CS262, Lab Assignment 6:
Debugging Code with GDB

Due: Sunday, March 27 at 11:59 pm ET

Debugging computer source code can be a frustrating experience. Since source code is becoming much more complex, many tools and techniques to assist in debugging have been developed. It is helpful to be aware of these tools and techniques so that you can use them while working on your projects and labs. With them, you can become successful in submitting correct, well-written source code.

Description:

The purpose of this assignment is to provide practice using the gdb debugger. GDB allows you to do many useful things to examine your code. However, this assignment will concentrate on some of the simpler usages of gdb - setting breakpoints and inspecting variables.

Download the answer sheet for this assignment and filling out meanwhile you are debugging the provided program for this assignment. The source code `lab6.c`, as well as the answer sheet can be found with the materials for this assignment.

Debuggers Background

Debuggers:

A **debugger** is a program that allows you to *see what is going on while a program is executing*. Most debuggers allow programmers to set breakpoints in their code (places where execution is paused) and examine the state of variables while the program is paused. Other features include the ability to set conditional breakpoints which will pause execution at the point where the value of a variable is changed or set to a specific value. The ability to see what is happening is very handy when you are trying to track down problems inside your code.

There are many different debuggers available which can be divided into two types:

1. **Command Line Interface (CLI):** These debuggers use commands typed on a keyboard to operate. Common debuggers of this type are DBX and GDB.
2. **Graphical User Interface:** These debuggers are often GUI wrappers over a Command Line Interface debugger. They allow programmers to use a mouse to set breakpoints, run code, and inspect variables. Certain GUI debuggers are part of a larger Integrated Desktop Environment (IDE). Examples of popular IDEs are Eclipse and Microsoft Visual Studio.

GDB:

This assignment will introduce you to one of the more popular CLI debuggers, gdb, which is the GNU Project debugger, and is part of the overall GNU consortium.

- The project page is at http://www.gnu.org/software/gdb/
- Debugging documentation is at https://sourceware.org/gdb/download/onlinedocs/gdb/index.html

Instructions – Start the script

Typescript file

(1) For this assignment you need to create a typescript named `lab6_typescript_<username>_<labsection>`
(2) Show that you are logged onto Zeus
(3) After performed (1)(2), follow instructions From "**Compiling**" To "**More on Breakpoints**"
(4) End the typescript by typing `Control-d`

Compiling

In order to use gdb to debug an executable, you must compile the program with "*debugging options*" turned *on*. The most common option is the "`-g`" option. The `-g` option will add debugging information to your executable. This will make your program size larger, and could potentially cause your program to run slower. To see this, first compile `lab6.c` on zeus as:

```
gcc lab6.c -o lab6
```

Check the size of the lab6 executable with the `ls -l` command:

```
ls -l lab6
```

1. Write the size (in bytes) of the executable ___________

The size in bytes is the number before the date in the output from the `ls -l` command

Now, recompile `lab6` with following command:

```
gcc lab6.c -o lab6 -g
```

2. Write the size (in bytes) of the new executable ___________

There are many options that can be used to generate debugging information in your code. The man page for gcc will list many of them. For now, just stick with the `-g` option.

Starting GDB

To start GDB, the easiest way is to type "gdb" (without quotes) at the command line. Do so now. You should see something similar to:

```
bash $ gdb
GNU gdb (Ubuntu/Linaro 5.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show
copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>.
(gdb)
```

What you actually see depends upon what type of computer you run the command on, and the particular version of gdb installed.

3. Copy the first line that is printed when you run gdb: ________________
4. Now, copy the portion in the quotes on the line that states "This GDB was configured as" For instance, in the example above, you would copy **`i686-linux-gnu`**:

   ____

Getting Help

GDB has internal help. You can get a list of help topics by typing "`help`" (without quotes) at the (gdb) prompt. Do so now, and count the number of topics shown for "List of classes of commands:"

5. Write the number of topics/classes: ________________

To get help on a specific topic, you type "`help <topic name>`" So, to get help on breakpoints, type "`help breakpoints`" (without quotes, of course). Try to get help on breakpoints.
As you can see, there are many different commands you can use regarding setting and using breakpoints.

The help command can also give you additional information on various commands. For instance, the "`break`" command is one you will likely use many times in your programming career. Try it now type "`help break`" to get specific information:

6. Write the first line printed after the executing the command here: ________________

Quitting GDB

Just as important as starting `gdb`, you will need to know how to quit `gdb`. This is done with the "`quit`" command. Type "`quit`" at the (`gdb`) prompt now.

Note on GDB Commands

Many `gdb` commands have shortcuts so you won't have to type the complete command. For instance, instead of typing "`quit`" to exit `gdb`, you can type "`q`" to have the same result. Or, for the `break` command, you can just type "`b`."

