CS240 Lab 6 Assignment
Gnu Debugger (gdb) Practice

You should use your Linux environment (**VSCode**) for this assignment, and enter the commands given in the Terminal. This exercise is similar to part 8 from the practice problems (but even if you did that problem, please repeat here for the lab assignment).

The intention of the assignment is to get introduced to and start using the **gdb** debugger. But there are a few questions that you should answer and submit as a hardcopy (the questions are in boxes within the exercises). It's okay to just submit the answers only (and not print the whole exercise).

If you did the practice problems for the Pointers assignment, you should already have the ***cmemory*** repository.

NOTE: the **$** in the commands listed represents the command-line prompt in the Terminal. So, don't type the **$** when entering the commands.

1. If you do **not** already have the repository, get it:

```
$ cs240 start cmemory

$ cd cmemory
```

The file *strings3.c* from the *cmemory* repository for the pointers assignment contains the following code:

Examine the code carefully, and read the comments given (highlighted in yellow):

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char** argv) {
// dynamically allocate space for 3 pointers.  commandA will be the address in memory where the
first pointer is stored.

//Each pointer will be the starting address of a string of characters.

//We will refer to commandA as an array of pointers to strings.

  char** commandA = (char**)malloc( 3 * sizeof(char*) );

// Initialize the strings and mark the end of the commandA array by assigning a NULL to the last
pointer in the array

  commandA[0] = "emacs";
  commandA[1] = "strings.c";
  commandA[2] = NULL;


// deallocate the memory used for the commandA array
  free(commandA);

//allocate a new array of 3 pointers called commandB
  char** commandB = (char**)malloc( 3 * sizeof(char*) );

  //initialize the strings for commandB
  commandB[0] = "ls";
  commandB[1] = "cs240-pointers";
  commandB[2] = NULL;

 //change the second string of the commandA array
  commandA[1] = "uh oh";

 //print the strings of each array
  printf("A: %s %s\n", commandA[0], commandA[1]);
  printf("B: %s %s\n", commandB[0], commandB[1]);


 // deallocate the commandB array
 free(commandB);

  return 0;
}
```

2. Compile the ***strings3.c*** program to produce an executable file ***strings3*** with the following command:

```
$    gcc -Wall --std=c99 -g -O -o strings3 strings3.c
```

⇒ **gcc** is the Gnu C Compiler
⇒ **-Wall --std=c99 -g -O** are options which tell the compiler what to do.
⇒ The **–g** option creates debugging information for use in the Gnu debugger, **gdb**

3. Start the debugger using your executable file ***strings3***

```
$ gdb    ./strings3

        GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-120.el7
        Copyright (C) 2013 Free Software Foundation, Inc.
        License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
        This is free software: you are free to change and redistribute it.
        There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
        and "show warranty" for details.
        This GDB was configured as "x86_64-redhat-linux-gnu".
        For bug reporting instructions, please see:
        <http://www.gnu.org/software/gdb/bugs/>...
        Reading symbols from /home/jherbst/cs2402021/cmemory/strings3...done.
 (gdb)
```

You will now enter **gdb** commands at the (gdb) prompt.

4. Set a breakpoint at the beginning of the program (C programs always start execution at the ***main*** function).

```
(gdb) break main

        Breakpoint 1 at 0x      : file strings3.c, line 7.
```

What address in memory does ***main*** begin (record the address from your own output. where you see the yellow highlight above):

5. Run the program:

```
(gdb) run
        Starting program: /home/jherbst/cs2402021/cmemory/./strings3

        Breakpoint 1, main (argc=1, argv=0x7fffffffe148) at strings3.c:7
        7          int main(int argc, char** argv) {
        Missing separate debuginfos, use: debuginfo-install glibc-2.17-
323.el7_9.x86_64
```

NOTE: You can ignore the message about Missing separate debuginfos...

The program has hit the breakpoint at the beginning of ***main*** function and paused execution of the program.

6. Execute a step (a single instruction) of the program:

```
(gdb) step

          char** commandA = (char**)malloc( 3 * sizeof(char*) );
```

After a step, **gdb** displays the *next* instruction to be executed, highlighted above in yellow, which will allocate space for an array of 3 pointers to chars. Basically, the array will contain the addresses in memory of some strings.

7. Take another step to perform the allocation, and then examine (**x**) the value pointed to by **commandA:**

```
(gdb) step


(gdb) x   commandA

          0x      :  0x00000000
```

What *address* in memory does **commandA** refer to (record the value you see on your computer where the yellow highlight is shown directly above)?

The 4 bytes in memory starting at **commandA** are shown, and are highlighted in green above,

The 4 bytes in memory at address **commandA** seem to be zeroes at this point, but not because our program has written those values. Memory can often contain values from initialization or previous use.

8. Take three more steps to execute the statements that initialize the **commandA** array:

```
          commandA[0] = "emacs";

(gdb) step

          commandA[1] = "strings.c";


(gdb) step


          commandA[2] = NULL;


(gdb) step
```

9. Display the contents of the **commandA** array (**/3a** means display **3** values stored in memory, where the values are assumed to represent pointers):

```
(gdb) x   /3a     commandA

          0x602010:       0x4006f0        0x4006f6
          0x602020:       0x0
```

gdb displays contents of memory from lowest to highest addresses, with a maximum of 16 bytes per line.

The address is shown in the left column, and the values stored starting at that address, in the right columns.

You could interpret this as a memory diagram which displays 4 bytes per row as:

Address	Data in Memory
0x602020:	0x00 0x00 0x00 0x00
0x602018:	0x00 0x40 0x06 0xf6
commandA=0x602010:	0x00 0x40 0x06 0xf0

Or, as an array:

commandA = [0x4006f0, 0x4006f6, 0x0]

Why does the last pointer in the array have a value of 0x0?

10. Use **gdb** to display the strings:

We are asking to e**x**amine the value that **commandA** is pointing to, displayed as a **s**tring:

```
(gdb) x   /s     *commandA

          0x4006f0:       "emacs"
```

Pointer arithmetic is used when you specify **commandA+1**, so we are asking for the next string:

```
(gdb) x   /s     *(commandA + 1)

          0x4006f6:       "strings.c"
```

11. Take a step to deallocate **commandA:**

```
free(commandA)
```

(gdb) **step**

12. Take another step to allocate space for another array of 3 pointers, and then examine the value of **commandB:**

```
char** commandB = (char**)malloc( 3 * sizeof(char*) );
```

(gdb) **step**

(gdb) **x commandB**

```
0x602010:        0x0
```

13. Also, examine **commandA** again:

(gdb) **x commandA**

```
0x602010:        0x0
```

What do you notice about the two values? . Why do you think this happened?

14. Take three more steps to execute the statements that initialize the **commandB** array:

```
commandB[0] = "ls";
```

(gdb) **step**

```
commandB[1] = "cs240-pointers";
```

(gdb) **step**

```
commandB[2] = NULL;
```

(gdb) **step**

15. Display the contents of the **commandB** array:

(gdb) **x /3a commandB**

```
0x602010:       0x400700        0x400703
0x602020:       0x0
```

Notice that these are different values than when commandA was initialized.

16. Now execute the next instruction:

```
commandA[1] = "uh oh";
```

(gdb) **step**

17. And again display the contents of the **command** array:

(gdb) **x /3a commandB**

Has **commandB** changed? Did you expect it to?

18. Complete execution of the program:

```
printf("A: %s %s\n", commandA[0], commandA[1])
```

(gdb) **step**

```
A: ls uh oh
```

```
printf("B: %s %s\n", commandB[0], commandB[1]);
```

(gdb) **step**

```
B: ls cs240-pointers
```

Explain what is surprising about the output.

How could you modify the program to prevent this incorrect output?

19. Quit out of **gdb:**

(gdb) **quit**