Lab 1

Deadline: Monday, June 29th

Setup

To get the starter files for this lab, run the following command in your `labs` directory.

```
$ git pull starter master
```

If you get an error like the following:

```
fatal: 'starter' does not appear to be a git repository
fatal: Could not read from remote repository.
```

make sure to set the starter remote as follows:

```
$ git remote add starter https://github.com/61c-teach/su20-lab-starter.git
```

and run the original command again.

Objectives:

- TSWBAT ("**T**he **S**tudent **W**ill **B**e **A**ble **T**o") compile and run a C program on the EECS instructional computers and examine different types of control flow in C
- TSWB ("**T**he **S**tudent **W**ill **B**e") introduced to the C debugger and gain practical experience using gdb to debug C programs
- TSWBAT work with integers, characters, boolean expressions, and bitwise operators to simulate regular expressions in C.

Compiling and Running a C Program

In this lab, we will be using the command line program gcc to compile programs in C. The simplest way to run gcc is as follows:

```
$ gcc program.c
```

This compiles program.c into an executable file named a.out. If you've taken CS61B or have experience with Java, you can kinda think of gcc as the C equivalent of javac. This file can be run with the following command:

```
$ ./a.out
```

The executable file is `a.out`, so what the heck is the `./` for? Answer: when you want to execute an executable, you need to prepend a file path in order to distinguish your command from a command like python3. The dot refers to the "current directory." Incidentally, double dots (`..`) would refer to the directory one level up.

`gcc` has various command line options which you are encouraged to explore. In this lab, however, we will only be using `-o`, which is used to specify the name of the executable file that `gcc` creates. You can use the following commands to compile `program.c` into a program named program, and then run it. This is helpful if you don't want all of your executable files to be named `a.out`.

```
$ gcc -o program program.c
$ ./program
```

Exercise 1: See what you can C

In this exercise, we will see an example of preprocessor macro definitions. Macros can be a messy topic, but in general the way they work is that before a C file is compiled, all `define` macro constant names are replaced exactly with the value they refer to. In the scope of this exercise, we will be using macro definitions exclusively as global constants. Here we define `CONSTANT_NAME` to refer to `literal_value` (an integer literal). Note that there is only a space separating name from value.

```
#define CONSTANT_NAME literal_value
```

Now, look at the code contained in `eccentric.c`. **Notice** the four different examples of basic C control flow. (Think: What are they?) Also, do you recognize these eccentric sayings and people from Berkeley? :)

First compile and run the program to see what it does. Play around with the constant values of the four macros: V0 through V3. See how changing **each** of them changes the program output.

Action Item

Modifying only these four values, make the program produce the following output.

```
Berkeley eccentrics:
====================
Happy Happy Happy
Yoshua
Go BEARS!
```

There are actually several different combinations of macros that can give this output. Here's the challenge for you in this exercise: **consider what the minimum number of *distinct* values that V0 through V3 can have such that they still give this exact output. For example, the theoretical maximum is four, when they are all distinct from each other.**

Stuck on how to run the program? Revisit the introduction. We'd like you to compile the program into an executable called `eccentric`; can you use the `-o` flag to do this?

Exercise 2: Catch those bugs!

A **debugger**, as the name suggests, is a program which is designed specifically to help you find bugs, or logical errors and mistakes in your code (side note: if you want to know why errors are called bugs, look here). Different debuggers have different features, but it is common for all debuggers to be able to do the following things:

1. Set a breakpoint in your program. A breakpoint is a specific line in your code where you would like to stop execution of the program so you can take a look at what's going on nearby.
2. Step line-by-line through the program. Code only ever executes line by line, but it happens too quickly for us to figure out which lines cause mistakes. Being able to step line-by-line through your code allows you to hone in on exactly what is causing a bug in your program.

For this exercise, you will find the GDB reference card useful. GDB stands for "GNU De-Bugger." Compile `hello.c` with the `-g` flag:

```
$ gcc -g -o hello hello.c
```

This causes gcc to store information in the executable program for `gdb` to make sense of it. Now start our debugger, (c)gdb:

```
$ cgdb hello
```

Notice what this command does! You are running the program cgdb on the executable file hello generated by gcc. Don't try running cgdb on the source code in hello.c! It won't know what to do. If cgdb does not work, you can also use gdb to complete the following exercises (start gdb

with `gdb hello`). The cgdb debugger is only installed on your cs61c-xxx accounts. Please use the hive machines or one of the computers in 27x Soda to run cgdb, since our version of cgdb was built for Ubuntu.

Note: you're welcome to install and run (c)gdb on your local computer, but be advised it will not install on (updated) MacOS machines. If this applies to you, you can use lldb which is another great debugger. The commands differ slightly, but there are great guides out there (like this one!) to help you get started. For this lab, though, use the lab machines and cgdb.

