Memory and The Stack!

Memory

So far we've talked about how c works via compilation, and how to write some files that can read and write data to and from standard in/out and files. What we haven't talked about though is what's really happening behind the scenes in memory. Computers would be nothing without memory at all! When we say "memory" what we mean can refer to a number of different things, though. Typically, though people are referring to either *volatile* or *non-volatile* storage media.

Volatile and *non-volatile* don't really mean much on their own, however, but ultimately think of `volatile` memory like "short-term" memory, where, if you turned your computer off, that data would just disappear, and *non-volatile* memory lasting through no power. *Volatile* memory refers to your random access memory, whereas *non-volatile* refers to longer term storage mechanisms like a hard drive (SSD, disc, flash, etc).

A lot goes on in an operating system to coordinate between *volatile* and *non-volatile* memory. If a program is running, and decides it needs data that exist in the file system, stored in *non-volatile* memory, the operating system work its magic and extract that data out of the *non-volatile* memory and create a copy of it in *volatile* memory.

When we work with this data we're working with everything in *volatile* memory that has an associated "address". So, in a program, suppose we create an `int a = 23;` and then our immediate next command is `char b = 'f';`. What will happen in memory is, as the computer's running our program, and many other programs simultaneously, it's assigning these things to spaces in memory, so we might get our *volatile* memory in the computer looking like (we're going to use the same terminology as Intermediate C Programming):

Symbol	Address	Value
a	7	23
b	32	f

So what's happening is that, as our program is running, the first line is read, and assigns `a` to address 7 with its value of 23. However, in the meantime, because of the multiprocessing and multiprogramming environment of our linux operating system, other programs are also running, taking up memory locations. When our program gets its turn back on the processors, it then runs the next command and assigns `b` to the address of 32 because other addresses between 7 and 32 have already been taken up, and stores the value of `f`. So, don't be alarmed if when debugging (in the coming module), you see addresses in non consecutive spots.

There are multiple types of volatile memory that our programs work with: `stack`, `heap`, and `program`. `stack` and `heap` memory are there to store data from the program, while `program` memory is there for the program itself. In this chapter, we're going to talk about `stack` memory, and we'll venture into `heap` memory in the following module.

The Stack

If you've taken a data structures class before, you've likely heard of a stack. A stack is a data structure that has a last in first out structure, where data are "pushed" onto the stack, and then "popped" off the top. If you're having trouble conceiving of how a stack works, think of clothing. On a cold day, you're likely to put on a shirt and then a coat before going outside. You "push" the shirt, then you "push" the coat. When you get to wherever you were going, you then remove your coat first, or "pop" the coat. You don't remove your shirt before removing your coat.

So how do stacks work with programming? The stack is there for our program when it runs. Let's write a very simple program and then discuss how the stack is working:

`simpleStack.c`

```c
#include <stdio.h>
int main( ) {
  int a = 23;
  char b = 'f';
  double p = 3.14;

  return 0;
}
```

If we were to turn this program, what would happen is that we'd wind up with a stack that looks like:

Symbol	Address	Value
p	49	3.14
b	32	f
a	7	23

Again, it should be noted that the addresses are completely arbitrary since they're controlled by the operating system. When the program executes, it goes line by line, so we see `a`, we then add that to the stack, alone:

Symbol	Address	Value
a	7	23

Then, in moving to the next line, we see char `b`, which then, gets added to the stack:

Symbol	Address	Value
b	32	f
a	7	23

It gets put on top of a. Then, we see double `p`, which, just like `b`, gets pushed onto the top of the stack:

Symbol	Address	Value
p	49	3.14
b	32	f
a	7	23

Then, when our program ends, each element is popped off of the stack one by one!

Functions and the Stack

That simple example above is all fine and dandy, but what happens when there are functions? How does the computer know what to add where? What about scoping? What if I have a variable named `a` inside of a function? Let's examine the program

`justOneFunc.c`

```c
#include <stdio.h>

void someFunc() {
  int a = 10;
}

int main() {
  int a = 23;
  char b = 'f';
  double p = 3.14;

  someFunc();

  return 0;
}
```

What happens to our stack as we start to add functions? How does the computer know where to go? When a function, think of it as having its own little special area in the stack. For example, imagine during the execution of the above program we just got to line 12. When we execute some func, we're hopping to an entirely different area of the code, but what's happening in the stack is that the function is being added to the stack, and it has a place that tells it where to return! For now, just recognize that the "frame" is just referring to whatever function we're in:

Frame	Symbol	Address	Value
someFunc	Return location	51	line 12
main	p	49	3.14
main	b	32	f
main	a	7	23

Next, let's see what happens when we execute the next line (which would be line 4 inside someFunc):

Frame	Symbol	Address	Value
someFunc	a	57	10
someFunc	Return location	51	line 12
main	p	49	3.14
main	b	32	f
main	a	7	23

So in the above stack, we've got two `a`s, but they're in different frames of reference! What happens once the function finishes? It will pop `a`, and then look at the return location, and then the code will go to that spot to resume execution!

