FEATURE

Techie's Guide to C Programming Part 8

The behind the scenes working of your C compiler can be interesting and ultimately useful in understanding how C operates.

STEVE RIMMER

In discussions of how various things work in C, you will occasionally glimpse what's going on behind the scenes. Detractors of C have described it as being little more than a glorified assembler, and in a sense this is true. C allows you to work just a few layers above the actual machine level of your programming environment. This is, in fact, one of its main strengths... it gives you access to almost the same level of your hardware that assembly language does, without necessitating that you write every byte of code explicitly.

It's quite possible to work in C and never really know how it does things, but to do so is to deny yourself understanding of a very powerful aspect of the language. Knowing how things are passed, how C uses its stack and memory and so on will make you more able to bend it to your needs.

It's important to keep in mind that once you have compiled a C program, the resulting EXE file is just a big machine language program and, as such, C doesn't do anything magical or beyond the scope of what you *could* write in assembly language. C just lets you do it more conveniently and... hopefully... in a more structured and organized way.

This month we're going to peek behind the lexical fiction of C to understand what the language is really doing when your programs run.

Stack of Stacks

A stack is one of the fundamental structures of any computer. Early processors, such as the 6502 which drove the Apple][+ and the eight bit Commodore machines, were crippled by their severely restricted stacks. The 8086 series of processors which drive the PC and its descendants were designed with languages in mind which use the stack extensively.

A stack is simply a chunk of memory. The 8088 always has a stack going somewhere, and this is referred to as *the* stack.

As you will realize in a while, you can create synthetic stacks within a program if you want to, and this is often a powerful programming approach once you fully understand the usefulness of stack structures. On the other hand, C allows you to make some pretty clever use of *the* stack, the processor's stack.

For our purposes, a stack really consists of a chunk of memory and pointer into that memory, which we'll call the "stack pointer". Initially, the stack pointer points to the highest location in the stack's memory. In practice, the stack is usually the highest object in memory, so the initial stack pointer points to the top of free RAM.

A stack is called a "first in last out" structure. If we "push" a number onto the stack, that number will be stored at the place where the stack pointer points, and the stack pointer will be decremented to point to the next location down the stack. Stacks always grow downward. Note that we haven't specified how big a position on the stack is yet... that will come in a moment.

Subsequent numbers which are pushed onto the stack will cause the stack pointer to be further decremented.

When we "pop" the stack, we get the last number pushed onto the stack back, and the stack pointer is incremented to point to the previously pushed number. All the numbers on the stack below the current location of the stack pointer are considered to be garbage. Once popped off the stack, a number is no longer valid.

Here's what this might look like in C. In this example, each element on the stack is an *int*.

```
#define stack_size256
int stack[stack_size]
int stack_pointer=stack_size-1;

push(i)
int i;
{
if(stack_pointer = = 0) {
puts("*** Stack overflow ***");
exit(1);
}
stack[stack_pointer--]=i;
}

pop()
{
if(stack_pointer = = stack_size) {
puts("*** Stack underflow ***");
exit(1);
}
return(stack[stack_pointer+ +]);
}
```

This is actually a bit simplistic... the stack pointer isn't even a pointer, but rather and index into an array of stack elements... but it illustrates two of the common problems with stacks. If the stack is pushed too often it will exceed the space allocated for it and trample on something else... most often your program... creating what is called a "stack overflow" condition. If it's popped too often it will "underflow", and the stack pointer will back up over whatever is above the stack.

In the case of the PC, the top of the stack usually resides at the top of a segment, so a stack underflow condition will usually trample something lower down in memory. Don't worry if you don't see why that is just yet.

Accessing the real processor's stack is handled by special machine language instructions, of course. You do not manipulate it directly from C.

The stack is used for a number of purposes. The most basic of these handles the calling and returning from of subroutines or, in C terminology, functions. Consider this simple program.

```
main()
{
print("Steal your face");
}

print(s)
char *s;
{
while(putch(*s+ +));
}
```

When *main* calls the function *print*... and, indeed, when *print* calls *putch*, a library function... the processor executes a machine language CALL instruction to where the actual machine language code which will do what *print* says to do is stored in memory. The mechanism of the CALL instruction is as follows.

