LAB #8

Each lab will begin with a brief demonstration by the TAs for the core concepts examined in this lab. As such, this document will not serve to tell you everything the TAs will in the demo. It is highly encouraged that you ask questions and take notes.

In order to get credit for the lab, you need to be checked off by the end of lab. You can earn a maximum of 3 points for lab work completed outside of lab time, but you must finish the lab before the next lab. For extenuating circumstances, contact your lab TAs and Jennifer Parham-Mocello.

Reminder: All of our labs involve paired programming. You do not have to keep the same partner for each lab, but you MUST work with someone in each lab!!! **First, find a partner for this lab.** It can be the same partner from the previous lab or a different partner.

(2 pt) Trace Code/Mystery Code

How would you determine what code is doing without using a computer? What if someone forgot to comment the code or properly space their code? You must trace this code and determine what it does without running the code. How will you convince your TA you know what it is doing? mystery.cpp

(2 pts) Debugging Logic Errors (by printing and tracing):

Below is code that has **logic errors**. Logic errors are usually caught by **using print statements** to make sure the contents of variables are what you think they are, printing indicator messages, and **commenting out code** to localize the error.

In the following code, go through the code and systematically fixing the logic errors by inserting print statements to check the contents of the variables. You might want to use comments to narrow the scope of your code to help find the errors. You can copy and paste the code below or download from here: prime_debug.cpp

```
#include <iostream>

#define PROMPT "Please enter a whole number:"
#define NOT_PRIME "The number is not a prime number./n"
#define PRIME "The number is a prime number./n"

using std::cout;
using std::cin;

void is_prime(int);

int main(){
    char number; /* number provided by user */
```

```
         cout << PROMPT;  /* promt user */
         cin >> number;  /* wait for user input */
         is_prime(number);  /*determine if number prime or not*/

         return 0;     /* exit program */
     }
   /* Prime numbers are defined as any number greater
    * than one that is only divisible by one and itself.
    * Dividing the number by two shortens the time it
    * takes to complete. */
     void is_prime(int number){
         for(int i = 3; i < number/2; ++i)
            if( number/i == 0 ){    /* if divisible */
               cout << NOT_PRIME << number; /* not prime */
               return;        /* exit program */
            }
         /* if number is not divisible by anything
          * than it must be prime */
         cout << PRIME << number;
     }
```

(2 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
```

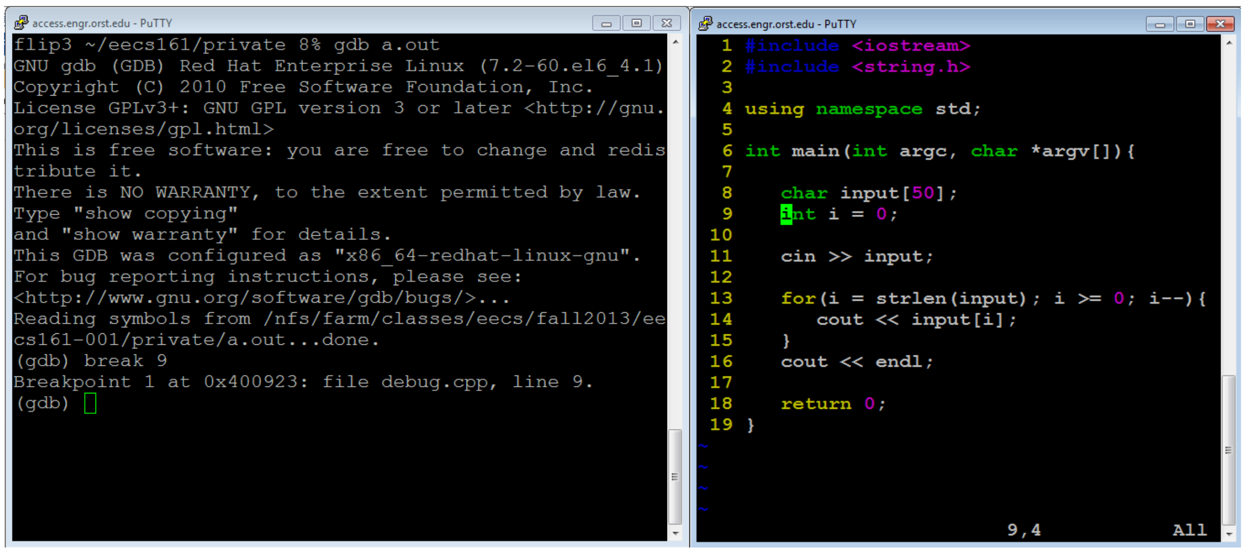
Lets start with a simple program that gets a line of text from the user, and prints it out backwards to the screen:

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

using namespace std;
```

```
int main(int argc, char *argv[]){
  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)
```



```
flip3 ~/eecs161/private 8% gdb a.out
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-60.el6_4.1)
Copyright (C) 2010 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 redis
tribute 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 /nfs/farm/classes/eecs/fall2013/ee
cs161-001/private/a.out...done.
(gdb) break 9
Breakpoint 1 at 0x400923: file debug.cpp, line 9.
(gdb)
```

```
  1 #include <iostream>
  2 #include <string.h>
  3
  4 using namespace std;
  5
  6 int main(int argc, char *argv[]){
  7
  8    char input[50];
  9    int i = 0;
 10
 11    cin >> input;
 12
 13    for(i = strlen(input); i >= 0; i--){
 14       cout << input[i];
 15    }
 16    cout << endl;
 17
 18    return 0;
 19 }
~
~
~
~
                                            9,4           All
```

