Lab 8

Get checked off for full points of incomplete work from the previous lab within the first 10 minutes of the lab.

In this lab, you can form a group of 2-3 individuals. **You must be checked off together as a group at the end of the lab.** Although you perform tasks as a group, ensure that you understand the work and ask questions to TAs as needed.

(4 pts) Debugging using a debugger

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

GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

GDB Manpage is a good source of information, i.e. man gdb.

The first thing you need to do to start debugging your program is to compile it with debugging symbols, this is accomplished with the -g flag:

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

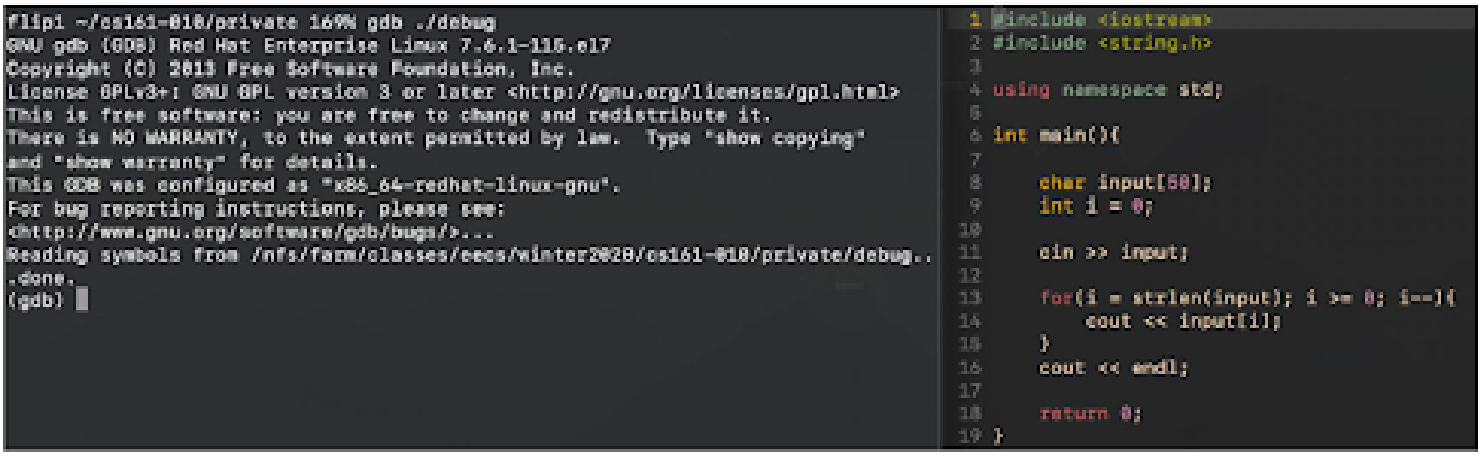
Let's start with a simple program that gets a line of text from the user, and prints it out backwards to the screen:

```
#include <iostream>
#include <string.h>

using namespace std;

int main(){
  char input[50];
  int i = 0;

  cin >> input;

  for(i = strlen(input); i >= 0; i--){
   cout << input[i];
  }
  cout << endl;

  return 0;
}
```

compile and start the debugger with:

```
g++ debug.cpp -g -o debug
gdb ./debug (start another session which will run gdb)
```

GDB Execution debug.cpp



Here is a mini tutorial for the 7 main commands that you will mostly be using in your debugging session

1. `break`
2. `run`
3. `print`
4. `next & step`
5. `continue`
6. `display & watch`
7. `where (or bt)`

1. **The `break` Command:**

`gdb` will remember the line numbers of your source file. This will let us easily set up break points in the program. A break point, is a line in the code where you want execution to pause. Once you pause execution you will be able to examine variables, and walk through the program, and other things of that nature.

Continuing with our example lets set up a break point at line 9, just before we declare `int i = 0;`

```
(gdb) break 9
Breakpoint 1 at 0x400923: file debug.cpp, line 9.
(gdb)
```

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

`run` will begin initial execution of your program. This will run your program as you normally would outside of the debugger, until it reaches a break point line. This means if you need to pass any command line arguments, you list them after `run` just as they would be listed after the program name on the command line.

