CS61C Spring 2015 Lab 1 - Running C Programs and Debugging With gdb

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 gdb (and its cousin, cgdb).
- Gain some practical experience using gdb to debug a buggy C program.

The CS61C Lab Machines

In CS61C, we will use the machines in 330 Soda for all work/grading purposes. Whenever you're working from home, you should ssh into one of the 30 hive machines (hive1.cs.berkeley.edu to hive30.cs.berkeley.edu) in order to do your work. Unless we specifically instruct you to do so, don't ssh into any other instructional machine to do 61C work.

Although not required, we also highly recommend that you learn to use a text editor that works well over ssh. Going forward, this will save you a considerable amount of time setting up your environment in almost every EECS class. The instructional machines (in our case the hive machines) have everything you need for this class pre-installed. The hive machines are powerful workstation-class desktops with the following specifications:

```
Model: Dell Optiplex 9020
CPU: Intel Core i7, 3.4 GHz, quad-core, Hyper Threading
Memory: 32GB RAM (4x8GB)
Disk: 500GB SATA drive
GPU: GeForce GT 740 1 GB DDR3 PCI Express 3.0 x16 Cuda GPU
OS: Ubuntu Linux
```

Setup

Copy the contents of ~cs61c/labs/01 to a suitable location in your home directory. You do not need to add the lab files to your git repo.

```
$ cp -r ~cs61c/labs/01/ ~/labs/01
```

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 (note that there is no file called program.c given in the lab files, this is just an example for your information).

```
$ gcc program.c
```

This compiles program.c into an executable file named a.out. This file can be run with the following command.

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

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.

```
$ 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. We see four different examples of basic C control flow. First compile and run the program to see what it does. Play around with the constant values of the four macro: V0 through V3. See how changing them changes the program output. 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 values that can give this output. You should consider what is the minimum number of different values V0 through V3 can have to give this same output. The maximum is four, when they are all distinct from each other.

Checkoff

- Explain the changes you made.
- Explain the minimum number of different values needed for the preprocessor macros.

Exercise 2: Debugger

For this exercise, you will find the GDB reference card useful. 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
```

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 271, 273, 275, or 277 soda to run cgdb, since our version of cgdb was built for Ubuntu.

Step through the whole program by:

1. setting a breakpoint at main
2. giving 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.

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.

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:

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. How do you tell GDB that you want to debug the code inside the function instead?
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.

Exercise 3: Debugging a buggy C program

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. It will give you the following result:

```
$ gcc -g -o ll_equal ll_equal.c
$ ./ll_equal
```

```
equal test 1 result = 1
Segmentation fault
```

Now, start gdb on the program, following the instructions in exercise 1. Set a breakpoint in the ll_equal() function, and run the program. When the debugger returns at the breakpoint, step through the instructions in the function line by line, and examine the values of the variables. Pay attention to the pointers a and b in the function. Are they always pointed to the right address? Find the bug and fix it.

Checkoff

- Explain the bug and your fix to the function.

Exercise 4: 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.

Checkoff

- Show your ll_has_cycle() function to the TA.