Here is a mini tutorial for the 8 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 6, 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: /nfs/farm/classes/eecs/fall2013/eecs161-001/private/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

Program 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\031\
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\065
```

```
\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\065
\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\00
0\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
```

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(int argc, char *argv[]){
       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 the gdb:

http://classes.engr.oregonstate.edu/eecs/summer2013/cs162-001/4-gdb.mp4
http://www.cs.cmu.edu/~gilpin/tutorial/
https://sourceware.org/gdb/current/onlinedocs/gdb/

(2 pts) Understanding Errors

Here is a functional program that compares two strings supplied as command line arguments to see if they are equal. If they are equal, it returns true, and if they are not equal, then it returns false. Copy and paste this program into a new source file.

```
#include <iostream>
using namespace std;

bool is_equal(char *str1, char *str2) {
   int i;
   for(i=0; str1[i]!='\0' && str2[i]!='\0'; i++)
      if(str1[i]!=str2[i])
         return false;

   if(str1[i]!='\0' || str2[i]!='\0')
      return false;

   return true;
}

int main() {
   char *str1=NULL, *str2=NULL;
   str1=new char[256];
   str2=new char[256];

   cout << “enter string 1: ”;
   cin >> str1;
   cout << “enter string 2: ”;
   cin >> str2;

   cout << is_equal(str1, str2) << endl;

   return 0;
}
```

Now, you are going to deliberately create errors, and record on a piece of paper what you notice about these errors. For example, when you **remove a semicolon from a line 12** in the program, then the compiler **gives you an error on a few lines below**, at the next statement, alarming you that you have a missing semicolon **before** the statement on line 14. Perform each of these errors, and record your understanding of the error and why it is the error it is☺ After you make the error, restore the program back to functional before making another error!!!

Syntax errors (What does the compiler say!!!)

- Remove a semicolon from line 14.
- Remove the curly brace from line 5.
- Add an & to the str1 and str2 arguments in the call to is_equal() on line 27.
- Remove the asterisk/splat from the char pointers str1 and str2 on line 18.
- Remove the return type, bool, from the is_equal() function on line 5.
- Remove one of the parameters in the is_equal() function on line 5.
- Comment out line 6, and declare i in the for loop on line 7.

Logic errors (What is the output!!!)

- Put two asterisks/splats on the parameters of line 5, and put ampersand, &, in front of the arguments on line 27.

Let's use the gdb debugger… If we were to have this type of error and didn't know what is going on, where would be the first place you would look? You should think to yourself, what are the values of str1 and str2 when the function is_equal() is called, i.e. not before the call, but after the call!

Use the gdb debugger to set the appropriate breakpoint in the program (after the call to the is_equal() function but at the declaration of the local variables) and display the values of str1 and str2. Are these what you expect?

Make your changes back and look at what you are passing when the program is correct, rather than when you added the extra splat and ampersand.

- Comment out line 11 and 12, and run with hell and hello as input.
  How would you use gdb to discover this error?

- Comment out line 19 and 20, and run with any strings as input.
  How would you use gdb to discover this error?

The above program allocates memory, but it does not free the memory. We will learn about a utility in Linux that allows us to view how much memory we have allocated and deallocated (freed) during a run of your program. You can run your program through this utility called `valgrind`. For example, type valgrind prog_name at the prompt.

```
     % valgrind str_equal
```

Make sure this program doesn't have any memory leaks using valgrind. What statements do you need to add?

(2 pts) Use a 2-d array

Think about your assignment #5, and how you are going to store the morse codes. If you can only have C-style strings, then what about making an array that holds c-style strings, instead of C++ strings, i.e. `char *morse[26]` vs. string morse[26]. If you do this, then you need to make a new c string on the heap for each letter, and copy the morse code to the string. For instance:

```
     morse[0] = new char[3];  //create space
     strcpy(morse[0], ".-");  //copy string to space
```

OR
You can also do this as a static array using more space than needed, i.e. `char morse[26][6]`, but then you can initialize at the time of declaration:

```
     char morse[26][6]={{".- "},…};  // make & initialize
```

Now, decide how you are going to do this for your assignment #5. Setup and initialize this 2-d array. How will you access the C-style string for each letter? Print the first few morse codes for A, B, C, D, E, F, G, etc. using morse.