

**Course Name**: Computer Architecture

**Course Number and Section**: **14:332:333:1a**

**Experiment**: [Lab # 2 – Introduction to C Programming Language]

**Lab Instructor**: Ali Essam Hameed Haddad

**Date Performed**: September 23, 2018

**Date Submitted**: October 08, 2018

**Submitted by**: [Christopher Cordi 182002852]

Electrical and Computer Engineering Department

School of Engineering

Rutgers University, Piscataway, NJ 08854

Exercise 1: Simple C program-

1. Purpose / Introduction / Overview

The purpose of this exercise is to manly get acquainted with the way macros work within a simple C code. The main idea is to learn how to use the compile function and to be able to change the macros within the code to get the required code to show.

2. Approach / Method

The first thing needed to do, is to make sure the file or code we are working with is saved somewhere within the computer. The easiest option is to save it to desktop. In our case we are working with the program *simple.c*.

After the file is saved, we open terminal. Next, on terminal, we need to change the directive in order to access the program so we can compile and use it to display an output.

To change the directory we need to use “cd /home/user/Desktop” in my case the desktop was were I saved the file. Next, we need to compile in order to display the program. Using the code “gcc -o simple simple.c” allows us to compile the *simple.c* program into machine code and rename it as *simple.* After, we are now able to display what the code carries out. Which is what is needed next.

Using the command “./simple”, will display exactly what is written within the code. We are then tasked to open the *simple.c*  file in order to get the intended output. The code clearly shows which values we are tasked with changing. After we make a change the best way to figure out if what was done was correct was to save the .c file and recompile, following the steps above again. Till the intended output is apparent.

3. Results

The result that I have found to work is to change the macros “V0-V3” to what is shown below. This will allow for us to get the intended output.

|  |  |
| --- | --- |
| OLD | NEW |
| /\* Only change any of these 4 values \*/  #define V0 0  #define V1 -1  #define V2 0  #define V3 0 | /\* Only change any of these 4 values \*/  #define V0 3  #define V1 3  #define V2 1  #define V3 3 |
| Busch Campus Center  Boo PENN STATE! | RU RU RU  Werblin Rec Center  Go RUTGERS! |

4. Conclusion / Summary

1.) What changing each of the four values did and what it meant.

|  |  |  |
| --- | --- | --- |
| Variable | Value | Description |
| V0 | 0->3 | Changing this value to anything greater than 0 allowed us to display RU, and the code allows the number to also specify how many times RU gets displayed. In our case we wanted it to be displayed 3 times, hence the value 3. |
| V1 | -1->3 | This is to specify which switch case to refer to and since we needed the output to display “Werblin Rec Center” we change the value to 3. |
| V2 | 0->1 | V2 is to display college of choice. Putting any positive value here results in the code to display “RUTGERS!” rather than “PENN STATE!” |
| V3 | 0->3 | V3 states if the value is 3 it will display “GO” if the value does not equal 3 it will default to display to “BOO” |

2.) The minimum number of distinct values needed for the preprocessor macros.

* If the need arose to have the minimum number of distinct values for the preprocessor macros we would be able to apply the code while only implementing 2 variables. In order for it to work out with 3 macros we would set V0=V1=V3=3 and V2=1. In this case we really only needed to use V0 & V2.

3.) The -o flag use in union with “gcc”.

* Using gcc with -o gives the ability to write the build output to an output file.

Code attatched

Exercise 2: Debugger

1. Purpose / Introduction / Overview

Debuggers are useful since it allows us to go step by step through our code being executed in the program. The benefit is so that what may seem almost impossible, since the code being executed goes line-by-line way quicker than we are able to comprehend, becomes possible and easy.

2. Approach / Method

The way we are tasked to go about using the debugger is to use the reference card to step through the entire program, we are also given tasks to follow. First, we must get into the debugger by following the steeps from exercise 1. Save the file *hello.c* to the desktop, use the gcc compiler function to compile the c code to machine language. Next, I ran the code using ./hello this allowed me to see what the code is outputting but not actually necessary. After, we are then going to implement the gdb command. First command is gdb hello once that runs we must complete the first task of setting a break point at main. In order to do so we use the command break main and that will set the breakpoint at main. After we need to use gdb’s run command which easy enough is just run, the last task is to use the single-step command which also is just as easy as plugging in step. The command run will allow us to go through the code and step will allow us to go line by line to see any faults in the system.

4. Conclusion / Summary

-Question 1 referenced in approach/method listed above.

-Question 2- A list containing the additional gdb commands.

