CS 161 Lab #7 – C-Style Strings, Dynamic Arrays, and gdb

- Start by getting checked off for (up to) 3 points of work from Lab 6 in the first 10 minutes.
- Your goal is to get all points for Lab 7 checked off by a TA by the end of this lab, so that you do not need to do additional work outside of the lab.
- You will work in small groups to complete the programming assignments, with each person writing their own program. Include each partner's name in your file header.

Goals:

- Practice working with C-style strings
- Practice dynamic allocation of arrays and understanding the computer memory model
- Practice using the gdb debugger

(4 pts) A. C-style strings

For the rest of this lab, each time "string" is used, it means "C-style string" (which is an array of characters that ends with a NULL terminator).

1. Design a program to read in a string (max 20 characters) from the user and report the percentage of characters in the string that are the character '`*`' (let's call this the "star-factor"). The program should print out the result.

Example output (user input is highlighted):

```
Please enter a string (<=20 chars): ***Hydra*** *Virgo*
String length: 19
Stars: 8
Star-factor (percentage): 42.1052%
```

2. What are useful test cases? Before writing the program, create 3 test cases (with expected output). Put your test cases in comments in the program.

3. Implement your program **(in lab7.cpp)**.

- Use a **static C-style string (character array**) to store the string, of size 21 (why?).
- Be sure to initialize your C-style string before it is used.
- Use `cin.getline()` to read in the user's string.
- Use `strlen()` to get the length of the string. Remember to `#include <cstring>`
- Watch out for integer division! Convert arguments to float, and multiply by 100.0 to get a percentage.
- Run your test cases and check that they work correctly.
- **Extra time? Update your program to use a function to compute the star factor:**
  `float star_factor(char str[]);`

(3 pts) B. Allocate Dynamic Arrays

1. Add a section to your program that generates **random** grades for a user's classes (evil!). Ask the user how many classes they are taking and generate a random grade (integer is fine) in a range that you choose. The program should print out the grades it generated for the user.

Example output, with random grades ranging from 70 to 100 (user input is highlighted):

```
How many classes are you taking? 5
Your grade in class 0 is 75
Your grade in class 1 is 95
Your grade in class 2 is 93
Your grade in class 3 is 72
Your grade in class 4 is 91
```

2. Implement your program **(in lab7.cpp)**.

- The program should dynamically allocate an array to hold the grades (using `new`), with size corresponding to the user's number of classes.
- Using an integer for each grade is fine. However, if you want to generate floating point numbers (and store them in a `float` array), you can generate a random number of tenths (or hundredths, etc.) and add it to the lowest value you want to allow. For example, to get random numbers (in tenths) between 2.3 and 4.0 (inclusive):
  `float(rand()%18)/10.0 + 2.3  /* this evaluates to a float */`
- Remember to free the array when you are done, with `delete []`
- **Extra time? Update your program to use a function to create, populate, and return the grades array, using one of these options:**
  ```
  float* generate_grades(int n_classes);
  int* generate_grades(int n_classes);
  ```

3. Use **valgrind** to check your program for memory leaks.

```
$ valgrind lab7
```

Did you find any? If so, fix them now and re-run valgrind until you get no leaks.

(3 pts) C. Using the gdb debugger

The purpose of a debugger is to allow you to see what is going on "inside" a program while it executes -- or what the program was doing at the moment it crashed.

We will explore using the GNU Project Debugger (GDB) as a tool for interactive debugging.

GDB gives you fine control over the execution of your program to help find bugs in action:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine the computer state (memory, variables, pointers) when your program has stopped.
- Change details of your program so you can experiment with correcting the effects of one bug and go on to learn about the next one.

The GDB man page is a good source of information. You can access it with

```
$ man gdb
```

The first step is to compile your program with debugging symbols preserved, using the -g flag:

```
$ g++ filename.cpp -g -o executable_filename
```

Let's start with a simple program that gets a line of text from the user, and prints it out backwards to the screen. Type the following program into a file called **`debug.cpp`** . Line numbers are shown for your reference.

```
1.   #include <iostream>
2.   #include <cstring>
3.   using namespace std;
4.
5.   int main() {
6.      char input[50];
7.      int i = 42;
8.      cin.getline(input, 50);
9.      for (i = strlen(input)-1; i >= 0; i--) {
10.         cout << input[i];
11.     }
12.     cout << endl;
13.     return 0;
14.  }
```

Compile the program and start the debugger with:

```
$ g++ debug.cpp -g -o debug
$ gdb ./debug
```

Complete the following mini-tutorial to learn the 8 main commands that you'll need for debugging:

1. `break (or 'b')`
2. `run (or 'r')`
3. `print (or 'p')`
4. `next (or 'n') and step (or 's')`
5. `continue (or 'c')`
6. `display and watch`
7. `where (or 'backtrace' or 'bt')`

1. The `break` Command:

Before we begin, note that you can get information about any `gdb` command by typing
`(gdb) help [command]`
at the `gdb` prompt.