Using GDB to Debug a Program

Now that you know how to start and quit GDB, let's debug the lab6 program you compiled earlier. Make sure you are using the version you compiled with the "`-g`" option. Type the following command to start `gdb`:

```
bash$ gdb lab6
```

gdb will start, and will automatically load your program so that it can be run within GDB.
You should see a line such as:

```
Reading                    symbols                    from
/home/user/CS262/lab6_loreto/lab6...done.
```

as gdb loads your program.

While debugging your code with gdb, you may find errors and wish to recompile and reload your program without exiting gdb. You load a program into gdb with the "`file`" command:

```
(gdb) file lab6
```

gdb will prompt you to make sure you wish to "Load new symbol table ..." You can also use the file command to load a program into gdb if you started it without specifying a file on the Unix command line. However, most times, gdb will realize that you recompiled your code, and automatically load the latest version.

Running a Program

You run your program using the "run" command:

```
(gdb) run
```

7. Try it now. What happened? ________________

Your program takes command line arguments. However, no arguments were used when executing the run command. If you are debugging a program with command line arguments, you simply add them to the run command:

```
(gdb) run 1
```

8. Try the run command again, this time adding "1" (without quotes) to the command line. What happened this time? _______________

Debugging the Program lab6

When option 1 is chosen while running `lab6`, it (incorrectly) adds the numbers from 1 to 10 and prints the result. You might be able to look at the code directly to see the error and fix it, but let's use some typical gdb commands to inspect the code and discover what the problem is. At this point, you will likely find it more convenient to have three Unix windows open on your computer. One window will have a `vim/vi` session where you edit the source code, one window will be used for compiling your code after making changes, and the third window will contain your gdb session.

Setting Breakpoints

There are two main ways to set breakpoints in gdb: (1) by line number, and (2) by function. When setting a breakpoint by function you use the syntax:

```
(gdb) break
```

Try to set a breakpoint so that the program pauses execution after entering the `DebugOption1()` function. The gdb command you will use is:

```
(gdb) break DebugOption1
```

You should see gdb give a response similar to:

```
Breakpoint 1 at 0x804860a: file lab6.c, line 55.
```

You can set many breakpoints, so each breakpoint you set is assigned a number. This is so you can toggle breakpoints on or off, or delete them.

Now run the program in gdb using 1 as the command line argument.

9. At what line number does execution pause? ___________________

Notice that gdb did not stop at line 54 which contained:

```
    int i, j;
```

The reason it did not stop there is because there is no code to execute on that line. That line only declares variables. Instead, the execution steps to line 55 which contains:

```
    int sum = 0;
```

Because the variable sum is assigned a value which requires an "execution" of code, gdb stops on that line.

A Brief Digression

Before going on to step through the program, there are some other useful commands you can do at this point. One is to print the current stack. To do this, you use the command "where". Try this command now. You should see a response similar to:

```
    #0 DebugOption1 () at lab6.c:55
    #1 0x08048564 in main (argc=2, argv=0xbffff3e4) at lab6.c:32
```

The lines above state where each function call occurred up to the point execution halted. As mentioned before (and shown by line #0), you are currently at line 55 in lab6.c, which is the start of the `DebugOption1` function. You can also see (from line #1) that this function was called from the `main()` function and the call occurred at line 32 in the file. You can also move up and down the stack using the "up" and "down" functions. Try typing "up" at the gdb command line. The general "position" of gdb is now in the `main()` function. That means you can print values of variables that are contained in `main()` but perhaps not accessible from `DebugOption1()`. For example, `main()` contains a variable named "`option`". According to the source code, if option is equal to 1, it will call the `DebugOption1()` function. Let's verify the value of option by using the following command:

```
    (gdb) print option
```

10. Write what you see as a result: ___________________

Print is one of the commands you will use most often when using gdb. Go back to the `DebugOption1()` function by typing the "down" command.

Stepping Through Code

Besides printing values of variables, and setting breakpoints, the other commands you will use most with gdb has to do with stepping through the code *line by line*. There are several commands that you will use to do this:

- **`step`** - *step to the next source code line*. If the line to be executed is a function call, the function will be entered and the execution will pause.
- **`next`** - similar to step, but will *skip over a function* call before stopping. The function call is still executed, but the program *does not step inside the function*.
- **`continue`** - continue running from the current line until the next breakpoint or the end of the program is reached.
- **`finish`** - continue until the current function exits and stop after the return (also called "step out of function").

Most of these commands can use just the first few letters in the command. For instance, '`s`' can be used for `step` and '`n`' can be used for `next`.

Step and next can take an integer as an optional argument. For instance, typing:

```
(gdb) step 5
```

will cause the program to step through five source code lines before pausing. In essence, it is similar to using the step command five times in a row.

