

## 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. For non-zero lab 6 or 7, you can earn full points for completion of the lab outside of lab time, but you must finish the labs before leaving today!!!! For extenuating circumstance, contact your lab TAs and Jennifer Parham-Mocello.

### Pair Programming

From this lab forward, you can choose a partner for pair programming. **You must be checked off together. You only need one computer for pair programming.** One person will be the driver, who controls the computer and codes, while the other is the navigator, who observes and advises. After 20 minutes, your TA will switch driver and navigator, continuing this pattern until the task is complete. Please read more about pair programming and the benefits: [Pair Programming Student Handout](#)

### Lab 8 Details:

In this lab, you will only implement one new concept that is used in your current assignment. Instead, you will debug programs written for you. Due to the number of students who did not finish Lab 6 and 7, I want to give you a little bit of a break!!!

**For non-zero lab 6 or 7, you can earn full points for completion of the lab outside of lab time, but you must finish these labs before leaving today!!!! If you don't finish lab 8, you can get all the points back at the beginning of Lab 9 next week, since lab 8 is just debugging and can be done on your own. If you finish Lab 8 in lab today, then you will be required to do the extended learning, which is part of your lab for next week.**

### (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;

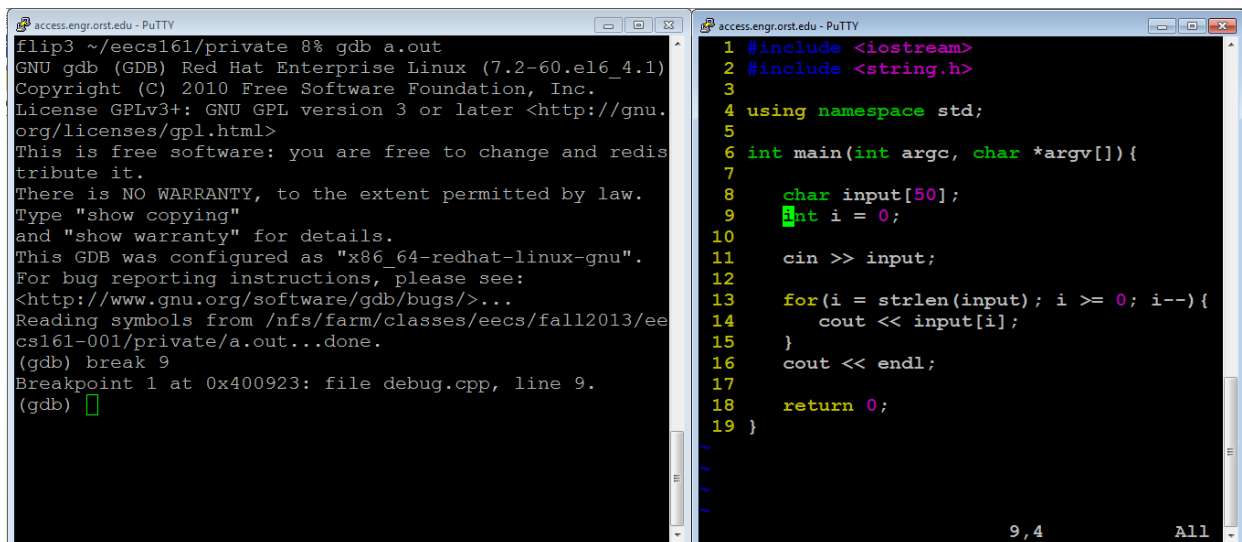
    cin >> input;

    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)
```



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=0x7fffffff008)
   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=0x7fffffff008)  
  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=0x7fffffff008)  
  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=0x7fffffff008) 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=0x7fffffff008) 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/>

### (3 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. Download the following program: [is\\_equal.cpp](#).

```
#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?

### (3 pts) Tracing Code (by printing and by hand):

Below is working code that has **bad variable names and indentation**. Tracing code can be done by **using print statements** (or a debugger) to view the contents of variables/arrays or by using a pen and paper to figure out what is happening.

In the following code, first go through the code and fix all the indentation issues, and then, systematically figure out what the code is doing. You must add comments to the lines of code as you figure out what the code is doing.

You can figure out what the code is doing by inserting print statements to check the contents of the variables/arrays, using the debugger, or tracing the code by hand using pen and paper. You can download the mystery code from here: [mystery.cpp](#)

## (2 pts) Command-Line arguments:

Take the `is_equal.cpp` program from above and modify it so that it gets two strings from the user as command-line arguments, rather than prompting the user for input. For instance, the user should be able to run the program as follows:

```
a.out hello hi
```

Remember, the command line arguments are in a 2-d array, `argv`, and each one dimensional array is just a C-style string, i.e. `argv[0]` is the name of the executable used to run the program, which would be “a.out” in this case.

You will need to add **`int argc`** and **`char *argv[]`** as parameters in `main()`.

## Extended Learning (Lab 9) Implement Towers of Hanoi

First, you can implement this is using a static **2-D array with 3 columns for the 3 posts**, and you 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
-----
```

**Begin by designing these two functions, `towers()` and `print_array()`.** To help you out, 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);
```

Here is an outline of the recursive towers function:

```
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)
```