Hack 9.0

Debugging
Computer Science I

Department of Computer Science & Engineering
University of Nebraska–Lincoln

Introduction

Hack session activities are small weekly programming assignments intended to get you started on full programming assignments. Collaboration is allowed and, in fact, *highly encouraged*. You may start on the activity before your hack session, but during the hack session you must either be actively working on this activity or *helping others* work on the activity. You are graded using the same rubric as assignments so documentation, style, design and correctness are all important. This activity is due at 23:59:59 on the Monday following the hack session in which it is assigned according to the CSE system clock.

Problem Statement

Debugging is an essential skill for any software developer. The novice approach of using debug print statements is inefficient, error-prone, and unreliable in general. Real debugging requires a more sophisticated tool called (surprisingly) a *debugger*. In this hack, we'll introduce you to the basic usage of GDB, GNU's Debugger tool.

Starter code for this hack can be found in the following repository:

`https://github.com/cbourke/CSCE155-Hack9.0`

Basic Walkthrough

We've provided a small program (see `arrayDebug.c`) that randomly generates an array of integers and computes their average. However, the program contains several bugs. Follow the instructions below to use GDB to inspect this program and find and fix these bugs. To do this hack, you'll need to work on CSE or have GDB installed on your own computer.

Bug, The First

1. To get started, we'll need to compile the program specifically for GDB by using the `-g` flag:

   ```
   gcc -g arrayDebug.c
   ```

   Using this flag means that GDB will preserve function and variable names and line numbers for the debugger. Normally this information is not retained in a compiled program. *Note*: if you use the `-Wall` flag, you may want to omit it in this walkthrough to avoid spoilers.

2. Take a quick look at the code to get an idea of what the program is trying to compute, but do not change anything yet. Try running the program; it takes 1 command line argument: an integer representing the size of the random array you want to generate. You'll want to keep it small, say:

   ```
   ./a.out 5
   ```

   You'll see that the program crashes with a segmentation fault. There is little information to go on, so we'll need to fire up a debugger and see what's going on.

3. To start the GNU debugger running your program with command line arguments you can use:

   ```
   gdb --args a.out 5
   ```

   Alternatively, you can start GDB without args using `gdb a.out` and once it has started, you can set (and reset) the argument(s) using `set args 10`

4. Once GDB has started, you'll be in an interactive "shell" in which you can give GDB (and your program) commands. Start by having GDB run your program by typing:

   ```
   run
   ```

   which will run your program and *pause* its execution when it gets to the segmentation fault.

5. GDB will display the line of code at which the error occurred. However, it will be more helpful to see the entire source file. To do this use the Text User Interface

(TUI) mode by typing:

`layout next`

Using the up and down arrows, you can scroll through the source file. Occasionally, the display may glitch. Don't panic, simply type:

`refresh`

and the screen will refresh itself. As a short-cut you can use control-L to refresh.

6. Examine some of the variable values to investigation what the issue is. To print a variable value, use:

   `print i`

   `print n`

   To print the contents of an array, you need to dereference it and provide GDB with its length. Example:

   `print *a@5`

   In this particular case GDB will be unable to print the contents of the array. Obviously something went wrong before the program got to this point.

7. To find out what went wrong with the array, we need to rerun the program and examine its execution *before* we got to the segmentation fault. To do this, set a *breakpoint* in the program. A breakpoint is a line or point in a program where the debugger will pause execution from which you can continue execution in a step-by-step manner. Set a breakpoint at the `randomArray` function where the array was created:

   `break randomArray`

   Restart the program by typing `run` and confirming.

8. GDB will pause execution at the beginning of the `randomArray` function. You can now step through the program line by line by typing `next` or just simply `n`. Note that if you want to resume execution, you can use the `continue` command.

   a) Step through an iteration of the for loop and print the array: `print *a@5`

   b) Step through the second iteration and print it again: `print *a@5`

   c) This can get tedious, so you can set a watchpoint that will print a variable any time it changes. Set a watch point for the array: `watch *a@5`

   d) Step through the rest of the iterations and watch the array get filled with random values.

   e) Step through the return of the function; when it gets back to the `main` function, print the contents of the array again: `print *arr@5` and you'll see that you cannot access the memory there.

9. Quit out of GDB using `quit` and edit the code to fix this bug (hint: what did the `randomArray` return?) and recompile.

Bug, The Sequel

1. Run the program again from the command line to see if it works. It should run but it probably won't give correct results (on some runs it may, but on most it will not). Start GDB again but this time use the following to immediately start it in TUI

   ```
   gdb -tui --args a.out 5
   ```

2. The problem probably lies with how we're computing the average, so set a breakpoint either at the start of the `average` function:

   ```
   break average
   ```

   or at a particular line within that function, example:

   ```
   break 54
   ```

3. Set another break point at the end of the function:

   ```
   break 58
   ```

   Note that you can set a breakpoint at a particular line in any file using the syntax:

   ```
   break fileName.c:42
   ```

4. Now, you can either step through the execution of the loop and/or set a watchpoint on the `sum` variable or you can skip ahead to the end of the loop by using `continue` which will resume execution until the next breakpoint.

5. Print the `sum` variable's value to see if it matches the value you expect. Or:

   - You can print all local variables using

     ```
     info locals
     ```

   - You can print all the parameter values using

     ```
     info args
     ```

   - You can print the value of expressions as well, example:

     ```
     print sum / n
     ```

   - Since return values are not stored in variables, another thing you can do is to use

     ```
     finish
     ```

     which will continue execution until the function returns and then prints the returned value.

6. Fix the bug(s) that you were able to identify and recompile/rerun the program until you are certain that all bugs have been addressed.

The Main Show

Now that you have some familiarity with GDB, you'll use your new debugging skills to debug a full program. We have provided source files and a makefile that builds a tic-tac-toe game project. The game allows a player to select several different modes including a 2-player game, a 1-player versus the computer (which makes random moves) and a 1-player versus a "smart" computer.

Similar to the previous part, we have introduced several bugs (6 to be exact). Run the program and use GDB to walk through section(s) of the code you believe to contain bugs and fix them all until the program runs without any problems.

Some tips/hints:

- Write down the sequence of inputs/moves that led to a crash/bug so that you can *reproduce* the sequence when debugging and so that you can retest the sequence when you think you've fixed the bug.
- When you observe a bug, describe exactly how it manifests, this will help you form a *hypothesis*
- Set breakpoints and watchpoints so that you can trace through the program to points where you believe bugs might be located.
- Try to fix only one problem at a time and keep track of your fixes. In a real project you would do so through a bug tracking system and commit messages, but for this hack you may do so through comments.
- Keep in mind that bugs may manifest themselves later in the program when the problem lies earlier in the program.
- You can get a bigger picture in GDB by printing out a full stack trace using

  ```
  backtrace full
  ```

Instructions

- You are encouraged to collaborate any number of students before, during, and after your scheduled hack session.
- Handin all the (corrected) files to webhandin and use the webgrader to verify that all bugs have been fixed.
- Each individual student will need to hand in their own copy and will receive their own individual grade.