At this moment, you will have been returned to the `gdb` command prompt. (Using `run` again after your program has been started, will ask to terminate the current execution and start over)

From our example:

```
Starting program: ./a.out

Breakpoint 1, main (argc=1, argv=0x7fffffffd008)
 at debug.cpp:9
9 int i = 0;
```

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

`print` will let you see the values of data in your program. It takes an argument of the variable name. In

our example, we are paused right before we declare and initialize i. Let's look what the value of i is now:

```
 (gdb) print i
$1 = -1075457232
 (gdb)
```

i contains garbage, we haven't put anything into it yet.

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

`next` and `step` do basically the same thing, step line by line through the program. The difference is that `next` steps over a function call, and `step` will step into it.

Now in our example, we will step to the beginning of the next instruction

```
 (gdb) step
11 cin >> input;
 (gdb)
```

before we execute the cin, let's check the value of i again:

```
 (gdb) print i
$2 = 0
 (gdb)
```

i is now equal to 0, like it should be.

Now, let's use next to move into the cin statement:

```
(gdb) next
```

What happened here? We weren't returned to the `gdb` prompt. Well the program is inside cin, waiting for us to input something.

Input string here, and press enter.

5. **The `continue` Command**

`continue` will pick up execution of the program after it has reached a break point.

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

```
(gdb) continue
Continuing.
olleh

[Inferior 1 (process 29333) exited normally]
(gdb)
```

Here we've reached the end of our program, you can see that it printed in reverse "input", which is what was fed to cin.

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

`display` will show a variable's contents at each step of the way in your program. Let's start over in our example. Delete the breakpoint at line 6

```
(gdb) del break 1
```

This deletes our first breakpoint at line 9. You can also clear all breakpoints w/ clear.

Now, let's set a new breakpoint at line 14, the cout statement inside the for loop

```
(gdb) break 14
Breakpoint 2 at 0x40094c: file debug.cpp, line 14.
(gdb)
```

Run the program again, and enter the input. When it returns to the `gdb` command prompt, we will display input[i] and watch it through the for loop with each next or breakpoint.

```
Breakpoint 2, main (argc=1, argv=0x7fffffffd008)
 at debug.cpp:14
14 cout << input[i];
(gdb) display input[i]
1: input[i] = 0 '\0'
(gdb) next
```

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

Breakpoint 2, main (argc=1, argv=0x7fffffffd008)
 at debug.cpp:14
14 cout << input[i];
1: input[i] = 111 'o'
(gdb) next
13 for(i=strlen(input);i>=0;i--) {
1: input[i] = 111 'o'
(gdb) next
```

Here we stepped through the loop, always looking at what input[i] was equal to.

We can also watch a variable, which allows us to see the contents at any point when the memory changes.

```
(gdb) watch input
Watchpoint 2: input
(gdb) continue
Continuing.
hello
Watchpoint 2: input

Old value =
"\030\320\377\377\377\177\000\000\065\n@\000\000\000\000\000\210\373\240\0
31 \065\000\000\000\360\t@", '\000' <repeats 13 times>,
"0\b@\000\000\000\000\000\000", <incomplete sequence
\320> New value =
"h\320\377\377\377\177\000\000\065\n@\000\000\000\000\000\210\373\240\031\
06 5\000\000\000\360\t@", '\000' <repeats 13 times>,
"0\b@\000\000\000\000\000\000", <incomplete sequence \320>
0x000000352067b82a in std::basic_istream<char, std::char_traits<char>
>&  std::operator>><char, std::char_traits<char>
>(std::basic_istream<char,  std::char_traits<char> >&, char*) () from
/usr/lib64/libstdc++.so.6

