CSE31: Lab #2 – GDB & Pointers

Overview

By now, you have "mastered" the skills of coding and compiling your programs in a terminal. The next step is to further explore the tools we can use in terminals to enhance our "programming experience". The goal of this lab is to learn how to use GDB (LLDB in you are using a MAC computer) to debug and fix errors in your programs. We will practice the use of pointers and how they are related to memory. You can refer to chapter 5 of K&R for references on pointers.

Getting started

Before we begin any activities, create a directory (Lab_2) inside the CSE31 directory we created last week. You will save all your works from this lab here.

You must show your answers to the lab activities before leaving lab in order to receive participation score.

Tutorial in GDB.

GDB (**G**NU **Deb**ugger) is a debugger for C and C++ (and many other languages) that runs on Linux system.

TPS (Think-Pair-Share) activity 1 Paired with the classmate sitting next to you and do the following tasks (you are allowed to have groups of 3 students):

1. **Record your partner's name.**
2. Independently search from the internet 3 online tutorials on how to **setup** and **use** GDB in your system.
3. Share with your partner about what you have found.
4. Bookmark your results in the browser of your computer.
5. Your TA will "invite" one of you randomly to share what you have discussed.

Setting up GDB

Follow the tutorials you have found, setup GDB in your computer.
If you are using **Windows**, the setup is the same as Linux setup (since you are in terminal environment)
If you are using **Mac OS**, it may take more time for you. You will need to install ***Homebrew*** and use the ***brew*** command to install GDB.

Running GDB

Copy your ***punishment.c*** from Lab #1 into your Lab_2 directory.

TPS activity 2 Paired with the same classmate sitting next to you and discuss the following questions based on what you learn from the GDB tutorials (15 minutes):

1. How do you compile your ***punishment.c*** so that you can debug it using GDB? Try it with your code and make the name of the executable ***punish***.
2. Once ***punishment.c*** is compiled, how do you load it in GDB? Try it with your program.
3. Once ***punish*** is loaded, how do you run it in GDB? Try to run your ***punish***.
4. What are breakpoints? How do you set a breakpoint at a certain line of your program? Try to set a breakpoint in ***punishment.c*** where the ***for loop*** begins.
5. Now run the program again. It should stop at the breakpoint you set in Q4. From here, how do you run the program line by line? Try to run the next 3 lines with your program.
6. While you are still running ***punish*** line by line, how can you see the value of a variable? Pick 3 variables in your program and display them in the terminal one by one.
7. Now you are tired of running line by line. How do you let the program finish its run? Try to finish running your ***punish***.
8. How do you exit from GDB?
9. Your TA will “invite” one of you randomly to share what you have discussed.

(Exercise) Create – pointers.c

Use your favorite text editor to create a C program called ***pointers.c*** and copy the following code to your file:

```
int main(){
   int x, y, *px, *py;
   int arr[10];
   return 0;
}
```

TPS activity 3 Paired with the same classmate sitting next to you and discuss questions (15 minutes):

1. How many variables were declared in line 1? How many of them are pointers (and what are they)?
2. What will be the values of ***x***, ***y***, and ***arr[0]*** if you run the program? Validate your answer by running the program. Why do you think it happens that way? You will need to insert print statements to display those values.
3. How do you prevent ***x***, ***y***, and the content of ***arr*** from having unexpected values? Try to fix them in the program.
4. The moment you have declared a variable, the program will allocate a memory location for it. Each memory location has an *address*. Now insert print statements to display the ***addresses*** of ***x***, ***y***.
5. Now insert code so that ***px*** points to ***x*** and ***py*** points to ***y***. Print out the values and addresses of those pointers using only the pointer variables (yes, pointers have addresses too!) You should see that value of ***px*** equals to address of ***x***, and the same is true with ***py*** and ***y***.
6. As we've learned in lectures, an array name can be used as a pointer to access the content of the array. Write a loop to print out the content of ***arr*** by using ***arr*** as a pointer (**do not use []**).
7. Are array names really the same as pointers? Let's find out! An array name points to the first element of an array, so ***arr*** should point to the address of ***arr[0]***. Insert code to verify this.
8. Now print out the address of ***arr***. Does the result make sense? Why?
9. Your TA will "invite" one of you randomly to share what you have discussed.

(Assignment 1, individual) Segmentation Faults

Recall what causes segmentation fault and bus errors from lecture. Common cause is an invalid pointer or address that is being dereferenced by the C program. Use the program ***average.c*** from the assignment page for this exercise. The program is intended to find the average of all the numbers inputted by the user. Currently, it has a bus error if you input more than one number.

Load ***average.c*** into GDB with all the appropriate information and run it. GDB will trap on the segmentation fault and give you back the prompt. First find where the program execution ended by using **backtrace** (***bt*** as shortcut) which will print out a stack trace. Find the exact line that caused the segmentation fault.

Answer the following questions:

1. What line caused the segmentation fault?
2. How do you fix the line so it works properly?

You can recompile the code and run the program again. The program now reads all the input values but the average calculated is still incorrect. Use GDB to fix the program by looking at the output of ***read_values***. To do this, either set a **breakpoint** using the line number or set a breakpoint in the ***read_values*** function. Then continue executing to the end of the function and view the values being returned.

Answer the following questions:

3. What is the bug here?
4. How do you fix it?

(Assignment 2, individual) Fix appendTest.c

Compile ***appendTest.c*** from the assignment page and answer the following questions while running the program:

1. Run the program with the following input: ***"HELLO!"*** for **str1** and ***"hello!"*** for **str2**. Is the output expected?
2. Do not stop the program, enter ***"HI!"*** for **str1** and ***"hi!"*** for **str2**. Is the output expected? What is the bug here? Try to fix the program so it will print the output correctly.
3. Do not stop the program, enter ***"Hello! How are you?"*** for **str1** and ***"I am fine, thank you!"*** for **str2**. Is the output expected? Why do you think this happens? **You don't need to fix this**.

You can now stop the program by pressing **ctrl-c**.

(Assignment 3, individual) Complete arrCopy.c

Study and complete ***arrCopy.c*** so it prints out the following sample result in the same format. You must only insert code in the segments where "//Your code here" is labeled. **Contents of all arrays must be accessed through pointers, so you must not use any array notation "[ ]" in the code. Hint: Use dynamic memory allocations!**

Sample result (input is in italic and bold):

```
Enter size of array:
5
Enter array content #1: 1
Enter array content #2: 3
Enter array content #3: 5
Enter array content #4: 7
Enter array content #5: 9
printArr: 1 3 5 7 9
printArr: 1 3 5 7 9
```

What to submit

When you are done with this lab assignments, you are ready to submit your work. Make sure you have included the following ***before*** you press Submit:

- Your pointers.c, average.c , appendTest.c, arrCopy.c, and answers to the TPS activities/assignments in a text file, and a list of Collaborators.
- Your assignment is closed **7 days after this lab is posted** (at 11:59pm).
- You must demo your submission to your TA **within 14 days** (preferably during next lab so you will have a chance to make correction and re-submit/re-demo.)