`gdb` has access to the line numbers of your source file. This lets us set *breakpoints* for the program. A **breakpoint** is a line in the code where you want execution to pause. Once you pause execution you will be able to examine variables, walk through the program, and make changes.

Set up a break point at line 7, just before we declare `int i = 42;`

```
(gdb) break 7
Breakpoint 1 at 0x4008b5: file debug.cpp, line 7.
```

`0x4008b5` is the location of that line of code in memory (on the stack). The number you see may be different.

2. **The `run` Command:**

`run` begins initial execution of your program. This will run your program as you normally would outside of the debugger, until it reaches a breakpoint line. At that point, you are returned to the `gdb` command prompt. (Using `run` again after your program has been started will cause `gdb` to ask whether you want to start over from the beginning.)

From our example:

```
(gdb) run
Starting program: ./debug
Breakpoint 1, main () at debug.cpp:7
7           int i = 42;
```

3. **The `print` Command:**

`print` will let you inspect the values of variables in your program. It takes an argument of the variable name.

In our example, the program is paused right before we declare and initialize `i`. Let's see what the value of `i` is now:

```
(gdb) print i
$1 = 0
```

In this case, `i` contains 0 prior to initialization. `print` can be shortened to `p`.

4. **The `next` and `step` Commands:**

`next` and `step` provide two ways to step line by line through the program. The difference is that `next` steps over a function call, while `step` will step into it.

Step to the beginning of the next statement:

```
(gdb) step
8       cin.getline(input, 50);
```

Before we execute the `cin.getline()`, let's check the value of `i` again:

```
(gdb) p i
$2 = 42
```

`i` is now equal to 42, because the assignment statement executed.

Now let's use `next` to move on to the `cin.getline()` statement:

```
(gdb) next
```

What is happening? The program is waiting for you to type something. Type in "hello" (without the quotes) and press enter.

```
hello
9         for (i = strlen(input)-1; i >= 0; i--) {
```

5. The `continue` Command

`continue` will resume execution of the program after it reached a breakpoint.

Let's continue to the end of the program now:

```
(gdb) continue
Continuing.
olleh
[Inferior 1 (process 23414) exited normally]
```

We've reached the end of the program, which printed "olleh", the reverse of what we typed in.

6. The `display` and `watch` Commands:

`display` shows a variable's contents at each step through the program. First let's look at our breakpoints:

```
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x00000000004008b5 in main() at debug.cpp:7
        breakpoint already hit 1 time
```

Delete breakpoint 1 (which was at line 7):

```
(gdb) del break 1
(gdb) info breakpoints
No breakpoints or watchpoints.
```

You can also delete all breakpoints using `del` with no arguments (gdb will ask you to confirm).

Now set a new breakpoint at line 10, the `cout` statement inside the `for` loop:

```
(gdb) break 10
Breakpoint 2 at 0x4008e3: file debug.cpp, line 10.
```

Run the program again and **enter an input string (hello)**. When it returns to the `gdb` command prompt, we will use `display input[i]` to watch how that expression changes as the `for` loop executes. While `print` shows a variable once, `display` shows the variable each time the program stops (e.g., each `next` or breakpoint reached).

```
Breakpoint 2, main () at debug.cpp:10
10              cout << input[i];
(gdb) display input[i]
1: input[i] = 111 'o'
(gdb) p i
$3 = 4
(gdb) continue
Continuing.
Breakpoint 2, main () at debug.cpp:10
10              cout << input[i];
```

```
1: input[i] = 108 'l'
(gdb) p i
$4 = 3
(gdb) c
Continuing.
Breakpoint 2, main () at debug.cpp:10
10            cout << input[i];
1: input[i] = 108 'l'
(gdb) p i
$5 = 2
(gdb) c
Continuing.
Breakpoint 2, main () at debug.cpp:10
10            cout << input[i];
1: input[i] = 101 'e'
```

Here we stepped through three iterations of the loop. Each step displayed `input[i]` automatically. We printed out `i` explicitly as well.