But what if our program looked slightly different and looked like:

```
1  #include <stdio.h>
2
3  void someFunc() {
4    int a = 10;
5  }
6
7  int main() {
8    int a = 23;
9    char b = 'f';
10
```

```
11    someFunc();
12
13    double p = 3.14;
14
15    return 0;
16  }
```

We would ultimately wind up with a stack that followed this pattern, starting at the execution of line 11:

Frame	Symbol	Address	Value
someFunc	Return location	51	line 11
main	b	32	f
main	a	7	23

Move to line 4 inside someFunc:

Frame	Symbol	Address	Value
someFunc	a	57	10
someFunc	Return location	51	line 11
main	b	32	f
main	a	7	23

End of someFunc, pop off a and go back to the return location:

Frame	Symbol	Address	Value
main	b	32	f
main	a	7	23

Move to line 13:

Frame	Symbol	Address	Value
main	p	61	3.14
main	b	32	f
main	a	7	23

And so on.

Functions with Arguments:

When you have a function with parameters, when it gets called, what happens is that those parameters are each pushed to the stack immediately, in the order they appear. Let's add a couple parameters to `someFunc`:

```
1  #include <stdio.h>
2
3  void someFunc(int x, char z) {
4    int a = 10;
5  }
6
7  int main() {
8    int a = 23;
9    char b = 'f';
10   double p = 3.14;
11
12   someFunc(42, 'i');
13
14   return 0;
15 }
```

When we go to execute this function, our call stack will look something along the lines of:

Frame	Symbol	Address	Value
someFunc	z	62	i
someFunc	x	57	42
someFunc	Return location	51	line 12
main	p	49	3.14
main	b	32	f
main	a	7	23

Wen we passed the data in for the parameters `x` and `z`, we get those values stored as the parameters' value in the stack! This program will then continue to execute by adding `someFunc`'s `a` to the stack, and then ultimately popping all of the `someFunc` data off the tack, continuing the program.

Functions with Return Values

If a function has a return value, then there's one additional value added to the call stack: the `value address`. Suppose we wanted `someFunc` to return the `a + x`:

```
1  #include <stdio.h>
2
3  int someFunc(int x, char z) {
4    int a = 10;
5
6    return a + x;
7  }
8
9  int main() {
10   int a = 23;
11   char b = 'f';
12   double p = 3.14;
13
14   int f = someFunc(42, 'i');
15
16   return 0;
17 }
```

When we first execute the function someFunc, what we'll see is:

Frame	Symbol	Address	Value
someFunc	Value Address	63	51
someFunc	Return location	58	line 15
main	f	51	garbage
main	p	49	3.14
main	b	32	f
main	a	7	23

What's happening is that we created an integer f, and we want to assign it to the response of function `someFunc`. However, since `someFunc` isn't finished executing yet, it's currently just garbage data. Now, we then add our function to the call stack with our return location of line 14, but then we have this new thing: a value address with its stored value as the actual address of the variable that we want to store our returned data in! The next steps are to continue adding everything to the stack:

Frame	Symbol	Address	Value
someFunc	a	82	10
someFunc	z	78	i
someFunc	x	74	42
someFunc	Value Address	63	51
someFunc	Return location	58	line 15
main	f	51	garbage
main	p	49	3.14
main	b	32	f
main	a	7	23

Once we finish the executions of the function, we then take the result of the executed return value (i.e. `a + x`) and assign it to where the value address is pointing: `f` (10 + 42 = 52):

Frame	Symbol	Address	Value
main	f	51	52
main	p	49	3.14
main	b	32	f
main	a	7	23

Multiple Functions

Finally, there's a question of what happens if there are multiple functions? The same thing that happens with just two! We keep just adding functions to the stack!

```
1  #include <stdio.h>
2
3  void yetAnotherFunc() {
4
5  }
6
7  void anotherFunc(){
8    yetAnotherFunc();
```

```
}

void someFunc(int x, char z) {
  int a = 10;
  anotherFunc();
}

int main() {
  int a = 23;
  char b = 'f';
  double p = 3.14;

  someFunc(42, 'i');

  return 0;
}
```

In the above program, we have a similar structure to what we've already had, however, we've added a couple of functions that are called from within other functions! This happens all the time in code, but how exactly does it work? Let's trace it out!

Frame	Symbol	Address	Value
anotherFunc	Return location	72	line 13
someFunc	a	67	10
someFunc	z	62	i
someFunc	x	57	42
someFunc	Return location	51	line 21
main	p	49	3.14
main	b	32	f
main	a	7	23

We, after calling `someFunc` and adding its values to the stack, then call `anotherFunc`. `anotherFunc` then calls `yetAnotherFunc`:

Frame	Symbol	Address	Value
yetAnotherFunc	Return location	77	line 8
anotherFunc	Return location	72	line 13
someFunc	a	67	10
someFunc	z	62	i
someFunc	x	57	42
someFunc	Return location	51	line 21
main	p	49	3.14
main	b	32	f
main	a	7	23

And once we finish with these functions, we pop them off the stack:

Frame	Symbol	Address	Value
anotherFunc	Return location	72	line 13
someFunc	a	67	10
someFunc	z	62	i
someFunc	x	57	42
someFunc	Return location	51	line 21
main	p	49	3.14
main	b	32	f
main	a	7	23

Pop again:

Frame	Symbol	Address	Value
someFunc	a	67	10
someFunc	z	62	i
someFunc	x	57	42
someFunc	Return location	51	line 21
main	p	49	3.14
main	b	32	f
main	a	7	23

Pop again:

Frame	Symbol	Address	Value
main	p	49	3.14
main	b	32	f
main	a	7	23