Lab 6

Due Oct 20, 2023 by 6:30p.m. **Points** 1

Lab 6: Using the Debugger

Introduction

The main purpose of this lab is to give you some practice using the debugger. Before that, we have one more string function for you to write.

You will also need to do the debugger part of the lab on the teach.cs machines (or your own Linux machine). This means that you should be sure that you can use ssh and and either git or scp to copy files.

Starting

As usual, go to Lab 6 on Markus and ensure that the starter code is in your repository.

More Practice with Strings

Complete the program `copy.c` according to the instructions in the starter code. You can do this part on your own machine.

Using gdb

It is common for programmers to debug using *print*, at least for simple issues, but debugging with print statements will become increasingly difficult as we delve into systems programming. Now is the time to learn how to use a common C debugger, `gdb`. It will seem hard at first, but you'll be thankful for these skills at the end of this course, and in courses like CSC369, the ability to use the debugger will save you *many* hours of debugging time.

Do this part of the lab on the teach.cs machines. Hopefully, you already have experience logging into the teach.cs machines, but just in case run the following from a terminal on your local machine.)

ssh **USERID@teach.cs.toronto.edu (mailto:USERID@teach.cs.toronto.edu)**

If you haven't already, you should clone a local copy of your repo here. (Remember to commit and push the version on your local machine first, so they don't get out of sync.)

To demonstrate that you have completed the lab, you will submit a `script` record of your interactions on the command line to your git repository in a file called `gdb_example.txt`. Try it out by typing `script` then typing a few unix commands. Perhaps make a directory for your work on this lab or list the files in your current directory or whatever you wish. Then after you've done something, type `exit`. This will stop recording your actions and save them in a file named `typescript`. Check out the *man* page for `script` to see that you can give it a filename as an argument to override this default name.

Note that `script` will record control characters such as backspace and newlines. This will make your output look hard to read. **Submit the file produced by script unaltered.** There is a program you (and we) can use to view the script output without the control characters. Run `col -b < gdb_example.txt` to see the file without the extra control characters.

The file `overflow.c` (in your repo in `Lab6`) contains a program to explore. You will change the values of `SIZE` and `OVERFLOW` to see what happens when `OVERFLOW` is bigger than `SIZE`.

First, read through the program and explain what it is doing aloud to yourself, to a stuffed animal, a patient family member, or to some unsuspecting bystander, if you prefer. Notice that we are printing the addresses of the variables. The purpose of doing that is to show where the variables are placed in memory.

Next, compile and run the program as shown here:

```
$ gcc -Wall -g -o overflow overflow.c
$ ./overflow
```

Don't miss the `-g` flag or gdb won't work properly. Check the values of *before*, *a*, and *after* -- did the program behave as expected?

Now change the value of `OVERFLOW` to 5. Compile the program and run it again. What changed? (If nothing changed -- if everything still seems okay -- then try this code on a lab machine. It depends on how the variables are placed into memory by the compiler, and your compiler may be doing something we didn't expect.)

Let's see why variables other than *a* were affected. The next step is to run the program in `gdb`. Here are a list of the 9 need-to-know commands in gdb:

gdb *executable*	start gdb on this executable
list [n]	list some of the code starting from line n or from the end of last call to list
break [n or fun_name]	set a breakpoint either at line n or at the beginning of the function fun_name
run [args]	begin execution with these command-line arguments
next	execute one line
print *variable or expression*	print the value once

display *variable or expression*	print the value after every gdb command
continue	execute up to the next breakpoint
quit	bye-bye!

Try this out on your `overflow` executable. Start by typing `gdb overflow`. Set a breakpoint in main by typing `break main`, and then start the program running by typing `run`. You want to watch the values of a few variables, so use `display` to show the value of some variables. Do this for each variable you want to watch. Step through the program one line at a time using `next` (after you enter `next` once, you can execute another line by just hitting "enter"). Keep a close eye on the first element in `after`, and notice the final value of `a`. When the program terminates, type `quit` to leave gdb.

Now start `gdb` again but before you start, run `script` `gdb_example.txt` to record your interaction. (You'll submit this interaction.) This time make sure you watch the array `after`. It is pretty slow to step through every line of your code, so use `list` to find the line number of the for loop where we start to assign values to the array. Set a breakpoint on that line number and also set a breakpoint somewhere before that line. Start your program using `run` and it should run up to the first breakpoint. Then use `continue` to jump to the second breakpoint you set which should be the for loop. At any point, you can use `continue` to execute up to the next breakpoint. If you tried it again now, it should jump to the second pass through the loop.

Instead, use `next` to step through one line at a time. Watch the value of `after[0]` carefully. When it changes, print its address. (Yes, you can do "`print &(after[0])`" inside `gdb`.) Then, print the address of `a[4]`. Does that help you understand what happened? Exit `gdb`. (Say 'y' when it asks if you want to Quit anyway.) Then exit your script by typing `exit`. Make sure the script file you generated is named `gdb_example.txt` and add it to your repository.

The last step is to try to make your program crash. (I had to set `OVERFLOW` to something like 5000 to get it to crash with a Segmentation fault.) Once you've found a value for `OVERFLOW` that produces a Segmentation fault, run the program in `gdb` without setting a breakpoint so that it runs automatically until the crash. Then use the `backtrace` command and the `print` command to investigate the value for `i`. Try `backtrace full` for even more helpful information. You don't need to record what you're doing on this step. We just want you to see the `backtrace` command.

Now that you know how useful gdb is, you should be thinking about how to run it on your own machine. On the mac you can use lldb which is very similar. Here is a **mapping (https://lldb.llvm.org/use/map.html)** between the two sets of commands.

Submitting Your Work

Make sure that `gdb_example.txt` is added to the `Lab6` directory. Remember to add, commit and push all changes to `copy.c` and `gdb_example.txt`. Do not commit any executables or any other additional files.

We will be auto-testing your solution to `copy.c` so you must not change the signature. We will be checking that your gdb_example.txt file has some of the commands specified above. If you have extra steps during your debugging, this is fine and you don't need to stress about the format of the output. We aren't expecting that every student's `gdb_example.txt` will be the same. We do however expect to see a serious attempt to use gdb effectively.