Pointers

Pointers can be scary. When you first encounter them, they're not entirely intuitive. This chapter will help demystify pointers, and hopefully make a bit more sense of them!

Scope

When we refer to scope, what we're referring to is everything with a series of curly braces `{...}`. Curly braces define a scope. Anything inside of the curly braces can see out, but anything on the outside of the curly braces can't see in. You could think of curly braces as the tinted windows of the code world!

Let's take a quick example of an `if` statement:

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv){
  int number = 20;
  if( number > 5 ){
    int k = number - 5;   // K is only in the scope of this if!
    printf("%d is still positive!", k);
  }
  // if we tried to access k here, the program would fail to compile since
  // k is no longer in scope. Try to uncomment and compile the next line:
  // printf(k);
  return EXIT_SUCCESS;
}
```

In the above code, we simply a basic program that checks to see if `number` is greater than 5, and if so, it subtracts 5 from that number. Not the most exciting program, however, it gets the point across.

We define number in a higher scope, so that means that, even though we enter a new scope (i.e. we enter into the `{...}` of the `if` statement) we can still see number. However, once we leave the block of code with the `if`, if we were attempting to access `k`, the program would not compile since `k` is outside of the `main` function's scope!

Pointers

The term pointer can be scary, but all a pointer is is a numerical value to a memory location that acts as a reference. So instead of printing the *value* of a variable, you're instead printing the *address*. Suppose you have the following code:

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int someNumber = 23;
  int * someNumbersAddress = & someNumber;

  printf("%d someNumber\n%d someNumbersddress\n", someNumber,
someNumbersAddress);
  return EXIT_SUCCESS;
}
```

When we then try to access `someNumber`, we'll see the value of `23`, but when we view `someNumbersAddress` we don't see 23, but instead we see the location where it's stored:

```
./a.out
```

> 23 someNumber 1023476452 someNumbersAddress

So what's happening in the stack trace would look something similar to:

Symbol	Address	Value
someNumbersAddress	13	7
someNumber	7	23

The value of `someNumbersAddress` is nothing more than a value pointing to the address of `someNumber`.

Extracting Values

To extract a value from a pointer, you want to use the `*` symbol before it:

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int someNumber = 23;
  int * someNumbersAddress = & someNumber;

  // Get the value from a pointer's reference:
  int thePointedValue = * someNumbersAddress;

  printf("%d someNumber\n%d someNumbersddress\n%d thePointedValue", someNumber, someNumbersAddress, thePointedValue);
  return EXIT_SUCCESS;
}
```

In the above, we're extracting the value of the address that `someNumbersAddress` is pointing to, and saving **a copy** of that value in `thePointedValue`:

```
./a.out
```

> 23 someNumber -20920176 someNumbersddress 23 thePointedValue

Saving Values

Just as you can get the value stored in an address from a pointer, you can also reassign values from a pointer:

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int someNumber = 23;
  int * someNumbersAddress = & someNumber;

  // Get the value from a pointer's reference:
  int thePointedValue = * someNumbersAddress;


  // Reassign the value at the address *someNumbersAddress
  * someNumbersAddress = 42;

  printf("%d someNumber\n%d someNumbersddress\n%d thePointedValue", someNumber, someNumbersAddress, thePointedValue);
```

```
  return EXIT_SUCCESS;
}
```

When we turn the program, notice the how the value changed at `someNumber`, but not at `thePointedValue`:

```
./a.out
```

> 42 someNumber
>
> -1175470128 someNumbersddress
>
> 23 thePointedValue

We reassigned the value stored in the address of `someNumber` to 42. Naturally, because that's where `someNumber` was stored, the value itself changed! But look at `thePointedValue`. Why didn't that change? It didn't change because it's memory location was not overwritten. `thePointedValue` has a different address and is just a copy of the value stored by `someNumber`.

