Computer Science 381
Programming Unix in C
The College of Saint Rose
Winter Immersion 2016

Lab 6: Debugging

Due: Monday, January 4, 2016

In this lab, you will get a bit more practice with pointers in C and learn how to use the GNU Debugger, `gdb`.

Debugging in C

In this lab, we will take a quick look at some tools for debugging that you might find helpful the rest of the way.

Java generally generates an exception when a program crashes and produces a useful stack trace that, in most cases, directs you to the exact line of code that caused the failure.

C is not nearly that helpful. In a Unix environment, C programs that crash typically generate errors such as "Segmentation fault", "Bus error", or "Floating exception", with no indication of exactly what the ultimate problem was or which statement caused it. We need some tools and techniques.

Some Buggy Programs

We start by writing a few programs that (intentionally) demonstrate some of the more common bugs that crop up in C programs.

Write short C programs that demonstrate these common C programming errors (2 points each):

- A program that allocates an array of 10 `int`s on the stack, but which has an initialization loop that sets values in the first 20 slots of the array (only 10 of which actually have memory allocated for them). Call this program `arrayindex.c`.
- A program that tries to assign a value to an uninitialized pointer to an `int`. Call this program `badpointer.c`.
- A program that tries to assign a value to a pointer initialized to `NULL`. Call this program `badpointer2.c`.
- A program that leaves off the `&` for the destination for an `int` parameter of a `scanf` function call. Call this program `badscanf.c`.
- A program that prints out values of $\frac{1}{n}$ for n from 10 down to 0. Compute as 1.0/n to force floating point division. Call this program `dividebyzero.c`.
- A program that computes the `int` average of a series of `int`s presented on the input. Stop reading numbers and report the average as soon as the `scanf` fails. Do **not** add error

checking that makes sure at least one number was successfully entered. Call this program `average.c`.

Question 1:
List any compiler warnings you see when you compile the above programs. (2 points)

Question 2:
List any run-time errors you see when you execute each of the above programs. For `average.c`, input a non-numeric value for the first number to force a division by zero. (5 points)

Question 3:
For your `arrayindex.c`, change the upper bound of your initialization loop to stop at 9, 10, 11, ..., 20, and indicate which result in crashes. If you program crashes, give the error message. (3 points)

Help from the Compiler

While C is a very forgiving language (it is often said that the language provides you many ways to shoot yourself in the foot), compilers can sometimes detect likely errors and report them as warnings. With the `gcc` compiler, we can enable all possible warnings by specifying the `-Wall` flag. For example, you can compile `arrayindex.c` with a command like:

```
gcc -Wall -o arrayindex arrayindex.c
```

Question 4:
Recompile each program with the `-Wall` flag. Do you get any additional compiler warnings? (2 points)

Most of the warnings reported by `-Wall` are things that should be addressed. It is strongly recommended to use it for all programs and to fix all reported warnings unless you have a good reason to believe that " you meant to do that".

The Debugging Printout

As is the case in many programming languages, a very common (and usually quite effective) debugging tool in C is our friend `printf`. Having printouts at strategic locations can help figure out which statement is generating an error.

In your `arrayindex.c` program, add a `printf` inside the intialization loop that prints each value of `i` before the assignment of a value to `a[i]`. Don't print a new line (`'\n'`) until outside the loop. Change the bound of your loop to 10, then 15, then 20, then 100, then 1000, then 100,000, and run each several times.

Question 5:
Describe your results from the above experiment. (3 points)

Part of the problem demonstrated here is that C uses "buffered" output. At times, your printout may not have completed before the program crashes, and your output never shows up. We can avoid the effect by printing to standard error rather than standard output. `printf` always prints to standard output (defined in C as `stdout`), but `fprintf` can also print to standard error (`stderr`):

```
fprintf(stderr, "%d", i);
```

It is often helpful to use `fprintf` to `stderr` when using debugging printouts to zero in on the line that is causing an error in your program.

Question 6:
Re-run the previous experiment with printouts going to `stderr` instead of `stdout`. Describe the differences in the output. (3 points)

The GNU Debugger

The best way to find more insidious bugs is to run your program in a symbolic debugger. With a debugger, your program can be stopped and started, you can step line by line through, you can print values of variables, and print stack traces when your program has stopped (or crashed).

The GNU debugger, `gdb`, is one such debugger. To make `gdb` useful, you will need to compile your program with the `-g` flag, which tells it to keep symbolic information that will allow `gdb` to take addresses of variables and machine instructions and map them back to your source code.

On mogul, the following would work:

```
gcc -g -o badpointer2 badpointer2.c
gdb badpointer2
```

then at the `(gdb)` prompt, type `run`.

When your program encounters an error, you can use commands such as `where` to see where in the source code your program was executing when it encountered the error, and `print` to examine contents of variables.

Question 7:
Give the output of `where` and `print` for your `badpointer2.c` program when it crashes when running inside `gdb`. (4 points)

Question 8:

Find one or two of the many useful tutorials on `gdb` available through a web search. (I find this one to be quite nice.) Experiment with breakpoints and the `continue`, `step`, and `next` commands. You might want to use one of your previous (longer) programs for this. Copy and paste from your terminal window showing your use of these commands. (6 points)

Submission

Please submit all required files as email attachments to *terescoj@strose.edu* by Monday, January 4, 2016. Be sure to check that you have used the correct file names and that your submission matches all of the submission guidelines listed on the course home page. In order to email your files, you will need to transfer them from mogul to the computer from which you wish to send the email. There are a number of options, including the `sftp` command from the Mac command line.

Grading

This lab is graded out of 40 points.

Grading Breakdown	
Lab questions	28 points
Programs with intentional bugs	12 points
Total	40