We can also watch a variable, which stops execution and displays the contents **at any point when the value changes**. For example, we can watch the `i` variable change in the for loop:

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: ./debug
hello
Breakpoint 2, main () at debug.cpp:10
10            cout << input[i];
1: input[i] = 111 'o'
(gdb) watch i
Hardware watchpoint 2: i
(gdb) c
Continuing.
Hardware watchpoint 2: i

Old value = 4
New value = 3
0x0000000000400900 in main () at debug.cpp:9
9       for (i = strlen(input)-1; i >= 0; i--) {
1: input[i] = 108 'l'
(gdb) c
Continuing.
```

```
Breakpoint 2, main () at debug.cpp:10
10        cout << input[i];
1: input[i] = 108 'l'
(gdb) c
Hardware watchpoint 2: i

Old value = 3
New value = 2
0x0000000000400900 in main () at debug.cpp:9
9       for (i = strlen(input)-1; i >= 0; i--) {
1: input[i] = 108 'l'
```

7. **The `where (or backtrace)` Command:**

The `where` (or `backtrace`) command prints a *backtrace* of all *stack frames*. A new **stack frame** is created each time a function is called. A **backtrace** shows the list of stack frames that were created to reach the current program point. This can be very helpful for figuring out what caused a program to crash.

Quit your current `gdb` session (`quit`) and use `vim` to modify the program just a little (line 6) so that it will crash:

```
1.   #include <iostream>
2.   #include <cstring>
3.   using namespace std;
4.
5.   int main() {
6.       char* input = NULL;
7.       int i = 42;
8.       cin.getline(input, 50);
9.       for (i = strlen(input)-1; i >= 0; i--) {
10.          cout << input[i];
11.      }
12.      cout << endl;
13.      return 0;
14.  }
```

We changed `input` to be a pointer to a char and set it to NULL to make sure it doesn't point anywhere until we set it. Recompile the program and run `gdb` to see what happens when it crashes.

```
(gdb) run
Starting program: ./debug
hello
```

```
Program received signal SIGSEGV, Segmentation fault.
std::istream::getline (this=0x601080 <_ZSt3cin@@GLIBCXX_3.4>, __s=0x0,
__n=50, __delim=<optimized out>)
    at ../../../../../libstdc++-v3/src/c++98/istream.cc:75
75                          *__s++ = traits_type::to_char_type(__c);
(gdb) where
#0  std::istream::getline (this=0x601080 <_ZSt3cin@@GLIBCXX_3.4>,
__s=0x0, __n=50,
    __delim=<optimized out>) at ../../../../../libstdc++-
v3/src/c++98/istream.cc:75
#1  0x00000000004008da in main () at debug.cpp:8
```

The program crashed when `getline()` tried to put data into the NULL pointer. This is in stack frame #0. We can move "up" to frame #1 to inspect what was happening in `main()` when `getline()` was called.

```
(gdb) up
#1  0x00000000004008da in main () at debug.cpp:8
9         cin.getline(input, 50);
```

This indicates that line #8 is where the program crashed. We can then inspect the `input` variable and figure out why there was a crash.

```
(gdb) p input
$1 = 0x0
```

A NULL pointer! The perfect clue to encourage us to check whether we allocated memory for the string (or forgot).

There are many more useful gdb commands, such as setting a breakpoint whenever a certain function is called (break my_function_name). Here is a useful quick reference:

http://users.ece.utexas.edu/~adnan/gdb-refcard.pdf

Remember, you can type

```
(gdb) help [command]
```

anytime to get immediate information about a command.

To get the points for this section, paste the output of last 4 steps into a document, save it as a PDF, and show it to your TA (then upload as below).

E. Get your work checked off by a TA.

Submit your program on Canvas (can be done while you are waiting to be checked off). All partners' names must be in the comment header to get credit.

1. Transfer your .cpp file from the ENGR servers to your local laptop.

2. Go to the Lab 7 assignment: https://canvas.oregonstate.edu/courses/1770357/assignments/7847929
3. Click "Submit Assignment".
4. Upload your .cpp file (lab7.cpp) and .pdf file (lab7.pdf) from Part C.
5. Click the **"Submit Assignment"** button.
6. You are done!

If you finish the lab early, this is a golden chance to work on your Assignment 4 (with TAs nearby to answer questions!).

Remember, the assignment you submit must be your work alone (no partners).
