LAB #9
Revisit Recursion, 2-D Arrays and Errors/Debugging

Remember, if you need to get Lab #8 graded, you need to show your lab **to the TAs within 10 minutes of getting to lab**, and you and your partner **will not receive lab credit, if you do not get checked off** before leaving each lab. Once you have a zero on a lab, then it cannot be changed because we have no way of know if you were there or not!!!

New Labs: We are going to change the way we do labs by providing relevancy through videos and using larger group activities to encourage design, while utilizing a broader set of strengths. Each lab will begin with a 10-15 minute video shown on the TV by a TA, followed by a large group activity. The group activity requires design, input from everyone, and no computers!!!!

TA Tutorial of GDB

The Lab TAs will provide a mini tutorial for 5 more main commands that you will mostly be using in your debugging session, using the code and information from Lab #8.

1. `break`
2. `print`
3. `next & step`
4. `continue`
5. `display & watch`

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 `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.

3. 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.

4. 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.

5. 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\24
0\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\0
31\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\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 =
"he\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
```

(4 pts) Practice Designing/Understanding the Problem as a Group (NO Computers Allowed– 40 minutes max)

Let's write the **Towers of Hanoi** program using a 2-D array. This is a game where you have 3 posts and n disks of different sizes, which are initially arranged in ascending

order on the 1st post. Your goal is to get the disks arranged on the 2nd post in ascending order using these following rules:

- You can only move one disk at a time.
- You cannot put a larger disk on top of a smaller disk.

First, begin by understanding the program by writing the steps on the whiteboard, as a group, that represents the moves among the posts. For instance, the smallest disk from the 1st post will be moved to the second post, i.e. 0 -> 2 or 1 -> 3 (if you want to write it as the 1st through 3rd post, rather than array indices for columns).

Write the steps for the base case, n = 1 disks, n = 2 disks, and n = 3 disks. You should notice that you have $2^n - 1$ moves for each of these cases. Also, note any pattern that you see, i.e. when do you see the base case.

On way to implement this is using a 2-D array with 3 columns for the 3 posts, and we can initialize the array with the numbers 1, 2, and 3 in the first column to represent the initial state of the game. The goal is to print out the board after each move in the game, seeing the following output. Example with **two disks**:

```
1  0  0
2  0  0
----------
0  0  0
2  0  1
----------
0  0  0
0  2  1
----------
0  1  0
0  2  0
----------
```

First, Understand the Problem Using the Pseudocode below:
The program contains a 2-D array and two functions, **towers() and print_array().** Your towers() function will be recursive with the following prototype:

```
void towers(int disks, int b[][COLS], int from_col, int to_col, int spare);

    If(number of disks is >= 1)
       Call Towers with (disks-1, b, from_col, spare, to_col)
       Move the disk
       Print the board
       Call Towers with (disks-1, b, spare, to_col, from_col)
```

Go through this algorithm as a group, given n=1, n=2, and n=3 disks with a starting column of 1, ending column of 2, and a spare column of 3, and make sure you can explain and understand what is happening in each recursive call!!!

Convince your TAs that you understand the problem, before beginning your design for the towers() and print_array() functions.

(4 pts) Each Pair: Implement Your Group Design
For this lab, you can hard code your array dimensions, number of disks, and from, to, and spare columns.

Convince your TA that your program works with n=1, n=2, and n=3 disks!!!

(2 pts) Understanding Errors:

Individual Work due by Lab #10!!! 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. Download this program is_equal.cpp (or type it out for practice!).

```
#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(int argc, char *argv[]) {
   if(argc >= 3)
      cout << is_equal(argv[1], argv[2]) << 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 11** 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 13. 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 13.
- Remove the curly brace from line 4.
- Remove the brackets from the argv arguments in the call to is_equal() on line 18.
- Remove the asterisk/splat from the argv in main on line 16.
- Remove the return type, bool, from the is_equal() function on line 4.
- Remove one of the parameters in the is_equal() function on line 4.
- Comment out line 5, and declare i in the for loop on line 6.

Logic errors (What is the output!!!)

- Put two asterisks/splats on the parameters of line 4, and put ampersand, &, in front of the arguments on line 18.

  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 10, and run with ./string_equal hell hello.

  How would you use gdb to discover this error?

- Comment out line 16, and run with ./string_equal hello

  How would you use gdb to discover this error?

Extended Learning:

Take your array dimensions, number of disks, from column, to column, and spare column as command-line arguments to your program.