Action Item

Step through the whole program by doing the following:

1. setting a breakpoint at main
2. using gdb's run command
3. using gdb's single-step command

Type help from within gdb to find out the commands to do these things, or use the reference card.

Look here if you see an error message like printf.c: No such file or directory. You probably stepped into a printf function! If you keep stepping, you'll feel like you're going nowhere! CGDB is complaining because you don't have the actual file where printf is defined. This is pretty annoying. To free yourself from this black hole, use the command finish to run the program until the current frame returns (in this case, until printf is finished). And **NEXT** time, use next to skip over the line which used printf.

Note: cgdb vs gdb

In this exercise, we use cgdb to debug our programs. cgdb is identical to gdb, except it provides some extra nice features that make it more pleasant to use in practice. All of the commands on the reference sheet work in gdb.

In cgdb, you can press `ESC` to go to the code window (top) and `i` to return to the command window (bottom) — similar to vim. The bottom command window is where you'll enter your gdb commands.

Action Item

Learning these commands will prove useful for the rest of this lab, and your C programming career in general. In the text file named `gdb.txt`, answer the following questions.

1. While you're in a gdb session, how do you **set the arguments** that will be passed to the program when it's run?
2. How do you **create a breakpoint**?
3. How do you **execute the next line of C code** in the program after stopping at a breakpoint?
4. If the next line of code is a function call, you'll execute the whole function call at once if you use your answer to #3. (If not, consider a different command for #3!) How do you tell GDB that you **want to debug the code inside the function** (i.e. step into the function) instead? (If you changed your answer to #3, then that answer is most likely now applicable here.)
5. How do you **continue the program after stopping** at a breakpoint?
6. How can you **print the value of a variable** (or even an expression like 1+2) in gdb?
7. How do you configure gdb so it **displays the value of a variable after every step**?
8. How do you **show a list of all variables and their values** in the current function?
9. How do you **quit** out of gdb?

Exercise 3: Debugging w/ YOU(ser input)

Let's see what happens if your program requires user input and you try to run GDB on it. First, run the program defined by `interactive_hello.c` to talk to an overly friendly program.

```
$ gcc -g -o int_hello interactive_hello.c
$ ./int_hello
```

Now, we're going to try to debug it (even though there really are no bugs).

```
$ cgdb int_hello
```

What happens when you try to run the program to completion? We'll be learning about a tool to help us avoid this situation. The purpose of this exercise is to make you **unafraid** of running the debugger even when your program needs user input. It turns out that you can send text to `stdin`, the file stream read by the function fgets in this silly program, with some special characters right from the command line.

Take a look at "redirection" on this website, and see if you can figure out how to send some input to the program without explicitly providing it while it's running (which, I hope you've realized, gets you stuck in CGDB). Look at this stackoverflow post for more inspiration.

Hint 1: If you're creating a text file containing your input, you're on the right track! Hint 2: Remember you can run things with **command line args (including the redirection symbols) from CGDB as well!**

Hopefully you'll appreciate how redirection helps you avoid that nasty situation with CGDB. Don't ever be afraid of the debugger! We know it looks kind of nasty, but it's there to help you.

Exercise 4: Valgrind'ing away

Even with a debugger, we might not be able to catch all bugs. Some bugs are what we refer to as "bohrbugs", meaning they manifest reliably under a well-defined, but possibly unknown, set of conditions. Other bugs are what we call "heisenbugs", and instead of being determinant, they're known to disappear or alter their behavior when one attempts to study them. **We can detect the first kind with debuggers, but the second kind may slip under our radar because they're (at least in C) often due to mis-managed memory.**

Remember that unlike other programming languages, C requires you (the programmer) to manually manage your memory. We'll cover this more later this week, but this is all you'll need to know for now.

To help catch these "heisenbugs" we will use a tool called Valgrind. Valgrind is a program which emulates your CPU and tracks your memory accesses. This slows down the process you're running (which is why we don't, for example, always run all executables inside Valgrind) but also can expose bugs that may only display visible incorrect behavior under a unique set of circumstances. Valgrind is installed by default on hive machines. Valgrind is available for local install on most platforms; however, a version for more recent versions of MacOS may not yet be available. While you are welcome to do this exercise on the hive it will likely be in your benefit to have Valgrind locally for project testing later on.

In this exercise we are going demonstrate two different examples of Valgrind outputs and walk through how each might be useful.

First try building two new executables, `segfault_ex` from `segfault_ex.c` and `no_segfault_ex` from `no_segfault_ex.c` (use the `-o` flag from before!). At this point you should be able to use `gcc` to successfully build these executables.

