Due: Thursday, February 2, 2017, 11:59:59 Midnight

Introduction

Today’s lab is going to introduce you to techniques designed to help you deal with compile time errors. These are errors in your code identified by the compiler. However, as you probably already know, just because your code compiles does not mean it will run correctly. Running your program could result in a segmentation fault, bus error, or incorrect output. All of these are called runtime errors. These programs are considered to have “bugs” in them and will need to be “debugged”.

Since you have experience with programming, you have probably discovered that as software projects get larger and more complicated, debugging becomes increasingly challenging. On your smaller projects, using the debugging technique of printing out variables probably was your preferred debugging method. However, printing out every variable simply doesn’t scale as projects get larger. As an example, if you were working on the iROAR programming team trying to isolate a bug in the registration process, you’d have a minimum of 17,000 students registering for multiple classes and labs. Printing out every variable would just give you screen after screen of information and bury you.

You may have been introduced to debugging using the **gdb** utility in a previous course. In this lab we discuss strategies to deal with compile time errors and provide a refresher on **gdb**. You will also be introduced to **Valgrind** a tool that can be used as a debugging tool as well as used to reveal memory leaks.

Lab Objectives

* Learn to read and handle compile errors
* Learn how to compile with clang which often provides more readable error messages
* Debug a C program using **gdb** (via breakpoints, checking the value of a variable during program execution, and tracing function calls)
* Use **Valgrind** to help you debug a program and understand memory management

Prior to Lab

* Review Dealing with Compiler Errors http://www.cprogramming.com/tutorial/compiler\_linker\_errors.html
* Review the gdb Cheat Sheet  
  http://darkdust.net/files/GDB%20Cheat%20Sheet.pdf
* Read the 10 Truths of Debugging https:

//www.cs.duke.edu/courses/spring04/cps001/notes/Debugging.pdf

Lab Instructions

**Part 1: Dealing with compile errors**

I am sure at some point you have had one or more compile error, probably heavy on the more, as least I have. Being able to understand and deal with these errors is crucial to becoming a successful programmer.

|  |
| --- |
| **Part 1**. First compile ***part1.c*** the following way:  **gcc part1.c functions.c –o part1a –Wall 2>error.err**  I am sure you went over this in 1010 or 1110, but 2> in the above means to print all error messages to a file called error.err. Now compile ***part1.c*** again using the following:  **clang part1.c functions.c –o part1b –Wall 2>error2.err**  ***Clang*** is a compiler similar to ***gcc.***  Open the error files and look at the differences. Often they will be the same, however, many times clang is more “user friendly” when describing errors.  I purposely incorporated several errors in this code to give you practice reading and dealing with compile errors. When dealing with compile errors, it is better to go to the top of the error list and work your way down the list. Once you fix an error or two recompile. Often, one error will precipitate another error, so fixing one may fix several others.  Your job is to correct all the errors in **part1.c, functions.c,** and **functions.h**. After you have corrected all errors, recompile the program using **gcc**. If there are linking errors, fix them. Once the program compiles with no errors, test it using the **input.ppm** file included. This program basically should read in and write back out a ppm file. |

**Part 2: Dusting off your gdb debugging skills and learning about Valgrind**

Some helpful **gdb** tutorials:

<https://www.cs.umd.edu/~srhuang/teaching/cmsc212/gdb-tutorial-handout.pdf>

<https://www.tutorialspoint.com/gnu_debugger/index.htm>

<http://www.dirac.org/linux/gdb/>

<http://cs.baylor.edu/~donahoo/tools/gdb/tutorial.html>

<http://www.thegeekstuff.com/2010/03/debug-c-program-using-gdb/>

Debugging is often the most frustrating part of programming. It’s important to develop good debugging strategies early on to help you deal with undesirable program behavior in the future. If you haven’t already, please read the 10 Truths of Debugging article.

The **gdb** utility is a powerful program that can help you isolate bugs in computer programs. Your instructors can attest that many students will come in to office hours saying, “My program seg faults and I don’t know where”. Using **gdb** will often guide you to the general location of your error. As a reminder, **gdb** allows you to:

* Start your program, specifying anything that might affect its behavior such as command-line arguments
* Make your program stop on specified conditions.
* Examine what has happened after your program has stopped running
* Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

We will work with **linkedList.c** provided for you. In a linked list data structure, information is contained within a “node” and each node points to the next node on the list. Linked data structures are examples of dynamic data structures and can easily grow as you have more data to add (as long as the computer has available memory!)