1- In order to pass command line arguments to a program by using r arg#

2- To set a conditional break point, one in which will only occur if the condition is met, we can use the command $cond {breakpoint#} {expression} an example of an expression would be like “i==12”

3- To execute the next line of C code in the program after stopping at a breakpoint would be to use the command stop

4- If the next line of code is a function call then we use step

5- To resume the program after stopping at a breakpoint we use the command continue

6- To see the value of the variable we can use output “variable name” an example is “$output i”

7- To configure gdb to print the value of a variable after every step we can use the command display “variable name” ex: “display V0”

8- To print a list of all variables and their values use info variables which will display all global and static variables, although if you wanted to know of the current function use info locals

9- To exit out of gdb simply use command q or quit, then when the prompt asks [y]terminate or [n] to cancel hit y then enter and you have exited gdb.

Code not needed since no changes were made- but attached anyway

Exercise 3: Debugging a buggy program with GDB

1. Purpose / Introduction / Overview

Getting a taste for debugging, we have something wrong with the code we are given and it may seem to difficult to look within the code and understand everything that may be going on within the code. The debugger within terminal allows us to figure out what is causing the segmentation fault. When running the initial *ll\_equal.c* function we get the notice that there is a segmentation fault, we then must fix it.

2. Approach / Method

First things first when starting up a new code. That is to see what the original code does so using the command ./ll\_equal we see that there is a fault. This is just so we can make sure the prompt in the lab makes sense. In our case it does rain true. To solve this problem, we first go into gdb of terminal. When in gdb we first set a break main, then use the command run, after that we can step through the function in order to see what the problem within the code is.

When initially running through it this way nothing sticks out for us to nowhere the problem is, so we must go more in depth to figure out the problem at hand. After the initial run, I figured out to look at the variables as I’m running the program to see what the values of them where instruction by instruction. I used the command display from exercise 2 to see what the values of a and b where at every step of the code. After several steps through the code the problem was found. (a) was not set to a value therefore could not work properly.

3. Conclusion / Summary

The bug and my fix to the solution.

-The bug happened to be at the point where the while loop started, there is a value designated to “a” which is (a!=Null) but misses its value. When looking at terminal’s gdb function and running through the steps with display a and display b, we can see that the problem arose since node “b” was not defined. The fix to this was to change the code from “while(a != Null)” to “while(a != Null && b != Null)” this is so that we can define “b” as a variable with “a”.

Code included

Exercise 4: Debugging a C program that requires user input

1. Purpose / Introduction / Overview

The main purpose of this exercise is to use the debugger but not for debugging purposes, more to learn what may happen if the code asks for users input. Ways to input within terminal or reference an outside .txt file in order to fulfill the interactive design of the code. This code had no problems within the lines of the code, strictly to see what will happen to gdb.

2. Approach / Method

My beginning approach, truthfully, was to first understand what the point of the exercise was. So to begin I started out running the code through gdb in order to see what was wrong. After running it and stepping through some lines of code the debugger fails and stops working. This resulted when the code was asking the name “the interaction part of the code”. While trying to figure out the meaning of using a text file to pass through that part of the code I was able to find out a command to use to help. After making a text file with just my name and saving it as a (.txt) file then using this command (/n “input.txt”) to answer the interactive part.

3. Conclusion / Summary

How I ran CGDB to completion on the executable created by compiling interactive\_hello.c without getting stuck.

-The steps I took in order to enable the code to be executed throughout the gdb function was to first get into gdb from terminal. After setting it gdb interactive\_hello I then used the command run and step, till I reached the point where it asks me for information. Then I used the command /n input to input my response to the code. After that is done we are now able to step through the function normally till we reach the end.

Code not needed since no changes were made- but attached anyway

Exercise 5: Pointers and Structures in C

1. Purpose / Introduction / Overview

The main purpose of this exercise is to be able to complete a function and have it implement a specified algorithm, which one is given to us, and having our code check if it is a singly-linked list and has a cycle

2. Approach / Method

Algorithm- 1. Start, two pointers at the head of the list, the first one tortoise and the second one hare. 2. Advance hare by two nodes. If in the case of null pointer, that’s the end of the list, meaning the list is acyclic. 3. Advance tortoise by one node. (A null pointer check is unnecessary.) 4. If tortoise and hare point to the same node, the list is cyclic. Otherwise, acyclic, go back to step 2 to fix.

Code Implemented- attached in folder as ll\_cycle.c

3. Conclusion / Summary

Explanation- The code I included was my adaption to how I can get the function to work. With no prior experience with C I tried my best to implement the algorithm above to work within the code. It was able to give me a response when I plugged this code into terminal to see the outcome. However, I do realize my adaption to the code may not be the best resulting.