stuff happening here. First off, as you'll notice, we're back using a variable once more. Again, we're testing the returned value from *getch* for the validity which drives our *while* loop – not all that sensible, as you can't actually type character zero at the keyboard to end the loop, but never mind that for now.

The line after the first *putch* is an example of an *if* statement in C. This says

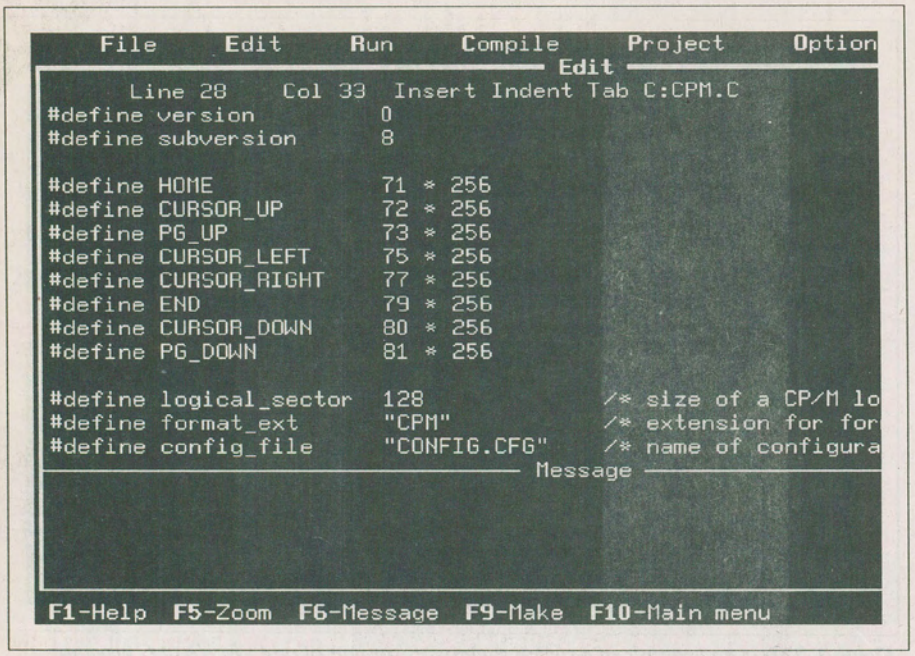

Figure 2. Here are some actual uses of the #define directive in a C program. Note that they define not only numeric values, but strings as well. This program, in its entirety, translates CP/M diskettes into PC ones.

that if the value of *i* is thirteen, the program is to print character ten as well. If we bear in mind that the ASCII code for a carriage return is thirteen and that of a line feed is ten, the purpose of this line becomes pretty obvious. It causes every carriage return to be followed up by a line feed.

However, the way we test the value of *i* may seem a bit odd. In BASIC, equal signs can be used in two ways. We might say A=21 to assign one thing to another thing. We also might say IF A=21 GOSUB 1000 to test the value of a thing. Under C, we differentiate between these two uses.

If we say *a=21* under C, this can only mean that the variable *a* is being assigned the value twenty one. If we wish to test the value of *a*, we must use two equal signs, as in *if(a==21)*. This makes C happy... don't argue with it.

Smart compilers, knowing the propensity of the human digit to miss keys whilst typing more rapidly than the human brain can properly think, check to make sure that these two uses are not interchanged. For example, if we say *while(i=getch( ))*, a good compiler will complain, as this is ambiguous at best. What we want to do, in fact, as we've seen in the example above, is to assign *i* the return value of *getch* and then test the value of *i* to see if it's true. However, this could also mean that the *while* statement is to regard things as being true so long as the value of *i* is the same as that of the return value from *getch*, which is very clearly not what we want to do.

For this reason, the contents of the tested part of the *while* loop above is enclosed in an additional set of parentheses to force C to evaluate it first and remove the ambiguity.

We might improve this line, actually, and we'll see how the two uses of the equal sign are properly employed. If we change the line with the *while* loop to

```
while((i=getch( ))!=27){
```

we will have modified to program to allow for an escape clause – quite literally. This means that the *while* loop should assign *i* the value of *getch* and keep looping so long as *i* does not equal twenty seven. This is the value returned when you hit the escape key. As such, hitting *Esc* will end the program.

The exclamation point is something different as well. When we wish to see if one thing equals another, we use two equal signs. We when wish to see if one thing does not equal another, we use an exclamation point and an equal sign.

There's another problem with this program. We can't backspace. We also can't deal with tabs. We need more *if* clauses. This is going to start getting a little messy.