First off, it takes the address of the next instruction after the CALL and pushes its onto the stack. Next it takes the address of that which is to be called and puts it in the IP register of the processor. The IP register... the instruction pointer... tells the processor where the next thing it's to do is located in memory. Then it returns control to the processor, which executes the code for the function. The "return address", the place where *main* is to resume after the call to *print*, is now stored on the stack.

The last thing in the code for *print* is a machine language RET, or return, instruction. Under C, a RET is implicitly placed at the end of every function by the compiler. The action of a machine language RET is to pop the most recent number off the stack... hopefully the return address we spoke of a moment ago... and put it in the IP register of the processor. As such, the processor returns to the next instruction after the CALL.

The second use of the stack is to store numbers. Consider this bit of code. In this function, we have a variable, *a*, which for reasons not adequately explained herein we want to use for two things in the course of the function.

```
my_function(n)
int n;
{
int a;

a = n * n;
push(a);

if(a = = 0) {
for(a=0;a;+ +a) puts("A equals zero!!!");
}

a=pop();
return(a);
}
```

Delightfully pointless, this function will return the square of its argument, and it will print out a warning ten times if its argument is zero. There are lots of better ways to do this even if you could think up a reason why you'd want to do it at all. The important thing, here, is that we've used our *push*. and *pop* functions to temporarily save *a* on our make- believe stack and then restore it.

This is a use of the stack which is seen frequently in machine language, because the processor has a limited number of registers, its equivalent of variables. If a program wants to save the contents of one for a while and then get it back, it will usually do so by pushing it onto the stack and popping it off later.

Here's the third use of the stack. This one is important. If you execute this bit of code...

```
my_function(12);
```

... have you ever wondered exactly what happens to the number twelve? Ob-

viously, it has to go somewhere. In fact, the process for passing arguments to functions under C is brilliantly flexible, but so weird you'd wonder how anyone ever thought it up.

When our program calls *my_function*, the first thing it actually does is to push twelve up onto the processor's stack. Then it calls the code which actually makes *my_function* go. Finally, it pops twelve off the stack and throws it away.

In between all this, *my_function* accesses the twelve on the stack by taking the current stack pointer and peeking back up the stack to find out what was pushed up there prior to the numbers which represent the return address of the function. This is a bit of dance, to be sure. One of the complexities of C is that each function has to know what sort of arguments to expect because it has to know how to peek back up the stack. This is why when we write a function with arguments, we must declare what each argument's type is.

The final use of the stack under C is as a place to store local variables. If we write a function like this...

```
dog_breath()
{
int i;

for(i=0;i;++i) puts("Dogbreath");
}
```

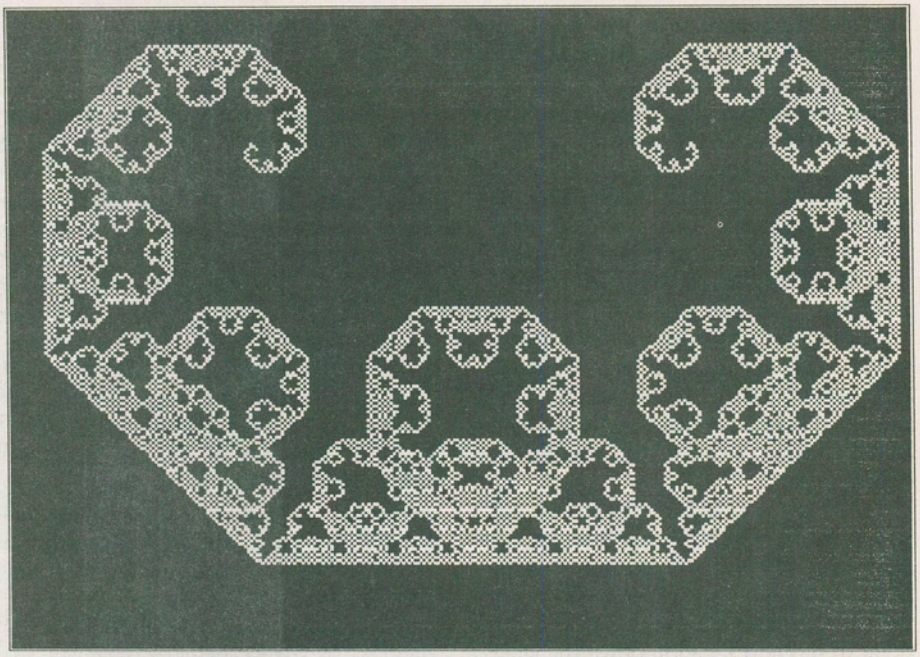
upon execution, the function will allocate some variable space for *i*. It does so by subtracting one from the current stack pointer, thus creating a gap in the stack. This gap will be used to hold the value of *i* for the duration of the function. When the function is complete, it will add one to the current stack pointer, thus closing up the gap, throwing away whatever was in *i* and restoring the stack to what it was, all ready to have the return address popped off it.

