Lab 02: GDB and Pointers
CSE-031-01 **Points: 20**

Overview

By now, you should 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 the terminal to enhance our "programming experience". The goal of this lab is to learn how to use GDB (or, optionally, LLDB in case you are using a MAC computer) to debug and fix errors in your programs. Note that if you are using LLDB, it is your responsibility to learn the 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. **Note that all the files shown in green below are the ones you will be submitting for this assignment.**

You must have a clear idea of how to answer the lab activities before leaving lab to receive participation score.

Tutorial in GDB and Setup

GDB (or **G**NU **D**e**b**ugger) is a debugger for C and C++ (and many other languages) that runs on Linux systems.

TPS (Think-Pair-Share) activity 1: Perform the following tasks while paired with your classmates assigned by your TA (you will be assigned to groups of 3-4 students):

1. Record your TPS partners' names.
2. Independently search the internet for 3 online tutorials on how to **setup** and **use** GDB (or LLDB) in your system.
3. Share your tutorials with your TPS partners.
4. Bookmark your results in the browser of your computer.

Your TA will "invite" one of you randomly after the activity to share what you have discussed.

Setting up GDB: Follow the tutorials you have found (or the tutorials that have been made available through CatCourses) to setup GDB in your computer.

Running GDB

Copy your `punishment.c` file from Lab 1 into your `Lab_2` directory.

TPS activity 2: Discuss questions 1 – 8 with your TPS partners in your assigned group (15 minutes) and record your answers in a text file named `tpsAnswers.txt` under a section labelled "TPS 2" (***you will continue to use this file to record your answers to all the TPS questions that follow in the lab handout***):

1. How do you compile your `punishment.c` so that you can debug it using GDB? Try it with your code and set the name of the executable to `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 that 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?

Your TA will "invite" one of you randomly after the activity to share what you have discussed.

Create `pointers.c`

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

```
#include <stdio.h>

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

   return 0;
}
```

TPS activity 3: Discuss questions 1 – 8 with your TPS partners in your assigned group (15 minutes) and record your answers in `tpsAnswers.txt` under a section labelled "TPS 3":

1. How many variables were declared in the first line of `main()`? 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 `printf` 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 `printf` statements to display the addresses of `x` and `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 the value of `px` is equal to the address of `x`, and the same is true for `py` and `y`.
6. As we have learned in lectures, an array name can be used as a pointer to access the content of the array. Write a loop to output the contents of `arr` by using `arr` as a pointer (**do not** use `[]` in your loop).
7. Are array names really the same as pointers? Let us 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?

Your TA will "invite" one of you randomly after the activity to share what you have discussed.

Individual Assignment 1: Segmentation Faults

Recall what causes segmentation fault and bus errors from lecture and the textbooks. 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 in a text file named `assignAnswers.txt` (***you will continue to use this file to record your answers to all the assignment questions that follow in the lab handout***) under a section labelled "Assignment 1":

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 till the end of the function and view the values being returned. Answer the following questions (in the text file `assignAnswers.txt` as mentioned above):

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

Individual Assignment 2: Fix `appendTest.c`

Compile `appendTest.c` from the assignment page and record your answers to the following questions in `assignAnswers.txt` **while running the program** under a section labelled "Assignment 2":

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 do not need to fix this**.

You can now stop the program by pressing **Ctrl-C**.

Individual Assignment 3: Complete `arrCopy.c`

Study and complete `arrCopy.c` so that it outputs the following sample result in the same format. You must only insert code in the segments labelled with `//Your code here`. Contents of all arrays **must be accessed through pointers**, so you must not use any array notation (`[]`) in your code.

Hint: Use dynamic memory allocations (`malloc`)!

Your program must produce an output that **exactly resembles the Sample Run, including identical wording of prompts, spacing, input locations, etc**.

Sample Run (user input shown in blue, with each run separated by a dashed line):

```
--------------------------------------------------------------------------------SAMPLE RUN 1
Enter the size of array you wish to create: 5
Enter array element #1: 1
Enter array element #2: 3
Enter array element #3: 5
Enter array element #4: 7
Enter array element #5: 9

Original array's contents: 1 3 5 7 9
Copied array's contents: 1 3 5 7 9
```

Collaboration

You must credit anyone you worked with in any of the following three different ways:

1. Given help to

2. Gotten help from
3. Collaborated with and worked together

What to hand in

When you are done with this lab assignment, submit all your work through CatCourses.

Before you submit, make sure you have done the following:

- Your code compiles and runs on a Linux machine (without the need for special libraries).
- Attached `pointers.c`, `average.c`, `appendTest.c`, `arrCopy.c`, `assignAnswers.txt`, and `tpsAnswers.txt`.
- Filled in your collaborator's name (if any) in the "Comments…" text-box at the submission page.

Also, remember to demonstrate your code to the TA or instructor before the end of the grace period.