(gdb) continue
Continuing.
Watchpoint 2: input

Old value =
"h\320\377\377\377\177\000\000\065\n@\000\000\000\000\000\210\373\240\031\
06 5\000\000\000\360\t@", '\000' <repeats 13 times>,
"0\b@\000\000\000\000\000\000", <incomplete sequence
\320> New value =
"he\377\377\377\177\000\000\065\n@\000\000\000\000\000\210\373\240\031\065
```

```
\0 00\000\000\360\t@", '\000' <repeats 13 times>,
"0\b@\000\000\000\000\000\000", <incomplete sequence \320>
0x000000352067b82a in std::basic_istream<char, std::char_traits<char>
>&  std::operator>><char, std::char_traits<char>
>(std::basic_istream<char,  std::char_traits<char> >&, char*) () from
/usr/lib64/libstdc++.so.6
```

Type `q` and hit enter to exit from GDB.

7. The `where (or bt)` Command

The `where` (or bt) command prints a backtrace of all stack frames. This may not make much sense but it is useful in seeing where our program crashes.

Let's modify our program just a little so that it will crash:

```
#include <iostream>
#include <string.h>

using namespace std;

int main(){
       char *input = NULL;
       int i = 0;

       cin >> input;

       for(i = strlen(input); i >= 0; i--){
        cout << input[i];
       }
       cout << endl;

       return 0;
}
```

Here we've 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 and run `gdb` on it again to see what happens when it crashes.

```
(gdb) r
Starting program:
/nfs/farm/classes/eecs/fall2013/eecs161-001/private/a.out hello

Program received signal SIGSEGV, Segmentation fault.
0x000000352067b826 in std::basic_istream<char, std::char_traits<char>
>&  std::operator>><char, std::char_traits<char>
>(std::basic_istream<char,  std::char_traits<char> >&, char*) () from
/usr/lib64/libstdc++.so.6
```

```
(gdb) where
```

```
#0 0x000000352067b826 in std::basic_istream<char, std::char_traits<char> >&
std::operator>><char, std::char_traits<char> >(std::basic_istream<char,
std::char_traits<char> >&, char*) () from /usr/lib64/libstdc++.so.6 #1
0x0000000000400943 in main (argc=1,
 argv=0x7fffffffd008) at debug.cpp:11

(gdb)
```

We see at the bottom, two frames. #1 is the top most frame and shows where we crashed. Use the up command to move up the stack.

```
(gdb) up
#1 0x0000000000400943 in main (argc=1,
 argv=0x7fffffffd008) at debug.cpp:11
11 cin >> input;

(gdb)
```

Here we see line #11

```
11 cin >> input;
```

The line where we crashed.

Here are some more tutorials for gdb:
http://www.cs.cmu.edu/~gilpin/tutorial/
https://sourceware.org/gdb/current/onlinedocs/gdb/

(6 pts) Implementing 1-D arrays in assignment 3

Task 1 (3 pts): Create a C-style string that will hold a sentence/paragraph inputted by the user. • You can make a static array for this, but make sure your getline has the appropriate length. • Use `cin.getline(array_name, num_chars_to_read)` for C-style strings. • What do you think may happen if you enter more characters than specified by `num_chars_to_read` in `cin.getline()`? The code below may help this problem.

```
if (cin.fail()) {
 cin.ignore(256,'\n'); //get rid of everything leftover
cin.clear(); //reset the failbit for the next cin
}
```

Task 2 (3 pts): Create an array of C++ strings for any N number of words that the user wants to search for and read those words from the user.

- Ask the user how many words they want to search for.
- Create an array of that many words
- Read each word from the user. Remember, you would use getline for C++ strings. `getline(cin, word_array_name[i]);`

You need to have a main function that asks for a sentence/paragraph and for the N words. After getting inputs from the user, print out both arrays to check if they are working properly. Remember to check memory leaks as well.

If any of your functions are more than 15 lines of code, what can you do to make it smaller? If you are having difficulty thinking of how to make it smaller, then ask a TA to help you!!!

Show your completed work and answers to the TAs for credit. You will not get points if you do not get checked off