CS61C Spring 2018 Lab 1 - C Pointers and GDB

HEY! WELCOME TO LAB 1 OF CS61C. IT'S LAB 1 BUT IT'S THE SECOND LAB. THAT'S ZERO-BASED INDEXING.

Goals

- Learn how to compile and run a C program on the *EECS instructional computers*
- Examine different types of control flow in C
- Introduce you to the C debugger
- Gain some practical experience using gdb to debug C programs
- Get more comfortable working with pointers

BEFORE YOU ASK FOR CHECKOFF: Have the following open and ready to show to the TA or lab assistant. Doing this before asking for a checkoff will speed up the checkoff process.

- The file `eccentric.c`
- Your answers to the questions regarding GDB commands (Exercise 2)
- The output from running the `ll_equal` program
- The file `ll_cycle.c`
- The output from running the `ll_cycle` program

Setup

Copy the contents of `~cs61c/labs/01` to a suitable location in your home directory. These are all the files you'll need to complete this lab. Below is an example of how to do so. Lab files will **always** be located in `~cs61c/labs`.

```
$ mkdir labs      # only run this if you don't have the 'labs' directory
$ cp -r ~cs61c/labs/01/ ~/labs
```

Note: `cp` means "copy" and the `-r` flag is for copying folders (*recursively*)

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 `./`? Answer: when you want to execute an executable, you need to prepend a filepath 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. Using `-o`, you would 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: Simple C Program

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 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. (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.

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

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

Checkoff

- Explain the changes you made.
- Explain the minimum number of distinct values needed for the preprocessor macros.
- What does the -o flag do with `gcc`?

Exercise 2: Debugger

What is a debugger?

This section is intended for students who aren't familiar with what debuggers are. 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 the 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.

Task: step through the whole program by:

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.

Task: Learn MORE gdb commands

Learning these commands will prove useful for the rest of this lab, and your C programming career in general. Create a text file containing answers to the following questions (or write them down on a piece of paper, or just memorize them if you think you want to become a GDB pro).

1. How do you **pass command line arguments** to a program when using gdb?
2. How do you **set a breakpoint** which only occurs when a **set of conditions is true** (e.g. when certain variables are a certain value)?
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** instead? (If you changed your answer to #3, then that answer is most likely now applicable here.)

5. How do you **resume the program after stopping** at a breakpoint?
6. How can you **see the value of a variable** (or even an expression like 1+2) in gdb?
7. How do you configure gdb so it **prints the value of a variable after every step**?
8. How do you **print a list of all variables and their values** in the current function?
9. How do you **exit** out of gdb?

Checkoff

- Set the breakpoint at main, and show your TA how you run up to that breakpoint.
- Show your TA your text document containing the additional gdb commands (or verbally recite them, whatever is easier).

Exercise 3: Debugging a buggy C program with GDB

You will now use your newly acquired gdb knowledge to debug a short C program! Consider the program `ll_equal.c`. Compile and run the program, and experiment with it. By default, it will give you the following result:

```
$ gcc -g -o ll_equal ll_equal.c
$ ./ll_equal
equal test 1 result = 1
Segmentation fault
```

Figure out what's causing that segmentation fault!

Start gdb on the program, following the instructions for compilation in exercise 1. We recommend setting a breakpoint in the `ll_equal()` function. When the debugger stops at the breakpoint, try stepping through the program to see if you can figure out what's causing the error.

Hint 1: run `bt` after the program crashes in gdb. It will output the program trace.

Hint 2: pay attention to the values of the pointers a and b in the function (print them!). Are they always pointed to the right address?

Hint 3: Look at the source code in `main` to see the structure of the nodes and what exactly is being passed into `ll_equal`.

Checkoff

- Explain the bug and your fix to the function.

Exercise 4: "Debugging" a C program that requires user 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!

Checkoff

- Demonstrate how you run CGDB to completion on the executable created by compiling `interactive_hello.c` without getting stuck.

Hopefully you appreciate how redirection helped 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 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.

After you have correctly implemented `ll_has_cycle()`, the program you get when you compile `ll_cycle.c` will tell you that `ll_has_cycle()` agrees with what the program expected it to output.

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.

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

- Show your `ll_has_cycle()` function to the TA.