In this way, space for variables is allocated only for as long as they're needed.

The astute reader will note that variables passed as arguments to a function exist as numbers on the stack and variables created by the function itself also exists as numbers on the stack. This is why C is able to treat them the same way. Handy, isn't it...

Real Stacking

The stack of an 8088 is sixteen bits wide. This means that every time a program pushes a number onto it, one sixteen bit *int* is stored. The stack pointer moves by two bytes. You cannot push an eight bit... one byte... number onto the processor's stack... rather, you must save it in part of a sixteen bit number and push this.



Larger objects, such as thirty-two bit *long* numbers or sixty-four bit floating point values are pushed onto the stack sixteen bits at a time, so that if you pass a *long* value to a function, two things actually get pushed onto the stack.

Knowing this, we can optimize our programs a bit. For example, there is no space saving in declaring a single *char* value rather than a single *int*, because any variable allocated from within a function... excluding static variables, which we won't talk about now... must have its space allocated on the stack, and that space comes in *int* size chunks. Likewise, there's no saving in writing a function which accepts *char* values over *int*s as arguments, nor one which returns them.

Let's look a bit further into this. This allocation takes up one sixteen bit position on the stack, as we've seen.

```
int a;
```

What about this one?

```
int *a;
```

In this second case, *a* is a pointer to an *int*. The question, then is, how much space a pointer takes up. The answer... depends.

On the PC, a location in memory is pointed to by two numbers. The first one is called the "offset". This tells us the local position of the number. In fact, it's relative to the current "segment" value, which is the second of the two numbers.

The memory in a PC is "segmented". This is a bit nasty. It happened for the following reason... more or less. When the 8088 chip which drove the first PC was designed, the trolls in marketing thought that it would be nice if it could access a megabyte of memory, which, in those days seemed like an awful lot. Unfortunately... said the engineers... the chip only had sixteen bit registers, which even a troll from marketing could see would only access sixty-four kilobytes. Actually, the trolls from marketing couldn't see this, 'coz they didn't know a register from an RRSP.

The engineers came up with a solution to this. They combined two sixteen bit registers to form one bigger twenty bit register. Yes, I know... when I went to school a couple of sixteens was good for thirty-two as well, but they only needed twenty bits to address all that memory. However, just to get back at those marketing suits, they used the upper bits in a sort of weird way. They divided a megabyte by sixty-four kilobytes and called the resulting number... sixteen... a segment. The upper part of the number, then, would be the number of sixteen byte segments in the address, and the lower part of the number would be the address in the current segment. A segment encompasses the sixty-four kilobytes directly above it.

A pointer under C, then, can be of several types. If the program and all its data will fit in a single sixty-four kilobyte segment, we can use what is called a "small model" for the program and all the

pointers will be sixteen bits wide... one place on the stack. If the program will fit in one segment but the data for the program will not, then we must use the "medium model", which means that the pointers must be thirty-two bits wide, although all the calls to functions can be done with sixteen bits, which saves one place on the stack for each call. This can be handled the other way around under some C compilers... sixteen bit pointers and thirty-two bit calls for big programs with small amounts of data.

