CSCE 155 - C

Lab 08 - Debugging

Dr. Chris Bourke

Prior to Lab

Before attending this lab:

1. Read and familiarize yourself with this handout.
2. Review the following free textbook resources:
   - GDB Tutorial http://web.eecs.umich.edu/~sugih/pointers/summary.html
   - GNU's GDB Tutorial http://dirac.org/linux/gdb/

Peer Programming Pair-Up

To encourage collaboration and a team environment, labs will be structured in a *pair programming* setup. At the start of each lab, you will be randomly paired up with another student (conflicts such as absences will be dealt with by the lab instructor). One of you will be designated the *driver* and the other the *navigator*.

The navigator will be responsible for reading the instructions and telling the driver what to do next. The driver will be in charge of the keyboard and workstation. Both driver and navigator are responsible for suggesting fixes and solutions together. Neither the navigator nor the driver is "in charge." Beyond your immediate pairing, you are encouraged to help and interact and with other pairs in the lab.

Each week you should alternate: if you were a driver last week, be a navigator next, etc. Resolve any issues (you were both drivers last week) within your pair. Ask the lab instructor to resolve issues only when you cannot come to a consensus.

Because of the peer programming setup of labs, it is absolutely essential that you complete any pre-lab activities and familiarize yourself with the handouts prior to coming

to lab. Failure to do so will negatively impact your ability to collaborate and work with others which may mean that you will not be able to complete the lab.

1 Lab Objectives & Topics

At the end of this lab you should be familiar with the following

- Be able to better understand the errors generated by the GCC compiler
- Determine the types of errors that occur in a program
- Debug a program using GDB (the GNU Project Debugger)

2 Background

During these first few weeks of classes, you've likely had an error in one of your programs. The error might have been something that made your program produce incorrect output, crash while it was running, or even fail to compile at all. Unfortunately, errors like these are common, and it's a rare (and fantastic) moment when code gets written the first time without any errors. Fortunately, because errors are so common, there are a myriad of tools designed to ease the pain of correcting them. Errors tend to fall into one of three categories: compilation errors, runtime errors, and logic errors.

Compilation Errors

Compilation errors occur when the compiler (GCC in our case) encounters something that it doesn't understand in the code you're attempting to compile. This will cause the compiler to stop and print a message to the screen describing the problem that it encountered. An example might look something like:

```
Lab08.c:5:8: error: 'carCost' undeclared (first use in this function)
Lab08.c:5:8: note: each undeclared identifier is reported only
        once for each function it appears in
```

These messages can sometimes be cryptic, but they give the information needed to find and correct the error. There are several different sections in the error, each separated by a colon.

- `Lab08.c`: The file that the error was encountered in. This is crucial in larger programs that use several different source and header files to keep the code organized.

- `5`: The first number after the file name contains the line number in the file that the error can be found on. So now we know that if we go to `Lab08.c`, we'll find the error on (or around) the fifth line.
- `8`: The column that the error is found in (i.e., how many characters (including spaces) precede the error on the line).
- `error/warning/note`: The type of problem encountered. Errors will stop the code from compiling. Warnings will compile, but still need to be fixed since the warning is likely trying to prevent you from doing something that is legal as far as C is concerned, but may not produce the desired result. Notes are messages from the compiler regarding the previous warning or error message.
- The rest: A brief description of the cause of the problem. In this case, the error is that the variable was undeclared. GCC encountered this variable for the first time on line 5, but it doesn't have a type to identify it. The note is to let you know that there may be other uses of `carCost` in the same function, but to cut down on tons of output, only the first time it was used undeclared is noted.

Note that the error may not be on exactly the line number given, but will still be associated with the error in some way. For example, compiling the following code:

```
1  int main(int argc, char **argv) {
2    int catAge
3    catAge = 6;
4    return 0;
5  }
```

will cause GCC to output this error message:

```
cat.c: In function 'main':
  cat.c:3:8: error: expected '=', ',', ';', 'asm' or '__attribute__'
        before 'catAge'
  cat.c:3:8: error: 'catAge' undeclared (first use in this function)
  cat.c:3:8: note: each undeclared identifier is reported only
        once for each function it appears in
```

The error is supposedly occurring on line 4, but `catAge = 6;` is a valid and error free C statement. However, the description says that it was expecting one of those symbols `'=', ',',` etc. *before* `catAge`. The thing preceding `catAge` is the declaration of `catAge`, which is missing the expected `';'` mentioned in the first error. Fixing this also fixes the second error mentioned because the variable `catAge` has now been correctly declared.

