CS 2110 Lab 22
Function Pointers

The TAs :)

November 12, 2018

1 Recitation Outline

1. Function Pointers
   (a) Helpful Reference(s)
      i. Function Pointers in C
   (b) Motivation
      i. Function pointers are helpful!
   (c) Syntax Breakdown (example, writing `sort`)
      i. Writing function pointers
      ii. Setting function pointers
      iii. Calling function pointers
   (d) qsort and void pointers
      i. Difference between our `sort` and `qsort`
      ii. Don't dereference void pointers!

2 Function Pointers

2.1 Helpful Reference(s)

The syntax for function pointers can be pretty difficult to understand on your first time around (or your second or third or tenth). When I was first learning it, reading this write-up helped me alot, so I'm probably going to be stealing some things from this for this write up. I greatly recommend that you give it a read:

Function Pointers in C

2.2 Introduction and Motivation

So in C, you can pass pointers to your variables and data structures between functions. Since a pointer is just a memory address though, we can also pass around pointers to *functions* seeing as how functions get loaded into memory when our program runs. This means that our functions can accept other functions as parameters to change their behavior at runtime. Do you remember lambdas from 1331? Well, this is pretty similar. Additionally, this means that structs can have members that are function pointers, which allows us to implement some object oriented programming concepts (i.e this basically lets structs have their own methods that can be dynamically set at runtime (read dynamic method binding)).

To illustrate how function pointers work, we are going to show you how to write the function signature for a `sort` function that accepts a comparator as a function pointer. The comparator will dictate how the `int`s in the array are compared.

2.3 Syntax Breakdown

To start off, we want to write a function that will sort an array of `int`s that also takes in a function pointer that tells us how to sort. Logically then the prototype will look something like this:

```
void sort(int *array, int size, /* FUNCTION POINTER */);
```

Now, let's think about how we want our comparator function to actually behave. As you might remember from 1331, a comparator function will take in two elements and return an `int` (negative for less than, zero for equal to, and positive for greater than).

So then, the signature for a valid `int` comparator should look something like this:

```
int compare(int a, int b)
```

Now, we know what the signature for a comparator looks like, but we want to declare a variable that's a pointer to such a function! Interestingly, it is as simple as doing this:

```
int (*comparator)(int, int);
```

As a side-by-side comparison so you can see the differences:

```
int compare(int a, int b); // Declaring a function called compare that takes two ints and
                           // returns an int.
int (*comparator)(int, int); // Declaring a variable called comparator that points to a
                             // function that takes two ints and returns an int.
```

All we really needed to do was add that extra star to denote that `comparator` is a pointer, surround the name in parentheses, and we're good to go. There's no need to have the parameter names so we leave them off.

That means the function signature for our `sort` function will look like this:

```
void sort(int *array, int size, int (*comparator)(int, int))
```

Now, how would we actually call this `sort` function? Assuming we want to use the `compare` function from the side-by-side comparison, we only need to do:

```
sort(myArray, myArraySize, compare);
```

See how we only needed to pass in the function name? It turns out that the name of the function acts as a pointer to it. Interestingly, you could also do this to get the exact same result:

```
sort(myArray, myArraySize, &compare);
```

In the case where you are dealing with functions, the "address of" operator is an identity operation.

Now, how would we actually go about invoking our function pointer within our `sort` function? Remember how I said that the name of a function acts as a pointer to it? Well, that basically means that calling a function pointer is no different from calling any other function. Just give it the name. Here is our full code that uses our `sort` function (which is bubble sort):

```c
#include <stdio.h>

// Compare a and b in ascending order
int compare(int a, int b) {
    // If a > b return positive
    // If a < b return negative
    // If a == b return 0
    return a - b;
}

// Bonus! For sorting in descending order
int compareDescending(int a, int b) {
    // If a < b return positive
    // If a > b return negative
    // If a == b return 0
    return b - a;
}

// Our sort function!
void sort(int *array, int size, int (*comparator)(int, int)) {
    int done = 0;
    while (!done) {
        done = 1;
        for (int i = 0; i < size - 1; i++) {
            // Checks if array[i] > array[i + 1] (i.e they're out of order)
            if (comparator(array[i], array[i + 1]) > 0) {
                // Swap
                int temp = array[i];
                array[i] = array[i + 1];
                array[i + 1] = temp;
                done = 0;
            }
        }
    }
}

// Just a function to print our array out
void printArray(int *array, int size) {
    printf("[");
    for (int i = 0; i < size - 1; i++) {
        printf("%d, ", array[i]);
    }
    printf("%d]\n", array[size - 1]);
}

int main(void) {
    int size = 10;
    int array[10] = {7, 8, 1 ,3, 2, 9, 4, 0, 5, 6};
    printArray(array, size);
    sort(array, size, compare);
    printArray(array, size);
    sort(array, size, compareDescending);
    printArray(array, size);
}
```

2.4 qsort

There actually exists a function built into the C standard library called `qsort` that works very much like this function we just wrote. The main difference is that it is more generic (i.e it can sort anything, not just `int`s).

It achieves this by accepting an array of `void` pointers and having its comparator function compare `void` pointers instead of `int`s.

As a result, its signature (if you type `man qsort`) looks like this:

```
void qsort(void *base, // The array
           size_t nmemb, // The number of array elements
           size_t size, // The size of each element
           int (*compar)(const void *, const void *)); // Takes in two void pointers
                                                       // and returns an int
```

Whenever you see a `void` pointer, it just means a pointer to something of unknown type. Since we don't know the type, we can't dereference `void` pointers. Before doing so, you need to cast it to a pointer of another type. So, if we wanted to use our `compare` function with qsort, we would need to change it to look like this:

```
// Compare a and b in ascending order
// Note: the const just stops us from modifying the values at a and b
int compare(const void *a, const void *b) {
    int *castedA = (int *) a;
    int *castedB = (int *) b;
    // If a > b return positive
    // If a < b return negative
    // If a == b return 0
    return *castedA - *castedB;
}
```

Our `compare` function makes an assumption that `a` and `b` are `int`s, which is fine if we're sorting an array of `int`s. Our call in main would just look like this:

```
qsort(array, size, sizeof(int), compare);
```