Lab 02 - Tools Lab

Due: Sunday, September 24, 2017 at 4:00 pm

1 Introduction

Mike and Sully are trying to hack into the door database to locate Boo, but they can't find her without fixing their buggy code. They need to learn how to use debugging tools in C!

2 Assignment

Much of the code you have previously written was probably written in an IDE - an *integrated development environment* - which provided many convenient features to help you write your code. For example, the Eclipse IDE, often used for Java programming, performs syntax highlighting and contains a built-in debugger. Thus far in CS0330, we have been writing, compiling, and running C code without any such tools. In this lab you will learn to use a variety of tools which will assist you in writing programs without the help of an IDE.

Your task for this lab consists of two parts: first you will use debugging tools to diagnose C bugs within the provided executable binaries; then you will fix the bugs within the source files of those binaries.

You need only complete the first part to be checked off for this lab.

Several of the tools which you may find useful for this lab are described in the document **tools.pdf**. This handout is provided on the website in the guides and documents section ("Tooling Guide"), and in the lab stencil. The most important tool for this lab, **gdb**, is described below and in **gdb.pdf**.

To get started, run the command **cs0330_install lab02**.

We have provided you the source code in addition to binaries.You are free to look at it as you debug the binaries, but we are certain that you will find monitoring execution with **gdb** to be more helpful. Remember that the point of this lab is to learn to use **gdb** well. It is vitally important for coming projects and labs.

3 gdb

gdb is a debugger that you can use to step through your C programs and ensure that your code is behaving as you expect it to. The main tasks that gdb performs include:

- Starting your program
- Making your program stop when certain conditions are true
- Examining what has happened when your program has stopped
- Changing things in your program so you can experiment with correcting the effects of one bug and continue your program

This document will cover the first three of those tasks.

gdb is best run on an executable file that was compiled using the -**g** flag, which provides debugging symbols so that gdb can refer to variables by name and reference lines of source code. The syntax for running gdb on the executable **hello** is

```
gdb hello
```

This will start gdb and permit you to run gdb commands. Once you have started **gdb**, you may run your program by typing the following:

```
run <args>
```

Where **<args>** are the arguments you would supply just as if you were running it in the command line. If you want to restart the program, either in the middle or after it's finished, you may type **run** and it will run again using the same arguments you originally provided (you can change the arguments by supplying new ones.) The debugger is terminated with the gdb command **quit**.

Below is an explanation of several tasks you can perform with gdb that may be helpful in debugging.

3.1 Setting a Breakpoint

A breakpoint is a place in your code where you want the execution of your program to pause. You can set these before you run your program, when your program has exited, or when **gdb** has paused execution of your program.

To set a breakpoint in your program at the start of a function, use the gdb command **break [file:]function**. For example, if you wish to create a breakpoint at the start of the function **bar()** in the file **foo.c**, you would use one of the following commands:

```
break foo.c:bar
break bar
```

To set a breakpoint at line 6 of **hello.c**, you can use one of the following commands:

```
break hello.c:6
break 6
```

A breakpoint can be deleted by running **clear** in the place of **break**, or **delete <num>** on the breakpoint given by **num**.

Running **info break** will print a list of all the currently-set breakpoints.

3.2 Setting a Watchpoint on Local Variables

You may set a watchpoint on a local variable with the **watch** command. Like **break**, you can set these any time the program is stopped. Doing so will cause the program to stop if the value of that variable changes. The command to set a watchpoint on the variable **bar** is:

```
watch bar
```

If your program has multiple variables named **bar**, gdb will attempt to determine which one you mean based on the program's current location and variable scoping, so be careful. You can't set a watchpoint on a variable that is currently out of scope.

info watch, another use of the **info** command (among many), will print a list of all the current watchpoints.

3.3 Setting a Conditional Breakpoint

You may want to have the program stop on a local variable only in certain cases. gdb will let you do that with conditional watchpoints and breakpoints. For example, if your program has a

variable **foo**, **watch foo if foo==5** will break only when **foo** is 5. Again, be sure to think about variable scoping!

3.4 Stepping and Continuing

Once your code has stopped (but not terminated), you have several options how to proceed.

- **next** will execute the next line of code, including the entirety of any functions called in that line.
- **step** will execute the next line of code, stepping into and stopping at the first line of any function that line may call.
- **continue** will resume execution of your code until the next breakpoint or the termination of the program.
- **finish** will resume execution of your code until the next breakpoint or the current function returns; if it is the latter, this command also prints the function's return value (if any).

To repeat the previous command, you can just send a blank line and gdb will execute the last-executed command. This is very useful if you want to quickly step through your program.

3.5 Navigating Frames with Backtrace

As you may recall from the Life lab, **backtrace** is a utility for printing out the call stack (similar to what happened when you didn't catch an exception in Java). When your code is stopped, you may view the values of all local variables (of not only the current function but of those that called the current function) at that point in execution by running the **backtrace full** command. Note that this will print only the memory address associated with a pointer variable, not the value at that address.

The call stack is divided into different sections associated with each function, which are called frames. You can use gdb to select a certain frame of the call stack by running the command **frame <n>**, where **n** is the frame number, which starts at 0 for the most recent one. You can also move to the next frame by running **up <n>**, where **n** is the number of frames you want to move up. Similarly, you can show a lower frame by running **down <n>**. The **backtrace** command can show only the **n** most recent frames by calling **backtrace <n>**.

3.6 Printing Variables

When your program is stopped, you may print the value of any variable by using the **print** command. You can also use print to show the values of more complicated expressions. For example, if **bar** is a variable of type **int** with value 5, the following will print 13:

```
print 2*bar + 3
```

gdb knows the type of variables, so they will be printed in a manner that makes sense. Notably, **`char*`** variables are printed as strings, and printing an array will show every value in the array as long as it's viewed from the same frame as it's declared in.