Having to program many possible reactions to the value of something, depending on the value, is something which happens quite commonly. As such, C provides us with a very elegant... and somewhat daunting... construct called the *switch* statement. This allows us to do away with a lot of *if*s, as well as tightening up our code a bit in the bargain.

Let's rehash the program once more.

```
main( )
{
int i;

while((i=getch( ))!=27){
switch(i){
case 8:
putch(8);
putch(32);
putch(8);
break;
case 9:
printf("    ");
break;
case 13:
putch(13);
putch(10);
break;
default:
putch(i);
break;
}
}
}
```

I said it was daunting. With each iteration through the *while* loop, the *switch* will evaluate the contents of *i*. If it matches one of the three specific cases we've set up – eight is a backspace character, nine is a tab and thirteen is, as we've seen, a carriage return, it will do whatever is in those cases. Otherwise, it will execute the *default* case, which is simply to print the character to the screen.

The word *default* here is a specific case defined by C. You have to have it called *default* in order for it to behave as a catch all for numbers that don't conform to any of the other, specific cases.

Each case starts with the *case:* statement, naturally enough, and ends with a *break;*. The *break* tells C that the case is done with, and that it should skip ahead to the end of the switch immediately. If you leave the break statement off, C will execute the contents of the next case in the

Continued on page 58

The Techie's Guide to C

Continued from page 18

list when it finishes with the current one. This is actually useful sometimes.

You'll probably note that, strictly speaking, there is no *break* needed after the *default* case, as it's the last case in the list, and there's no where for C to fall through to if the *break* had been left off. This is true enough. In this case, C actually ignores the existence of the last *break* when it compiles the program. However, if we were to tack another case onto this *switch* some time in the future, we might forget to go back and add the necessary *break* to the *default* case. As such, it's a good rule to always end cases with *break*s, whether they're really needed or not, unless there's a good reason to omit them.

The contents of the three cases should be pretty easy to work out. The case for eight, the backspace, prints a backspace, a space and another backspace, so it moves to the previous character, blows it away and then moves back into the now vacant position. The case for nine, the tab, just prints five spaces. You'll recall this use of *printf* from last month. Finally, the case for thirteen, the carriage return, should be pretty obvious.

Definition of Terms

Before we vanish into the swirling mists of eternity, let's consider a bit of compiler lore. This has nothing to do with the above example; it's just a useful thing to have your head around when you're writing programs.

One of the slickest features of C is its *#define* statement. This is properly called a "pre-processor directive", which is a term that means that the authors of the language liked big words. Like the *#include* directive we saw last month, this is something which happens before the compiler starts worrying about the contents of your program.

Here's a common use of *#define*

```
#define pi 3.1415926

main( )
{
printf("The value of pi is %f",pi);
}
```

The syntax for the *printf* statement is a little weird; don't sweat it right now. The important point is that the *#define* statement has associated the number 3.1415926 with the label *pi*.

This may seem like just another sort of variable. It's not. What actually happens here is that before it compiles the program the compiler goes through it and finds every occurrence of the label *pi*. It then mechanically replaces each one with the number 3.1415926. This results in a lot faster code than would have come to pass if we'd used a variable for *pi*.

If you had a program with lots of occurrences of *pi* in it, you would use a *#define* to initially establish what *pi* actually is. You might want to experiment and see what effect changing the precision of *pi*, the number of digits after the decimal point, has on the working of your program. In this case, all you need do is to change the number in the *#define*.

You can assign anything to a label in a define. For example,

```
#define name "Wombat McAngleiron"
```

tells C that *name* should be replaced with the string *Wombat McAngleiron*. Now, if you attempt to use *name* in your program in some place where a string would not be appropriate, C will most certainly complain.

Finally, you can even use this facility to meddle with function names. For example, if you print character seven to the screen, the speaker on your PC will beep. Thus, we might

```
#define beep( ) putch(7)
```

This will appear to create a function called *beep* which makes the speaker sound. This is slightly more efficient of space and speed in your program than would be creating a real function called *beep*.

Beam Me Up, Scotty

If you're not exactly illuminated about C language programming as yet, don't sweat it. It's a bit like listening to rock 'n roll... you have to let it wash over you for a while until the lyrics start to make sense. Unfortunately, depending on the rock 'n roll you check out, when they finally do make sense, the lyrics might turn out to be telling you to go eat live sheep. At least C language, when it finally does become clear to you in a flash of blinding insight, won't involve mutilating livestock.

Next month, we'll have a look at some specific PC compatible C language compilers. If you're just itching to be able to actually write some code, you'll want to get the next edition of ET&T to find out what's best to write it in.

Bye... ■