CS 161 Lab #8 – Structs and Debugging with GDB

- Get checked off for **(up to) 3 points** of work from the previous lab in the first 10 minutes, **if you had a non-zero grade for Lab 7 already**.
- **To get credit for this lab, you must be checked off by a TA by the end of lab.**
- **This lab should be done solo, not via pair programming.**

Goals:

- Practice designing and using C++ structs
- Gain experience using a debugger (GDB)

(3 pts) A. Lab Quiz (Canvas)

Visit this link on Canvas to take the Lab 8 quiz:
https://oregonstate.instructure.com/courses/1771939/quizzes/2535696

Re-take the quiz until you get all of the questions right! Canvas saves your last score, not your highest score. If you don't get 100% within the time available, finish outside of lab.

(3 pts) B. Working with Structs

Let's create a new data type called `umbrella` with two member variables:

```
struct umbrella {
  float length;
  string color;
};
```

Write a program called `lab8_umbrellas.cpp` that includes this data type (struct) definition before the `main()` method.

1. Inside `main()`, declare a **static** array that contains three umbrellas.
2. Write a `for` loop to read in values from the user for the `length` and `color` members of each umbrella. You do not need to check for valid user input.
3. Write a `for` loop to find the **index** of the longest umbrella.
4. Output the **index, length, and color** of the longest umbrella.

Example output (user input is highlighted):

```
Enter length for umbrella 0: 5.3
Enter color for umbrella 0: red
Enter length for umbrella 1: 7.9
Enter color for umbrella 1: green
Enter length for umbrella 2: 4.1
Enter color for umbrella 2: black
The longest umbrella (index 1) has a length of 7.9 and is green.
```

(4 pts) C. Using a Debugger (GDB)

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 this program into a file called **debug.cpp** :

```
#include <iostream>
#include <cstring>

using namespace std;

int main() {
   char input[50];
   int i = 42;
   cin.getline(input, 50);
   for (i = strlen(input); i >= 0; i--) {
      cout << input[i];
   }
   cout << endl;
   return 0;
}
```

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`
2. `run`
3. `print`
4. `next & step`
5. `continue`
6. `display & watch`
7. `where (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 8, just before we declare `int i = 42;`

```
(gdb) break 8
Breakpoint 1 at 0x4008b5: file debug.cpp, line 8.
(gdb)
```

`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:

```
Starting program: ./debug

Breakpoint 1, main () at debug.cpp:8
8           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.

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
9       cin.getline(input, 50);
```

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

```
(gdb) print 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
10        for (i = strlen(input); 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:8
        breakpoint already hit 1 time
```

Delete breakpoint 1 (which was at line 8):

```
(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 11, the `cout` statement inside the `for` loop:

```
(gdb) break 11
Breakpoint 2 at 0x4008e3: file debug.cpp, line 11.
```

Run the program again and enter an input string. 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:11
11            cout << input[i];
(gdb) display input[i]
1: input[i] = 0 '\000'
(gdb) print i
$3 = 5
(gdb) next
10      for (i = strlen(input); i >= 0; i--) {
1: input[i] = 0 '\000'
(gdb) next

Breakpoint 2, main () at debug.cpp:11
11            cout << input[i];
1: input[i] = 111 'o'
(gdb) print i
```

```
$4 = 4
(gdb) next
13      for (i = strlen(input); i >= 0; i--) {
1: input[i] = 111 'o'
```

Here we stepped through two 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 memory 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
(gdb) watch i
Hardware watchpoint 2: i
(gdb) continue
Continuing.
hello
Hardware watchpoint 2: i

Old value = 0
New value = 5
0x00000000004008e1 in main () at debug.cpp:10
10      for (i = strlen(input); i >= 0; i--) {
(gdb) continue
Continuing.
Hardware watchpoint 3: i

Old value = 5
New value = 4
0x0000000000400900 in main () at debug.cpp:10
10      for (i = strlen(input); i >= 0; i--) {
(gdb) continue
Continuing.
Hardware watchpoint 3: i

Old value = 4
New value = 3
0x0000000000400900 in main () at debug.cpp:10
10      for (i = strlen(input); i >= 0; i--) {
```

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 so that it will crash:

```
#include <iostream>
#include <cstring>

using namespace std;

int main() {
   char* input = NULL;
   int i = 42;
   cin.getline(input, 50);
   for (i = strlen(input); i >= 0; i--) {
      cout << input[i];
   }
   cout << endl;
   return 0;
}
```

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.
0x00007ffff7b499b0 in std::istream::getline(char*, long, char) () from
/lib64/libstdc++.so.6
(gdb) where
#0  0x00007ffff7b499b0 in std::istream::getline(char*, long, char) ()
from /lib64/libstdc++.so.6
#1  0x00000000004008ca in main () at debug.cpp:9
```

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  0x00000000004008ca in main () at debug.cpp:9
9         cin.getline(input, 50);
```

This indicates that line #9 is where the program crashed. We can then inspect the `input` variable and figure out why there was a crash.

```
(gdb) p input
$5 = 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.

Ask a TA to check your work (get points) and submit your programs to TEACH

1. Transfer your .cpp file from the ENGR servers to your local laptop.
2. Connect to TEACH here: https://teach.engr.oregonstate.edu/teach.php
3. In the menu on the right side, go to **Class Tools -> Submit Assignment.**
4. Select **CS 161 020 Lab_8** from the list of assignments and click "SUBMIT NOW".
5. Select your .cpp file (`lab8_umbrellas.cpp`).
6. Click the **Submit** button.
7. You are done!

Point totals: 3 pts (quiz) + 3 pts (structs) + 4 pts (gdb)

If you finish the lab early, this is a chance to work on your Assignment 5 design (with TAs nearby to answer questions!).
