FEATURE

Techie's Guide to C Part 15

One of the less than obvious aspects of C programming is how return values are handled. This month we'll scrutinize the registers.

Steve Rimmer

A C language program ultimately consists of functions called by functions called by functions and so on until you work your way back up the tree to the *main* function, which is called by DOS. This structure imposes a simple, easy to understand order on the potential spaghetti dinner of writing a program. A function is any black box which accepts zero or more values and optionally returns something.

Under C, the nature of "something" is largely left up to your imagination, and the way in which you deal with the things your functions return will influence the structure of the code you write in C. The effective use of return values will go a long way towards making your programs compact, understandable... and functional.

Unfortunately, without understanding how return values really do their magic, it's quite easy to tell C to perform something with one that it can't do. It's just as easy to pass over opportunities to create more effective code.

What Goes Up...

There's a very simple rule for return values under C. Things passed *to* a function are stored on the stack, using the techniques we have discussed previously. Things which are returned *from* a function are always stored in the machine registers of the processor of your computer. In this way, the return value of a function can be ignored or used at the discretion of the calling function.

You might consider the *getchD* library function under C, a function which waits for a keyboard character when it's called and returns it as the low order byte of an integer when someone actually wakes up and belts a key. Now, you can use this to actually see what the next key is by doing something like this.

```
a=getch();
```

or you can just wait for a keypress...

```
getch();
```

In the first case, the variable *a*, presumably some memory on the stack, will be loaded with the value returned by *getch*. In the second, the registers are all ignored when *getch* returns.

The return values of C functions always appear in predictable registers. For sixteen bit values such as *int*s and pointers under the small memory model, the return value is stored in the AX register. For functions which return *char* values, the returned value can be found in the AL register. For functions which return thirty-two bit values, or far pointers consisting of segment and offset values, the low order word is returned in AX and the high order word in DX.

If you keep in mind that whatever you return must fit into the available registers... into thirty-two bits for most applications which do not involve floating point numbers... you'll avoid a number of obvious problems.

In theory, any object can be returned by a function. In practice, as we've seen, this is not so. For example, this would not work.

```
struct ffblk getfirst(s)
char *s;
{
struct ffblk f;
/* ...some code goes here */
return(f);
}
```

This function returns a forty byte struct variable... or, at least, it thinks it

does. As we've just seen, you can't actually do this. What you could do is this:

```
struct ffblk *getfirst(s)
char *s;
{
struct ffblk f;
/* ...some code goes here */
return(f);
}
```

This second function returns a *pointer* to a forty byte struct, which *is* something C can do. In this case, the actual returned value is a sixteen or thirty-two bit value, which will fit in the available registers.

As an aside, if you're interested in seeing what the actual returned values from functions are, you might want to nose around using some "pseudo-variables" provided under Turbo C. If you were to put this line of code in your program,

```
printf("AX=%X",_AX);
```

the value of the AX register at that point in your code would be displayed. There are similar pseudo-variables for all the useful 8088 registers under Turbo C... you can use them to see what really comes back from C functions if you like.

The second version of our imaginary function, above, has a problem too... but it's very obscure. If you were to write such a function, the results would probably be useless. They would be correct when the function returned... the returned value would point to a valid *ffblk* struct in this case... but it wouldn't stay that way very long.

Recall that variables allocated within functions are actually placed on the stack for the life of the function and then thrown away. The returned value of our function, then, would point into a stack variable which would have been deallocated just before the function returned. If you call another function before you use the data... or if a keyboard interrupt were to be thrown, for example... the data in this variable would be overwritten and the value returned by the function would find itself pointing to garbage.

The solution to this problem is to have C allocate the variable in a place which will not be overwritten. This type of storage is called *static*... it's allocated when the program starts up and survives untouched even when the function which owns it isn't being used. Here's how this works.

```
struct ffblk *getfirst(s)
char *s;
{
static struct ffblk f;
/* ...some code goes here */
return(f);
}
```

This function will return a pointer to valid data. When you return pointers from functions, it's extremely important that you take care to make sure that whatever you're pointing to will actually exist when it ultimately gets used.

Pointer Checking

Unless it's told otherwise, C assumes that all functions returned signed integers. You can tell it otherwise by declaring at least a partial *prototype* for a function like the one we've been looking at somewhere near the top of your program. You would say this:

```
struct ffblk *getfirst();
```

This tells C to expect a pointer to an *ffblk* struct from *getfirst*, rather than an *int*. In a large model program, this means to expect a thirty-two bit number rather than a sixteen bit one, and failing to do this can result in some pretty spectacular system crashes if you attempt to write to an illegal pointer created in this way.

This prototype also helps C type-check your code. Not only does it know that *getfirst* returns a pointer, but it also knows what sort of pointer it can legally be assigned to without a cast. This would cause the compiler to complain,

```
char *p;

p=getfirst("*.*");
```

whereas this would not.

```
struct ffblk *p;

p=getfirst("*.*");
```

Both variables actually have the same structure, but C keeps you from accidentally interchanging them.

Return Ticket

It's important to realize that a return value can be used just like a variable. For example, allowing that a hypothetical function called *message* returned a pointer to a string, you could print the string this way.

```
char *p;
p=message();
puts(p);
```

You could also do it this way.

```
puts(message());
```

Let's further hypothesize that the function *message* returns a string based on an error code, as returned by the function *screwup*. You could so this,

```
int i;
char *p;

i=screwup();
p=message(i);
puts(p);
```

or you could be much more elegant.

```
puts(message(screwup()));
```

In nesting return values like this, it's important to note that the innermost function will always be called first, its return value evaluated and then passed as an argument to the next innermost function, and so on.

You might well ask whether the nested version actually creates better or tighter code... or if it just looks like it does. The answer is not all that clear. In theory, the registers set by *screwup* should be pushed directly onto the stack and *message* should be called. This would be decidedly more efficient than saving each value in a dedicated stack variable. In practice, your complier may or may not always handle things this way. It very often likes to create temporary stack variables behind your back to hold the results of nested function calls like this one, minimizing the actual space and time saving of such a structure. This will usually be the case if your nested functions involve one or more functions which accept more than one argument.

In the above example, the compiler should not have to use any temporary variables, and the result should be much tighter code through using nested return values.

The most important thing to remember about using the values that are returned by your functions is that they behave like normal C values... *int*s, pointers and so on... and should be treated as such. As long as you apply the same rational to them what you would to other C numerical entities they won't creep up behind you and grab you by the ears. ■