Let's try using the step command. If you have been following the directions up to this point, you will be at the beginning of the `DebugOption1()` function (line 55 in `lab6.c`). Keep the window that shows the source code handy. The variable sum has not yet been assigned.

11. Use the print statement mentioned above to print the current value of sum.
    What is its current value? ___________________

It is possible that the current value of sum is some large random integer. This is because sum has not yet been initialized (sometimes the value of sum will be 0, but it won't always be the case).

12. Now, use the step command to step over the current line.
    At what line does the execution pause? ___________________
13. Print the value of sum again. What is its value now? ___________________

Use `next` and/or `step` to continue stepping through the program until the following line is reached (Note: you may wish to use `next` instead of `step` to step over the `fprintf()` call):

```
sum += sum + i;
```

14. Print the value of sum: __________________

Breakpoints Again

Now we will demonstrate the other typical way breakpoints are set (by line number). Note the line number of the current place where execution is paused. To set a breakpoint at a specific line number, simply type "`break <line number>`". Type the following command:

```
(gdb) b 63
```

(Note “b” the shortened command for break). The second breakpoint has now been set. Every time the continue command is given, the execution will pause at line 63, until the loop finishes.

Displaying Variables

We want to observe how the value of sum changes as the program runs. We could use the print command each time we reach a breakpoint. However, this can become tedious, especially if there are several variables we wish to monitor. gdb has a command called "display". The syntax for is "display ." Type the following command to turn on the display of the variable sum:

```
(gdb) display sum
```

This command is similar to print, except that every time execution pauses, the values of all variables that are set to display are printed. To turn off the display of specific variables, use the "undisplay ." command.

Now, continue execution several times with the continue (cont) command. Stop when sum is equal to 57.

15. What is the value of the variable 'i' at this point? ________________

Does it make sense that 0 + 1 + 2 + 3 + 4 + 5 + 6 = 57? Do you see the bug(s) in the program? If you don't, start the run of the program from the beginning with the "run 1" command and step through the code within this loop carefully. The breakpoints and the display are still set, even though you are restarting the program. You may also wish to display the i variable instead of printing it out each time the program pauses.

16. Once you see what the bug(s) is/are, describe it/them below:

____

Re-running After Changes

Fix the bug in your source file window and save your changes. Recompile the executable in the window you are using for compiling.

Now, type the command to run the executable ("run 1"). If you were still in a debugging session, gdb will ask you if you wish to start again from the beginning (y or n). Pick 'y' and hit return. gdb will notice that the executable has changed and will reload the symbol table (i.e. load the new program). Note that the breakpoints will still be present. However, you will have to redisplay any variables you had set to display automatically. Note also, that if you added or removed lines of code during editing, your breakpoints may be at different source code lines. If this is the case, you may need to update your breakpoints.

More on Breakpoints

Suppose you have set several breakpoints, but you don't currently remember at what lines they were set. You can use the command:

```
(gdb) info breakpoints
```

to list them.

You can disable or enable breakpoints as you are debugging. Disabling a breakpoint leaves the breakpoint present, but execution will not pause when the breakpoint is hit. To disable or enable a breakpoint, you must use the `breakpoint number` or `id`. This value can be found with the "`info breakpoints`" command, and to disable it use the "`disable <breakpoint id>`" command. A disabled breakpoint is enabled with "`enable <breakpoint id>`".

If you don't need a breakpoint, you can delete it with the "`delete <breakpoint id>`" command.

End of the script

Conclusion

The above narrative discusses several important and useful gdb commands. At this point, use the given code to take some time to experiment with the commands you have learned. The more you experiment, the more comfortable you will be using gdb. And the more comfortable with gdb you are, the faster you will be able to find bugs in your code. The faster you find bugs in your code, the sooner you will complete your assignments and your quality of life will improve. See what wonderful things gdb can do for you?

There are many additional commands that can be used with gdb, and this assignment only scratched the surface. It is recommended that you use the help command liberally and seek out other gdb tutorials so that you may gain practice with some of the finer points of using gdb. As it is with an editor such as `vi/vim` or `emacs`, you will be spending a lot of time using debuggers. It is best to become as familiar as you can with it since you will be using it (or other debuggers) often throughout your career.

Submission

- Create a `pdf` of your answered "Answer Sheet", name the file as `lab6_<username>_<labsection>.pdf`
- Create a tarfile of your `lab6` directory, it should contain the files: `lab6_typescript_<username>_<labsection>, lab6_<username>_<labsection>.pdf`
- Submit the tarfile for your assignment to Blackboard

NOTE: For this assignment you **MUST** submit your typescript as support that you ran gdb following the given steps. Submission of just the answer sheet is not accepted. *If you do not include the typescript, the score for lab6 is automatically set up to 0.*

Congratulations! You have completed your assignment