This also applies to errors with unbalanced brackets, `{ }` and parentheses `( )`. It may

find the first of the pair but be missing the last one, which will result in the compiler assuming that there's a problem on the last line of the file. Be careful in this case, because that may not (and likely isn't) the place where the missing parenthesis or curly brace should go. Proper indentation helps immensely when trying to find the missing parenthesis/brace.

In most editors, the line number is not displayed by default. The following is a brief description regarding how to show the line numbers in the most common editors used in this class.

- Vim/vi: Type `:set number` while in normal mode.
- Emacs: `M-x linum-mode`. To enable this on startup for all file types, place `global-linum-mode 1` in your `.emacs` file in your home directory.
- Nano/Pico: Line numbers can't be displayed along the margins as they would in Notepad++ , Vim, or Emacs, but you can view the current line that the cursor is placed on by pressing CTRL+C.
- Notepad++: Should be enabled by default.

Runtime Errors

Runtime errors are errors that cause the program to crash (i.e., end in an unexpected place and manner). Depending on the operating system you're running the program on, a variety of error messages might be displayed. Dividing by zero in C is unhandled: it might cause the program to stop, or it could keep running as if everything were fine, but it's still an example of a runtime error.

A common runtime error is a segmentation fault (segfault). This occurs in several situations, such as when the program tries to write/access memory that it doesn't own or when a buffer overflow occurs (which typically happens when you try to write to memory outside the bounds of an array, for example).

Forgetting to place an ampersand in front of the variable in a `scanf` typically also causes segmentation faults. The program will try to write the scanned information to the address in memory represented by the number stored in the variable being scanned to, rather than the address of the variable itself.

The compiler will not catch these errors, but this lab will give you a couple of methods to help determine where and why the error occurred.

Logic Errors

Logic errors are errors that cause the program to operate incorrectly (such as producing incorrect output) but will compile and run without error. An example of such an error

would be writing a function `add` in the following manner:

```
1  int add(int num1, int num2) {
2    return num1 * num2;
3  }
```

The code above would compile and execute without error, but obviously when you call a function named `add`, you expect it to return the sum of the numbers and not the product.

3 Activities

Clone the code for this lab from GitHub using the following URL: https://github.com/cbourke/CSCE155-C-Lab08. The project contains a version of Conway's Game of Life, but contains errors. The basic rules of this game can be found here: http://en.wikipedia.org/wiki/Conway%27s_Game_of_Life.

3.1 Correcting Compilation Errors

1. Try compiling the program using the supplied makefile (type `make` in the same directory as the makefile). Several errors will be produced. Correct each of these compilation errors.
2. Fill out question 1 on the worksheet.

3.2 Correcting Runtime Errors

1. Try running the program created in the previous step (`./gameOfLife.x`). The program should crash due to a segmentation fault. The following steps will show you how to quickly find and correct segfaults using gdb.
2. Close the currently running `gameOfLife` program by using Control-C.
3. Type gdb on the command line. GDB is typically used in conjunction with the `-g` flag in gcc. The flag produces debugging symbols that GDB can use to aid in the debugging process (specifically, it allows for things like showing which line of code the error occurred on and retaining variable names in the debugger). The `-g` flag has already been added to the compilation command in the makefile, so nothing extra needs to be done.
4. Now we have to tell GDB which file we want to debug (i.e., the program we generated). On the GDB command prompt, type `file gameOfLife.x`. You

should see output that looks something like `Reading symbols ... done`

5. Now run the code from within GDB by typing run. The program will crash in the same place that it crashed before, only now we can tell exactly what line of code caused it to crash. Type `backtrace` at the GDB prompt to see the list of functions on the call stack (that is, all the functions that were called immediately preceding the crash). For this first error, your output should look like:

```
#0  0x0000000100001056 in placeBeacon (gameBoard=0x0, cell=...)
        at gameOfLife.c:52
#1  0x0000000100000f7c in main (argc=1, argv=0x7fff5fbff318) at
        gameOfLife.c:21
```

   The bottom function (`main`) is the first function that was called. `placeBeacon` is the second function, and also where the crash occurred (line 52). However, this is a bit deceiving. Notice that the first argument to `placeBeacon` is the address `0x0`, which corresponds to `NULL`. On line 52, this address is dereferenced, but clearly something happened before `placeBeacon` that caused the `gameBoard` to be `NULL`.

