ECE/CS 250 – Prof. Bletsch
Recitation #2 – C

Objective: In this recitation, you will learn how to write, compile, and run simple C programs.

Complete as much of this as you can during recitation. If you run out of time, please complete the rest at home.

PART 1

1. Basic I/O and Type Casts

Write a program that asks the user to input the name of his/her favorite Duke basketball player, the player's height in inches (this height should be an integer), and the player's average number of points scored per game (this last input should be an integer). The program should then output the player's name followed by "scored an average of X points per inch", where X is the average points divided by the height in inches. IMPORTANT: X must be a floating point number, which means you must do some type casting to compute it.

2. Functions and Structs

Write a program that uses a two-element struct called HoopsPlayer. This struct has two elements: an `int` for the player's number, and a `float` for his/her average points per game. Write a program that in `main()` loops and, in each loop, asks the user to input the info for one player (ask for an int then ask for a float). If the user inputs "-1" for the player number, then the loop ends (without asking for a float). For simplicity, you may assume you'll get at most 10 players. After the loop ends, `main()` must call a function called `sortList(<args>)` that prints the list of players sorted in ascending order of points per game. IMPORTANT: for `sortList()` to work, you have to figure out how to have `sortList()` get access to the list. A simple way is to declare the list as a global variable. More sophisticated programmers can pass a pointer instead.

3. Play with Strings

On Homework 2, you'll be re-implementing the programs from Homework 1, except in assembly language. As such, you won't have the built-in functions of C, especially those that help with strings (`strlen`, `strcpy`, etc.). Let's practice manual string manipulation in C. Write a program `arglen.c` that prints out the *length* of the first command line argument (`argv[1]`), or a usage message if no argument is given. **Do so without using the `strlen()` function; instead you should implement your own version called `my_strlen()`.** Example usage:

```
$ ./arglen
Syntax: arglen <string>

$ ./arglen hi
2

$ ./arglen architecture
12
```

4. Pointers Are Fun

Write a program that, in main(), declares an array of 100 ints and sets them to the values 0 through 99. (That is myArray[x]=x.) Have main() pass a pointer to this array to a function called sumArray(int* ptr) that returns the sum of all the entries in the array.

END OF PART 1

If you're in recitation now, please continue and dig into PART 2 until recitation ends.

PART 2

5. The Joy of Seg Faults

Part (a): Write a program that mis-uses pointers such that it causes a segmentation fault. (Enjoy being asked to do something wrong!) **Be sure to use the -g flag when you compile** to include debugging info in your binary – without this, you won't be able to determine which line of code caused the fault!

Part (b): Identify the exact line causing the fault using **gdb** (a command-line debugger that lets you step through programs line by line and can report the cause of a segfault) Below is an example where typed commands are blue and evidence of the fault is red.

```
$ g++ -g -o bad bad.c
$ ./bad
Segmentation fault (core dumped)
$ gdb bad
GNU gdb (Ubuntu 8.1-0ubuntu3) 8.1.0.20180409-git
  ...
Reading symbols from bad...done.
(gdb) run
Starting program: /home/tkb13/bad

Program received signal SIGSEGV, Segmentation fault.
0x0000000008000605 in some_function () at bad.c:4
4          *p = 55;
(gdb) bt
#0  0x0000000008000605 in some_function () at bad.c:4
#1  0x0000000008000614 in main () at bad.c:8
(gdb) print p
$1 = 0x0
(gdb) list
1       char* p;
2
3       void some_function() {
4         *p = 55;
5       }
6
7       int main() {
8         some_function();
9       }
10
(gdb) quit
A debugging session is active.

        Inferior 1 [process 1819] will be killed.

Quit anyway? (y or n) y
```

Look up each of the gbd commands used…what do they do? Now use them to diagnose *your* example segfaulting program from part (a).

Part (c): Identify the exact line causing the fault using **`valgrind`** (a memory access checking tool used to detect memory leaks and pointer errors). Unlike gdb, it's not an interactive program, but rather just reports on how your program uses memory. Below is an example where typed commands are blue and evidence of the fault is red.

```
$ valgrind ./bad
==22035== Memcheck, a memory error detector
==22035== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
==22035== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==22035== Command: ./bad
==22035==
==22035== Use of uninitialised value of size 8
==22035==    at 0x4004BC: main (bad.c:3)
==22035==
==22035== Invalid write of size 1
==22035==    at 0x4004BC: main (bad.c:3)
==22035==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
==22035==
==22035==
==22035== Process terminating with default action of signal 11 (SIGSEGV)
==22035==  Access not within mapped region at address 0x0
==22035==    at 0x4004BC: main (bad.c:3)
                                                ...
==22035== For counts of detected and suppressed errors, rerun with: -v
==22035== Use --track-origins=yes to see where uninitialised values come from
==22035== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 2 from 2)
Segmentation fault (core dumped)
```

Learn to read what the tool is telling you!! Referring to the code dump from gdb in part (b), think about the following questions; discuss with a peer or instructor if you're not sure.

- What line of code triggered the segmentation fault?
- What variable is the "uninitialised value of size 8"?
- Why does line 3 constitute an "Invalid write of size 1" (particularly, why is it a write, and why is size 1)?
- What address is `p` set to? How should we fix this program?

Now run valgrind in this way on your own broken program from part (a).

In C programming, your debugging tools are your eyes – you'll be blind if you don't learn to use them!

6. Catching Memory Leaks with Valgrind

Aside from helping to diagnose pointer mistakes, valgrind can also detect if heap memory allocated with `malloc()` and `calloc()` aren't being freed with `free()`. Consider the simple program **`leaky.c`** (linked on the course site). It allocates an array of integers, initializes it, then sums it.

Compile it (with `-g` of course!), then run it.

Now run it with valgrind. Look at the part of the output labeled "LEAK SUMMARY". Are any allocated blocks of memory lost? How many blocks? How big?

We want to know *where* we're leaking memory, or more precisely, what malloc/calloc call returns a chunk of memory that doesn't get freed later on. Valgrind can track this for us.

Re-run valgrind on the program with the `--leak-check=full` option. What part of the valgrind output tells you the line of code that did the un-freed allocation?

Fix the program by freeing the memory, then re-test it to confirm that all allocations are now freed.

7. Malloc/Free and Linked Lists

We're going to re-do task 2, but with `malloc`/`free`. Write a program that uses a two-element `struct` called HoopsPlayer. This `struct` has three elements: an `int` for the player's number, and a `float` for his average points per game, and an appropriate next pointer. Write a program that in `main()` loops and, in each loop, asks the user to input the info for one player (ask for an int then ask for a float). If the user inputs "-1" for the player number, then the loop ends (without asking for a float). You may NOT assume any limit on the number of players entered and thus you MUST use malloc to allocate new entries in the list. After the loop ends, `main()` must call a function called sortList(<args>) that prints the list of players sorted in ascending order of points per game. IMPORTANT: main() must pass a pointer to sortList(). You may not use global variables for this purpose.

During this task, I highly encourage you to run your program through the debugger gdb. Even if your program works fine, you should get used to using gdb. **Remember that you have to compile with the `-g` flag to create a binary for use with gdb.**

NOTE: This recitation introduced **`gdb`**, **`valgrind`**, and compiling with the **`-g`** option. Make sure you understand all of these things, as they're hugely useful to coding in C.

ALL DONE?

Nice! Don't head out, though. Work on the current homework and talk to the TAs for help. You can leave if your homework tester shows all passing and you've turned in all the written & code materials.