Functions and Pointers

Suppose you wanted your function to pass back more than one thing? Unless you're attempting to pass back an array of values, you're best bet is to use a pointer to act as a secondary return value! Take a look at the following code:

`pointer2.c`

```
#include <stdio.h>
#include <stdlib.h>

int getTwoValues(int a, int b, int * returnValue1Address){
  int c = a + b;
  int d = a * b;
  * returnValue1Address = c

   return d;
}

int main(int argc, char** argv) {
  int num1 = 10;
  int num2 = 20;

  int returnValue1 = 0;
  int returnValue2 = 0;
```

```
18
19    int * returnVal1Address = & returnValue1;
20
21    returnValue2 = getTwoValues(num1, num2, returnVal1Address);
22
23    printf("Return values 1 and 2 are:\nreturnValue1: %d\nreturnValue2: %d",
    returnValue1, returnValue2);
24
25    return EXIT_SUCCESS;
26  }
```

What's happening above? Take a look at the output:

```
1  ./a.out
```

> Return values 1 and 2 are:
>
> returnValue1: 30
>
> returnValue2: 200

What happened above is that we passed in 3 value to the function `getTwoValues`. The first two values were just numbers that we wanted to add and multiply. The third argument, however, is a pointer to the address of `returnValue1`. By passing in that pointer, we can directly access that memory location! Let's look at the stack trace via `GDB`:

```
1  (gdb) break 6
```

> Breakpoint 1 at 0x400546: file pointer2.c, line 6.

```
1  (gdb) break 19
```

> Breakpoint 2 at 0x400589: file pointer2.c, line 19.

In the above we set two break points, one inside of our function, and one inside of our main.

```
1  (gdb) run
```

> Starting program: /home/mjlny2/2750Materials/module7/pointers/a.out
>
> Breakpoint 2, main (argc=1, argv=0x7fffffffe308) at pointer2.c:19 19 int * returnVal1Address = & returnValue1;

```
(gdb) bt
(step)
(gdb) bt full
```

> #0 main (argc=1, argv=0x7fffffffe308) at pointer2.c:21 num1 = 10 num2 = 20 returnValue1 = 0 returnValue2 = 0 returnVal1Address = 0x7fffffffe204

We stopped first where we assigned our pointer to an address. We then printed the full stack! See `returnVal1Address = 0x7fffffffe204`? That's the address where the `0` is stored for returnValue1!

```
(gdb) cont
```

> Continuing.
>
> Breakpoint 1, getTwoValues (a=10, b=20, returnValue1addr=0x7fffffffe204) at pointer2.c:6
>
> 6 int d = a * b;

By typing `cont`, we don't have to bother stepping to our next breakpoint! We'll just stop when the code reaches that execution point. Notice that we're seeing the function signature of `getTwoValues` and how it's variable `returnValue1addr` has the exact same address as our `returnVal1Address = 0x7fffffffe204`?

```
(gdb) step
```

> 7 * returnValue1addr = c;

Assign the value of `c` to `* returnValue1addr`

```
(gdb) step
```

> 9 return d;

```
(gdb) bt full
```

> #0 getTwoValues (a=10, b=20, returnValue1addr=0x7fffffffe204) at pointer2.c:9
>
> c = 30
>
> d = 200
>
> #1 0x00000000004005a4 in main (argc=1, argv=0x7fffffffe308) at pointer2.c:21
>
> num1 = 10
>
> num2 = 20
>
> returnValue1 = 30
>
> returnValue2 = 0
>
> returnVal1Address = 0x7fffffffe204

We then step over the code, and reach to the near end of the function! Notice that in our main, the value of `returnValue1` has changed to 30! That's because our function is directly accessing that memory space, and because we haven't actually returned from the function yet with the value from `returnValue2`, its value is still at its initialized zero!

```
(gdb) step
```

> 10 }

```
(gdb) step
```

> main (argc=1, argv=0x7fffffffe308) at pointer2.c:23 23 printf("Return values 1 and 2 are:\nreturnValue1: %d\nreturnValue2: %d", returnValue1, returnValue2);

```
(gdb) bt full
```

> #0 main (argc=1, argv=0x7fffffffe308) at pointer2.c:23
>
> num1 = 10
>
> num2 = 20
>
> returnValue1 = 30
>
> returnValue2 = 200
>
> returnVal1Address = 0x7fffffffe204

Finally, we've returned to our main function fully, the function `getTwoValues` has been popped off of the stack, and we have successfully returned two values from our function with a pointer!

Type Errors

Be very careful with pointers. Assigning a pointer's data directly to a non-pointer data-type or attempting to point pointers of different types to each other can be dangerous. Be explicit with your naming conventions with pointers, so that you don't accidentally attempt to read a double's value into an integer!

Arrays and Pointers

You've likely worked with arrays before! An array is nothing more than a container of values that we can access as a single variable with indices! Behind the scenes, though, arrays aren't just buckets that store data, they're really just pointers!