6. To find out where the error originated from, we'll step through the program line by line. First, we'll tell GDB to insert a breakpoint somewhere, which will temporarily pause execution of the code. Type `break main` to tell GDB to stop execution as soon as the `main` function is called.

7. Now type `run` again, and restart the program. You can use the commands `next` and `step` to continue execution line by line. The `next` command will execute entire functions (i.e., it won't execute the function line by line), while `step` will "step into" a function, and execute each line of that function one at a time. Use `next` until you arrive at line 19, and view the contents of the board variable by typing `print board`. Notice that even after `createGameBoard` was executed, `board` remained `NULL`. Now we've can narrow the problem down to the function `createGameBoard`.

8. There are still several runtime errors to correct. Many of them will be identified slightly more directly by GDB (it will report the line number a segfault occurs on), while one or two will require you to step through the program while examining variables and their addresses.

9. Reopen GDB, type `make` at the GDB command prompt, and rerun the program with `run`. Once again, type `backtrace`, identify where the program crashed, fix the error, and continue on until the program no longer crashes (hint: common reasons for segfaults are `NULL` pointer errors, out-of-bounds array access, and accessing previously freed memory).

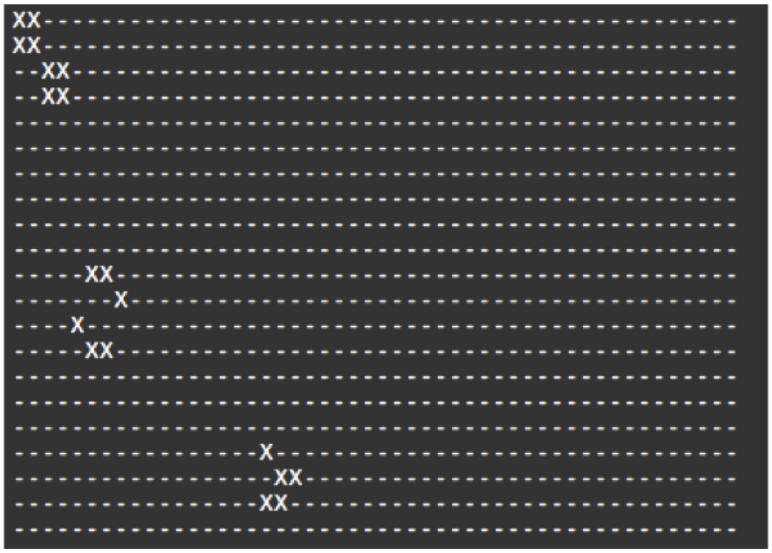

Figure 1: Game of Life Depiction

3.3 Correcting Logic Errors

1. There are still several errors in the program, though none will be as obvious as compilation or runtime errors. When corrected, the program should look something like Figure 1. The two shapes on the left will oscillate but not move, while the shape towards the bottom will move down and to the right.
2. Using the rules found on the Wikipedia page for the game (linked above) to fix the program. It's up to you whether or not to use GDB to help you find these errors (most are isolated to a single function).
3. Answer questions 2 and 3 on the worksheet.

4 Advanced Activity (Optional)

Experiment with GDB. There are tons of various commands you can use to help you debug your code. Take a look at http://sourceware.org/gdb/onlinedocs/gdb/index.html#Top (specifically the running, stopping, reverse execution, data, stack, and source links).

Emacs is an editor with some fairly advanced built in functionality. While there is a bit of a learning curve to Emacs (and Vim), the effort required to learn it will pay off in the long run. Try using Emacs as an editor for a while until you're comfortable with some of the basic commands. Use this website to help get you started. A cheat sheet, such as this one can be extremely helpful for a quick reference.

Now that you can move around Emacs a bit more comfortably, open today's files and try using some of the commands outlined here to use GDB within Emacs. Notice how the line of code currently being executed is displayed within the editor itself. This can be extremely handy when trying to debug a program.

Alternatively, most IDEs have a debugging facility built in. You can try redoing this lab in CodeBlocks using GDB through the CodeBlocks GUI.