THIS LAB IS NOT ABOUT LINKED LIST. WE WILL LEARN MORE ABOUT LINKED LIST TOWARD THE END OF THE SEMESTER. PLEASE DO NOT GET HUNG UP ON THE CONCEPT OF LINKED LIST.

I put notes in this code to help you understand what is going on. I repeat, don’t get hung up on understanding linked list at this point.

For this portion of the lab, compile and run LinkedList.c. You’ll notice that the program crashes once you try to run it. Without looking at the source code:

* Recompile linkedList.c using the **gdb** flag: **gcc –g linkedList.c –o LL**
* Run **LL** in **gdb** and use **gdb** to isolate the error (see the tutorials for this).
  + Hint: spend some time in **gdb** and looking at your program variables in order to figure out where you should set breakpoints.

Personally, I sometimes find it hard to locate my problems in gdb (maybe because I don’t use it as much as I should). But for the most part, gdb will get you in the vicinity of the error. You need to learn to think logically about what could be the problem and follow the gdb leads. REMEMBER, IN MANY CASES, GDB IS NOT GOING TO POINT YOU TO THE EXACT PROBLEM.

|  |
| --- |
| If you have not successfully isolated the error using gdb. Try using Valgrind (instructions below) to locate the error.  Once you have isolated the error, create a copy of **linkedList.c** called **linkedListFixed.c** and fix the error within the code to allow the program to function properly. |

Valgrind is another debugging tool designed to help you locate bugs in your code as well as understand memory management. As with gdb, Valgrind is not perfect. But more importantly, programs run under Valgrind are really slow. I recommend running your program using Valgrind only when you want to check for memory leaks or when using it as a debugger.

Below is an example of a program that, while it does not have an error, it has memory leaks. Numbers 1 & 2 below demonstrated how to compile the program and run it using the valgrind tool.

/\* memoryLeak.c

example to use for valgrind

this program will reserve a block of memory

large enough to hold 100 characters \*/

#include <stdlib.h>

int main(void) {

char \*x = malloc(sizeof(char) \* 100);

return 0;

}



1. Compile the program using gcc –g memoryLeak.c -o memoryLeak
2. Run Valgrind via valgrind --tool=memcheck --leak-check=yes ./memoryLeak
3. View the following:



==16204== Memcheck, a memory error detector

==16204== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al

==16204== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info

==16204== Command: memoryLeak

==16204==

==16204==

==16204== HEAP SUMMARY:

==16204== in use at exit: 100 bytes in 1 blocks

==16204== total heap usage: 1 allocs, 0 frees, 100 bytes allocated

==16204==

==16204== 100 bytes in 1 blocks are definitely lost in loss record 1 of 1

==16204==

==16204== at 0x4C2B6CD: malloc (in /usr/lib/valgrind/vgpreload\_memcheck-amd64-linux.so)

==16204== by 0x400505: main (memoryLeak.c:11)

==16204==

==16204== LEAK SUMMARY:

==16204== definitely lost: 100 bytes in 1 blocks

==16204== indirectly lost: 0 bytes in 0 blocks

==16204== possibly lost: 0 bytes in 0 blocks

==16204== still reachable: 0 bytes in 0 blocks

==16204== suppressed: 0 bytes in 0 blocks

==16204==

==16204== For counts of detected and suppressed errors, rerun with: -v  
==16204==ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 2 from 2)

The highlighted lines in the center shows how much memory is lost and where in the program the lost memory was allocated.

There will be times when the --leak-check=yes option will not result in showing you all memory leaks. To find absolutely every unpaired call to free or new, you'll need to use the --show-reachable=yes option. Its output is almost exactly the same, but it will show more unfreed memory.

Another example:

|  |
| --- |
| bad.c is a small program that has a couple errors. By sight, you should be able to find these errors fairly easily.  Compile bad.c and then run it using Valgrind, see example above. Use the output from Valgrind to determine the problem with this file. Hint: There are two main issues.  Correct the problems, run Valgrind again to make sure you corrected all problems. Using comments within the code, explain what the problem was within this program. Rename the corrected file badFixed.c |

Additional Information:

* Valgrind tutorial: <http://www.cprogramming.com/debugging/valgrind.html>

Submission Instructions

Use handin (http://handin.cs.clemson.edu) to submit a tarred file called Lab3.tar.gz containing **linkedListFixed.c, badFixed.c, part1.c, functions.c and functions.h**

You need to provide a header in each of the files submitted. I am not going to explain what the header looks like; you should know by now.