Lab 02 - Tools Lab

Out: Monday, September 15, 2014

1 Introduction

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 CS033, 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.

2 Assignment

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 and in the lab stencil. The most important tool for this lab, `gdb`, is described below.

To get started, run the command `cs033_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 the point of this lab, however, 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. 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 Running Your Code

When you run `gdb` you will be presented with the gdb interactive prompt `(gdb)` where you can type commands to `gdb`. To run your program in `gdb`, type `run` at the `gdb` prompt. Anything on the line after `run` will be treated as a program argument. The program will go until it terminates or reaches a breakpoint.

3.2 Setting a Breakpoint

A breakpoint is a place in your code where you want the execution of your program to pause. 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 the following command:

```
break hello.c: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.

3.3 Setting a Watch Point on Local Variables

You may set a watch point on a local variable with the `watch` command. 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.

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

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 Printing Local Variables with Backtracing

`backtrace` is a utility for printing out the call stack (similar to what happened what 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.

3.6 Evaluating and Printing Arbitrary Expressions

When your program is stopped, you may print the value of any expression by using the `print` command with any expression. For example, if `bar` is a variable of type `int` with value 5, the following will print 13:

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

3.7 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

You will now use your newfound knowledge of gdb to debug a few programs we've written for you.

- *file1*: 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.
- *file2*: takes each of its command line arguments (numbers), increments each by one, and prints them back out.
- *file3*: sums its command line arguments (numbers), and prints the result.
- *mergesort*: prints out the lexicographically sorted version of its argument string.

Each of these programs has its own share of bugs. Unless otherwise stated, like in the `merge()` function of *mergesort.c*, 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. 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 use any subset of the tools described in the *tools_writeup.pdf* document — some will be more helpful than others. You probably will not need to employ all of the tools, but `gdb` is not the only tool that will be useful!.

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 or at least identifying the *specific* issues each of them are facing. This will be a true test of your new `gdb` prowess. The TAs encourage you to work through the programs in sequential order.

- *file1* is a longer program with several issues that should be more readily apparent to you as you follow execution in the debugger than some of the following programs.
- *file2* is another command line program with an issue relating to pointers.
- *file3* is an example of a class of particularly tricky errors to catch that we hope you can spot.
- *mergesort* implements an in-place recursive mergesort algorithm. In particular, it exemplifies one of the risks that can come with deep recursion. Note that C has no recursion limit, just a finite amount of memory.

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.