Under the large memory model, both pointers and function calls are handled as thirty-two bit numbers. Most programming is done under this memory model. There is another memory model, the "huge" model, which we'll speak of at another time.

The advantage of the small model is that its programs take up less space and run faster. The advantage of the large model is that its programs and their resulting data can be bigger than a single segment.

The great thing about C is that you can change memory models... in most cases... by just telling your compiler that you want to. In the case of Turbo C, it's a single menu option. Change from the small model to the large model and on the next compile everything will be adjusted for you. You never have to worry about how much space to allow on the stack and so on.

Flapjacks

If you understand the fundamentals of stack manipulation under C, you will be a lot closer to understanding how the language works. You will also understand how to avoid a lot of the potential problems which C programs are heir to, because a lot of them involve misuses of the underlying stack structure of C.

In addition, as you get a bit more advanced in C, you'll learn how to optimize your programs by thinking about how the code you write interacts with the real time world of the processor that, ultimately, makes everything go.

To finish our discussion of the C language stack off with something interesting, you might want to check out the following little program. This is written in Turbo C, and uses the Turbo graphics library. Other compilers might require a few trivial changes to the code to make it compile. It generates the accompanying picture when it runs, and you can change the default parameters by giving the program command line arguments to see how the picture changes.

The important part about this program, though... aside form the fact that it creates pretty pictures... is how it uses its stack. Recalling that we said that calling a function causes its return address to be pushed up onto the stack... not to mention its local variables allocated there... what would you think would happen in a program where a function repeatedly calls itself? This is what happens in this program.

To make this go, type the source code into a file called curve.c. Create a second file called curve.prj, and type the following two lines into it.

```
curve
graphics.lib
```

Set the Turbo C project to CURVE.PRJ and compile the program. If all goes well, you'll see a text message and, upon hitting a key, the curve will start to form on your screen.

When you get tired of playing with the curves, you might go back to figuring out how this program is using the stack.

```
/*
C Curve generator
copyright (c) 1988,1989 Alchemy Mindworks Inc.
*/

#include "stdio.h"
#include "graphics.h"
#include "math.h"
#include "conio.h"

#defineTHETA45

double SINTH,COSTH;
double prm[5] = { 250,50,400,50,2 };

main(argc,argv)
int argc;
char *argv[];
{
int d,m,e = 0;

cprintf("C curve generator copyright (c)"
"1988, 1989 Alchemy Mindworks Inc.\n");
if(argc1) for(d = 1;dargc; + + d)
prm[e + + ] = atof(argv[d]);
cprintf("Parameters are %g %g %g %g RESOLUTION %g\n",
prm[0],prm[1],prm[2],prm[3],prm[4]
);
cprintf("Hit any key...");
if(getch() = = 27) exit(0);
init();

COSTH = cos(THETA*M_PI/180);
SINTH = sin(THETA*M_PI/180);

curve(prm[0],prm[1],prm[2],prm[3]);
getch();
deinit();
}

curve(x1,y1,x2,y2)/* draw one curve */
double x1,x2,y1,y2;
{
double xd,yd;
double mx,my;
double xdr,ydr;

xd = x2-x1;
yd = y2-y1;
if(((xd*xd) + (yd*yd))
(prm[4]*prm[4]) || kbhit())
line((int)x1,(int)y1,(int)x2,(int)y2);
else {
xdr = xd/2/COSTH;
ydr = yd/2/COSTH;
mx = x1 + xdr*COSTH-
ydr*SINTH;
my =
y1 + xdr*SINTH + ydr*COSTH;
curve(x1,y1,mx,my);/* it calls.... */
curve(mx,my,x2,y2);/* ... itself!!! */
}
}


init()/* get into graphics mode */
{
int d,m,e = 0;

detectgraph(&d,&m);
if(d) {
puts("No graphics card");
exit(1);
}
printf("d = %d\n",d);
initgraph(&d,&m,"");
e = graphresult();
if(e) {
printf("Graphics error %d: %s",
e,grapherrormsg(e));
exit(1);
}
setcolor(getmaxcolor());
}

deinit()/* get out of graphics mode */
{
closegraph();
}
```

■