CS 2110: Lecitation 11

Name:

Monday, June 25th, 2018

Introduction

Today we will be exploring some concepts in the C programming language.

Make sure to ask the TA's questions!

Part A: Function Pointers

Objective: You will be using `qsort` and function pointers to sort an array of elements in different ways. If you take a look at the man page for `qsort`, you will see that the fourth parameter to `qsort` looks like this:

```
int (*compar)(const void *, const void *)
```

The type of this parameter is a function pointer, which is the address of a function. The signature of the function must match `int` as the return type, and `(const void*, const void*)` as the parameter list. You can pass in the address of a function by simply using that function's name as the parameter. For example, here is the signature of the `atexit` function:

```
int atexit(void (*)(void));
```

The `atexit` function takes in a function pointer to a function which will be executed on normal termination of the program. The function must be a `void` function with no parameters:

```
void bye(void) {
  printf("Goodbye!\n");
}

int main(void) {
  atexit(bye);
  // Do some stuff...
  return 0;
}
```

Assignment: In the provided `sort.c` file, edit where necessary to sort the array appropriately. Use `qsort` from `stdlib.h` to accomplish this. Use `make` to build the application and `./sort` to run it.

Hints: Use the `man` pages to see how `qsort` is used!

For the second sort: You should sort the elements alphabetically. You don't have to write low-level string comparison code to accomplish this because there's already a function in `string.h` that can compare strings. It is recommended that you use that function in your implementation of this lab.

For the third sort: You should sort the elements by length, with the requirement that names of the same length are then sorted by name. Try calling the function you wrote for the second sort!

Part B: Shallow vs. Deep Copy

Objective: Learn about problems you may encounter with pointer use, and understand the distinction between shallow and deep copies.

Assignment: Reference the provided template file, and complete with the appropriate code. When you are finished, build the code into an executable with `make` and run it with `./copy`.

Hints: `sizeof` does **NOT** give you the number of elements in an array, but it can be used to get the number of elements in an array. Google how to do it!

Remember from HW07 what happens when a function is called: Parameters are copied onto the stack, so modifying those parameters in the function will only change their values within that function call. You cannot simply pass in `int`'s to a `swap` function. You'll need to pass in something else if you want to change the original `int`'s.

Likewise, if you have a pointer to some data and you copy the value of that pointer, then you now have two pointers to the same data. What do you think will happen if you change the value that one pointer points to, then try to read the value using the other pointer?

Part C: Dynamic Memory Allocation

Objective: You will complete a program which takes in several numbers from the user, then outputs their average. The numbers are separated by spaces, and they stop when the user presses `<enter>`. Example:

```
make avg
23 91 46 455 5<enter>
Average: 124
```

The structure for `main`, as well as the function to read numbers from the console, have been written for you.

Assignment: Your job is to write the code that handles dynamically resizing the array that holds the numbers the user is typing. You may **NOT** use `realloc` for this. See the man pages for:

```
malloc
free
memcpy
```

Testing: Your implementation should not have any memory leaks. To check this, first install `valgrind` if you haven't already done so:

```
sudo apt-get install valgrind
```

Then run the `val` target in the given `Makefile`:

```
make val
```

Of course, you have to enter some numbers as well for the program. It should say:

```
"All heap blocks were freed -- no leaks are possible"
```

Otherwise you have a memory leak which means you `malloc`'d, but didn't `free` it before you returned from `main`. There should not be any memory errors like invalid reads. Try some sequence of numbers. The average should be correct, and there should be no memory leaks or memory errors.

Hints: If you get an error for invalid read with `memcpy`, make sure you are not copying too many values from the old number array into the new array. Copying too many values may not produce a `segfault`, but reading out of the bounds of your `malloc`'d memory block is a memory error and would be penalized on homework.