Now let's try running the executables. What outputs do you observe?

Let's start with `segfault_ex`. You should have observed a segmentation fault (segfault), which occurs when a program crashes from trying to access memory that is not available to it (more on this later in the course; This is actually an artifact from early memory design. Today we work with 'paged memory' instead of 'segmented memory' but the error message remains!).

This C file is very small so you should be able to open the file and understand what is causing the segfault. Do so at this time, but do not change the file. Why does the segfault occur?

Now let's understand what to do if we have a very large file and need to find a segfault. Here is where Valgrind is our new friend. To run the program in Valgrind use the command:

```
$ valgrind ./segfault_ex
```

This should cause Valgrind to output where the illegal access occurred. Compare these results to what you determined by opening the file. How could Valgrind help you address a segfault in the future? Now try running Valgrind on `no_segfault_ex`. The program should not have crashed but there is still an issue with the file. Valgrind can help us find the (seemingly invisible) problem.

Unfortunately here you will see that Valgrind seems to not be able to tell you exactly where the problem is occurring. Use the message provided by Valgrind to **determine which variable has undefined behavior and then try to infer what must have happened** (Hint: What is an uninitialized value?).

At this point we do not expect you to be familiar with `sizeof` (that comes next week!) so all we want you to get out of this section is some intuition around where the problem likely occurred.

Hopefully after walking through this example you'll be able to understand and answer the following:

- Why is Valgrind important and how is it useful?
- How do you run a program in Valgrind?
- How do you interpret the error messages? **Don't be afraid of them. Try your best and ask us for help.**
- Why might uninitialized variables result in "heisenbugs"?

We are introducing you to Valgrind now because it is an extremely important tool that you will want as soon as you start writing C. **However to really appreciate it we will need to learn about C's memory model, which we will do next week.** After you have covered memory in lecture, come back to this lab and try answering the following questions:

- Why **didn't** the `no_segfault_ex` program segfault?
- Why does the `no_segfault_ex` produce inconsistent outputs?
- Why is `sizeof` incorrect? How could you still use `sizeof` but make the code correct?

Exercise 5: Pointers and Structures in C

Here's one to help you in your interviews. In `ll_cycle.c`, complete the function `ll_has_cycle()` to implement the following algorithm for checking if a singly- linked list has a cycle.

1. Start with two pointers at the head of the list. We'll call the first one tortoise and the second one hare.
2. Advance hare by two nodes. If this is not possible because of a null pointer, we have found the end of the list, and therefore the list is acyclic.
3. Advance tortoise by one node. (A null pointer check is unnecessary. Why?)
4. If tortoise and hare point to the same node, the list is cyclic. Otherwise, go back to step 2.

If you want to see the definition of the `node` struct, open the `ll_cycle.h` header file.

Action Item

Implement `ll_has_cycle()`. Once you've done so, you can execute the following commands to run the tests for your code. If you make **any** changes, make sure to run **ALL** of the following commands again, in order.

```
$ gcc -c ll_cycle.c
$ gcc -c test_ll_cycle.c
$ gcc -o test_ll_cycle test_ll_cycle.o ll_cycle.o
$ ./test_ll_cycle
```

Hint: There are two common ways that students usually write this function. They differ in how they choose to encode the stopping criteria. If you do it one way, you'll have to account for a special case in the beginning. If you do it another way, you'll have some extra NULL checks, which is OK. The previous 2 sentences are meant to urge you to not stress over cleanliness. If they don't help you, just ignore them. The point of this exercise is to make sure you know how to use pointers.

Here's a Wikipedia article on the algorithm and why it works. Don't worry about it if you don't completely understand it. We won't test you on this.

Setup
Objectives:
Compiling and Running a C Program
Exercise 1: See what you can C
Exercise 2: Catch those bugs!
Exercise 3: Debugging w/ YOU(ser input)
Exercise 4: Valgrind'ing away
Exercise 5: Pointers and Structures in C
Checkoff

By the way, the pointers are called “tortoise” and “hare” because the tortoise pointer is incremented slowly (like a tortoise, which moves slowly) and the hare pointer is incremented quickly (twice as fast as the tortoise; like a hare, which moves quickly).

As a closing note, the story of the tortoise and the hare is always relevant, especially in CS61C. Writing your C programs slowly and steadily, using debugging programs like CGDB, is what will win you the race.

Checkoff

Please submit to the **Lab Autograder** assignment (same as last week!).

Note that the autograder for the gdb answers is very simple. Make sure that:

1. All of the lines in the `gdb.txt` file start with the answer number (e.g. `1. answer`).
2. Only put the command **name** in the answers. So, if we have a command that takes an argument, i.e. `command arg`, only put `command` in the `gdb.txt` file.