3.7 TUI (Text User Interface) Mode

`layout` is a terminal interface which allows the user to view the source file while debugging. The TUI mode is enabled by default when you invoke gdb as **`gdb tui`**. You can also switch in and out of TUI mode while gdb runs by using various TUI commands and key bindings, such as **`tui enable`** or **`Ctrl-x Ctrl-a`**. To disable TUI mode, you can type **`tui disable`**. If your TUI display has issues, **`Ctrl-l`** or disabling and reenabling it should fix it.

Once you are running TUI mode, there are several commands you can use to change the display. One of them is **`layout <name>`**. The **`name`** parameter controls which additional windows are displayed. To display the source code, you would use **`layout src`**.

When you have multiple windows open, you can then use the command **`focus <name>`** to switch focus between windows. For example, use **`focus cmd`** to make the command window active for scrolling (so that when you use the arrow keys you scroll through gdb commands instead of moving the text window). Similarly, **`focus src`** will make the source window active for scrolling.

3.8 Abbreviations

gdb fortunately accepts abbreviations for many of its commands. Instead of typing the full command name, you can instead use the following:

- **`r`** instead of **`run`**;
- **`b`** instead of **`break`**;
- **`d`** instead of **`delete`**;
- **`i`** instead of **`info`**;
- **`wa`** instead of **`watch`**;
- **`n`** instead of **`next`**;
- **`s`** instead of **`step`**;
- **`c`** instead of **`continue`**;
- **`bt`** instead of **`backtrace`**;

- **p** instead of **print**; and
- **q** instead of **quit**.

4 The Task

4.1 Diagnose

gdb is very useful for finding bugs that may fall into all sorts of different categories. Here is a non-exhaustive list of some common bug types and some ways to show them using **gdb**:

- Pointer vs. variable: Sometimes an operation is performed on a pointer when it should be performed on a variable (or vice versa). For instance, you might intend to increment a value pointed to by some pointer but instead mistakenly increment the pointer itself. This can be shown by printing a variable before and after it's supposed to be modified.
- Null pointer dereference: Memory at location **0** is not accessible. If you have a **NULL** pointer, this is exactly what happens, as **NULL** is a macro for 0! This leads to a segmentation fault. Check your pointers to find this bug.
- Array out of bounds: Memory outside of an array can be garbage, or worse, some other variable in your program! Make sure your index isn't greater than or equal to the array length through printing variables in **gdb**.
- Stack overflow: The stack keeps growing due to some kind of infinite loop. This also leads to a segmentation fault! Use **backtrace** after a segmentation fault, and if there are hundreds of recursive calls, then the stack has probably overflown.
- Mistyped conditional: Sometimes the wrong branch of an if/else statement is being taken or a loop doesn't terminate correctly. You can type C expressions in **gdb**, so perhaps evaluating the conditional directly may be appropriate. You can also set breakpoints or step through to make sure you're hitting the lines you expect to hit.

You will now use your newfound knowledge of **gdb** to debug a few programs we've written for you. We're looking for specific **gdb** commands for each in order to show each error - the emphasis is your ability to use **gdb**, NOT your general knowledge of C. The bugs in the code are mostly some variation on the types of errors above, so please make sure to tell us which kind and show it in **gdb**. Although these programs are relatively straightforward and may even be debugged by inspection, we'd like to emphasize specific **gdb** techniques so they can be used when the programs become much more complex.

- **file1**: prints out the lexicographically sorted version of its argument string. (Hint: C has no recursion limit, just a finite amount of memory.)

- **`file2`**: takes each of its command line arguments (numbers), increments each by one, and prints them back out. (Hint: There are some pointer shenanigans. How can we show the pointers are valid, or what they point to are valid?)
- **`file3`**: takes an initial "command" argument of 0 or 1. If the command is 0, it will check if the following two arguments are anagrams of each other. If the command is 1, it will take the next argument and print out an anagram of it. (Hint: There are a number of issues related to looping and memory access, including one bug that doesn't fall neatly into the categories above.)
- **`file4`**: sums the last 3 command line arguments (numbers), and prints the result. (Hint: More pointer shenanigans, this time with a more complex program flow. A variable is defined in one place, and it is messed with in a lot of places under different parameter names. Figuring out how it got messed up isn't as important as showing it did get messed up.)

Each of these programs has its own share of bugs. Unless otherwise stated, assume the bugs may be in any function in the program.

Good luck! Once you have completed this part of the lab, you may call a TA over and have them verify your answers so that you can proceed to the second part of the lab. In order to be checked off, you may paste your GDB output to a text or some other file for the first 3 programs and explain what type of error(s) occurred for each program from the output, but we expect you to debug **`file4`** live in front of us. Do not continue until you have verified your answers with a TA!

Hint: use `gdb` to set a breakpoint on main to start. You may want to look over the **`tools.pdf`** document, but **`gdb`** will be the most useful tool for this lab.

At this point, you may call a TA over to be checked off for the lab. You're done - feel free to leave or continue as you desire.

4.2 Fix

Try your hand at fixing these programs. This will be a true test of your new `gdb` prowess. The TAs encourage you to work through the programs in sequential order.

All of these programs are command line programs whose number of arguments goes into **`argc`** and whose argument array goes into **`argv`**. When parsing command line arguments, make sure that these programs know what kind of arguments to expect, and make sure that they never cause segmentation faults, even if the user entered unexpected arguments. This is an important habit for writing CLI programs. Since **`main()`** populates **`argv`** at runtime and C has no bound checking, we must take care to only reference memory that is initialized. For example, if

the command line reads **`./my_prog 1 2 3`**, any reference to **`argv[4]`** in your program may cause a segmentation fault. Good luck!