Lab 02 - Tools Lab

Out: September 17-18, 2012

1 Introduction

Much of the code you have previously written was 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 binaries; then you will receive the source code of each binary and fix the bugs within.

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`.

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

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` 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.

The CS033 students Jasmine and Aladdin have just learned C, and are eager to try out their new knowledge. To demonstrate their prowess, Jasmine and Aladdin have written four simple command-line programs. They are:

- *absval*: takes an even number of program arguments, generates a complex number for each argument pair, and prints the absolute value of each of those complex numbers.
- *gcd*: takes 2 arguments and prints their greatest common divisor[1].
- *increment*: increments each integer argument and prints the results.
- *shout*: shouts its arguments back at the user.

Each of these programs exhibits weird C bugs that Jasmine and Aladdin just can't fathom, and so they've asked you to help them track down what might be wrong. Additionally, it seems Jasmine and Aladdin have refused to provide you with their source code, so you'll have to diagnose the problem off of just the binaries. All they know is that their algorithms for *absval* and *gcd* are correct, and that their `main()` functions are all written perfectly - the bugs must be elsewhere in their code.

Good luck! Once you have completed this part of the lab, call a TA over and have them verify your answers so that you can proceed to the second part of the lab. You will not be able to continue until you have verified your answers with a TA. Once a TA has verified your answers, run `cs033_install lab02_repair` to get the stencil for the second part of the lab.

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 need not employ all of the tools.

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

[1] http://en.wikipedia.org/wiki/Greatest_common_divisor

4.2 Fix

After demonstrating your prowess with debugging tools, Jasmine and Aladdin have relented and humbly presented you with their source code for repair. However, being so impressed with your command-line skill, they have deigned to visit additional requests upon your most skilled mind.

Jasmine and Aladdin want their code to compile without any compiler warnings whatsoever. They also want their compiled binaries to have debugging symbols and be optimized for binary size (their magic carpet has a very small hard disk). They don't know how to do any of this, but do know that their *absval* program won't compile without the flag `-lm`, which links C's math function library.

Good luck.

5 Getting Checked Off

To get checked off for your lab, raise your hand and a TA will come to you. Be sure to get checked off for both parts of the lab; we won't release the source code to you for the second part of the lab until you've completed the first part.