Let's consider the code:

`basicArray.c`

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  const int ARR_SIZE = 5;
  int arr [ARR_SIZE];

  int i;
  for(i = 0; i < ARR_SIZE; i++ ) {
    arr[i] = i;
  }

  return EXIT_SUCCESS;
}
```

Let's compile this and run it through `gdb`:

```
gcc -ggdb basicArray.c
gdb a.out
```

Now let's add a couple breakpoints, and run:

```
(gdb) break 8
(gdb) break 12
(gdb) run
```

> Starting program: /home/mjlny2/2750Materials/module7/pointers/a.out
>
> Breakpoint 1, main (argc=1, argv=0x7fffffffe308) at basicArray.c:9
>
> 9 for(i = 0; i < ARR_SIZE; i++) {

```
(gdb) bt full
```

> #0 main (argc=1, argv=0x7fffffffe308) at basicArray.c:9
>
> ARR_SIZE = 5
>
> arr = {0, 0, 0, 0, 0}
>
> i = 0

We lucked out in that our array is filled with all 0s, however, that's not always the case. Since we're immediately overwriting our array in the next loop, we'll be ok! Let's continue to the next break point

```
(gdb) cont
```

Now that we're at the next breakpoint, let's look at the contents of our array:

```
(gdb) print *arr@5
```

> $1 = {0, 1, 2, 3, 4}

What we did above was print the array's contents, where 5 is referring to the length!

Let's now take a look at how this array works:

```
p/x arr
```

> $2 = {0x0, 0x1, 0x2, 0x3, 0x4}

All our arrays are doing is accessing the same memory location, just with a slight offset! `arr[0]` is the starting point, `arr[1]` is right next to `arr[0]`, just displaced by 1 space (that space is however many bytes the datatype is).

Arrays and Functions

Because arrays ultimately just pointers, they are also able to be accessed directly! This is what people refer to as "pass by reference". Consider the program:

`passByRef.c`:

```c
#include <stdio.h>
#include <stdlib.h>

void changeAValue(int array[], int SIZE){
  int i;
  for (i=0; i<SIZE; i++){
    array[i] = 42;
  }
}

void printArray(int array[], int SIZE){
  int i;
  for (i=0; i<SIZE; i++){
    printf("%d",array[i]);
  }
}

int main(int argc, char** argv) {
  const int ARR_SIZE = 5;
  int arr [ARR_SIZE];
  int * arrPointer = arr;

  int i;
  for(i = 0; i < ARR_SIZE; i++ ) {
    arr[i] = i;
  }

  printf("initialized: \n");
  printArray(arr, ARR_SIZE);

  printf("passing by reference to changeAValue");
  changeAValue(arr, ARR_SIZE);
  printArray(arr, ARR_SIZE);


  return EXIT_SUCCESS;
}
```

When running this code through gdb to see the stack memory locations, we can see:

```
(gdb) break changeAValue
(gdb) run
(gdb) bt full
```

> #0 changeAValue (array=0x7fffffffe1c0, SIZE=5) at shit.c:6 i = 32767 #1 0x00000000004006f2 in main (argc=1, argv=0x7fffffffe308) at shit.c:32 ARR_SIZE = 5 arr = {0, 1, 2, 3, 4} arrPointer = 0x7fffffffe1c0 i = 5

Above, notice that the arrPointer is pointing to the exact same location as `array` in the `changeAValue` value? This is because arrays are passed by reference, so to mutate them in a function will mutate them in real life!

```
./a.out
```

> initialized: 0 1 2 3 4 passing by reference to changeAValue 42 42 42 42 42

Type Rules

All pointers must refer to the type of what they're pointing to!

Pointer Arithmetic

Because arrays are pointing to a memory location behind the scences, you may be wondering how they're storing data! An array allocates memroy,

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
    const int ARR_SIZE = 5;
  int arr [ARR_SIZE];
  int * arrPointer = arr;

  int i;
  for(i = 0; i < ARR_SIZE; i++ ) {
    arr[i] = i;
  }

  for(i = 0; i < ARR_SIZE; i++ ) {
    printf("%d \n", *arrPointer);
    arrPointer++;
  }

```

```
19
20    return EXIT_SUCCESS;
21  }
```

Because arrays are just pointing to a location in memory, we can use our pointer to increment up through the memory locations and grab each element by incrementing it! This works because our pointer is the same data type! If we had a pointer to a character, on the other hand, we'd need to multiply it by 4 because characters are only 1 byte, whereas integers are 4 (but again, mixing pointer types can be very dangerous)!

```
1  gcc pointerMath.c
2  ./a.out
```

```
0 1 2 3 4
```