LAB #8 –Debuggers and Dynamic Arrays

Remember, if you need to get Lab #7 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!!!!

Learning to use a debugger…

GDB with a core file:
https://www.youtube.com/watch?v=mlfz6c9frSU

(2 pts) Debugging with Core file:

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:

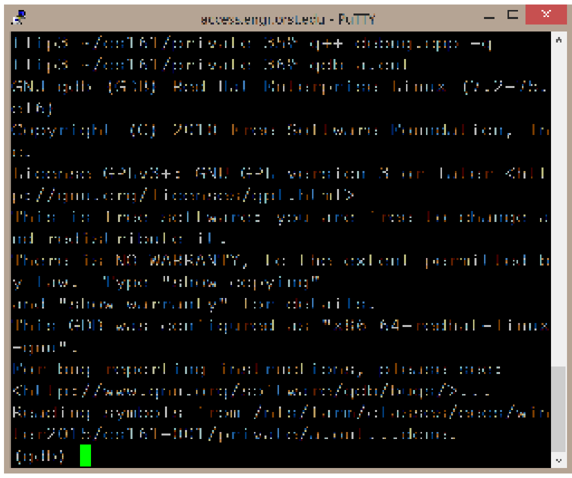
```
g++ filename.cpp -g -o filename
```

Lets start with a simple program that gets a line of text from the user, and prints it out backwards to the screen. This program should crash when you run it.

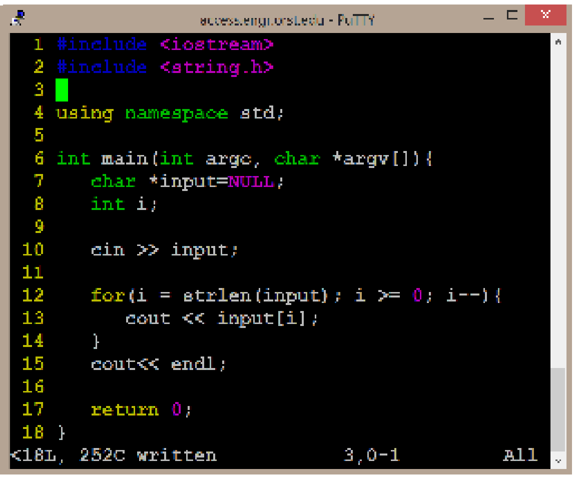
```
#include <iostream>
#include <string.h>

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;
}
```

compile and start the debugger with:

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



```
flip3 ~/cs161/private 358$ g++ debug.cpp -g
flip3 ~/cs161/private 359$ gdb a.out
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-75.
el6)
Copyright (C) 2010 Free Software Foundation, In
c.
License GPLv3+: GNU GPL version 3 or later <htt
p://gnu.org/licenses/gpl.html>
This is free software: you are free to change a
nd redistribute it.
There is NO WARRANTY, to the extent permitted b
y 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/win
ter2015/cs161-001/private/a.out...done.
(gdb)
```



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

using namespace std;

int main(int argc, char *argv[]){
   char *input=NULL;
   int i;

   cin >> input;

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

   return 0;
}
<18L, 252C written              3,0-1         All
```

Here is a quick tutorial to help you with debugging segmentation faults:

1. `run`
2. `where (or bt)`

1. 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/winter2015/cs161-
001/private/a.out
hello

Program received signal SIGSEGV, Segmentation fault.
0x000000325f67b826 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
Missing separate debuginfos, use: debuginfo-install glibc-2.12-
1.149.el6_6.5.x86_64 libgcc-4.4.7-11.el6.x86_64 libstdc++-4.4.7-
11.el6.x86_64
(gdb)
```

2. 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. At the (gdb) prompt from the `run` command, `type` where.

```
(gdb) where
#0  0x000000325f67b826 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  0x000000000040093c in main (argc=1,
    argv=0x7fffffffc788) at debug.cpp:10
(gdb)
```

In the last line of output by the debugger, you can see the program crashed at line 10 in debug.cpp. 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, and you can see the line of code where the program crashed.

```
(gdb) up
#1  0x000000000040093c in main (argc=1,
    argv=0x7fffffffc788) at debug.cpp:10
10          cin >> input;
(gdb)
```

Here we see line #10

```
10 cin >> input;
```

The line where we crashed!!!

(3 pts) Practice Designing as a Group (NO Computers Allowed– 30 minutes max)

We need to get into a bigger group before you begin your paired-programming. Get into groups of 10 for a 30 person lab, i.e. 5 (or less) pairs in 3 different groups. Each group will have a dedicated TA as a project leader. The role of the TA is to make sure all members in the group are participating and that everyone understands the requirements of the problem being solved. In addition, your group might want to dedicate someone with good handwriting to capture your thoughts and design. Each group will begin by writing a flowchart for the solution, and then, write the pseudocode.

Since Assignment #5 is large, we are going to work together to understand the design for using the optional –test argument and redirecting your input from the keyboard to a file. You need to begin by focusing on the create_array() and initialize_array() functionality for your assignment.

Creating your array: you create your array by prompting the user or not prompting the user for the dimensions, depending on whether the –test option is supplied.
Initializing your array: you populate your array with random numbers or an input stream, depending on whether the –test option is supplied.

- **If you run your program without the –test**, then your program will **prompt the user for the rows and columns**, and then **randomly generates integers**, 1-50 inclusively, for the array. You run your program like this:
  ./prog_name

  ```
  Enter the number of rows: 7
  Enter the number of cols: 8
  Your matrix is:
  08 02 22 38 15 05 40 10
  23 50 10 26 38 40 18 38
  26 20 02 12 20 39 08 40
  24 05 26 17 14 34 21 36
  23 09 10 44 20 45 35 14
  40 33 34 31 33 17 28 22
  31 15 03 04 16 14 09 16
  ```

  In order to randomly generate numbers, you use the rand function in the cstdlib (or stdlib.h) library. In order to get different random numbers each time you run the program, you need to seed the random number generator, only once. We use time as our seed (or number used to generate the random numbers), which is in the time library. How to do this:

  ```
  #include <cstdlib>  //this is for srand and rand
  #include <ctime>    //this is for time
  int main() {
     int num;
     srand(time(NULL));
     num = rand() % 50 + 1;  //rand returns ints between 0-RAND_MAX
     return 0;
  }
  ```

- **If you run your program with –test**, then your program **does not prompt the user for dimensions**. It will **read the rows, cols, and integers from a file** by redirecting the input from the keyboard (standard in) to a file. You run your program using the <, redirect symbol, and where the input is coming from:
  ./prog_name –test < test.txt
  Your matrix is:
  ```
  00 01 01
  00 01 01
  00 00 00
  ```

  The corresponding file, **test.txt**, can look like this (expect rows, cols, and ints):
  ```
  3 3
  0 1 1
  0 1 1
  0 0 0
  ```

  Since the input is coming from a file, instead of a user, you do not need a prompt, cout, and you only need to use input statements, cin, to read the data. For example:
  ```
  int main() {
     int rows, cols **a;

     cin >> rows;
     cin >> cols;
     a = create_array(rows, cols);
     for(int i=0; i<rows; i++)
        for(int j=0; j<cols; j++)
           cin >> a[i][j];
     return 0;
  }
  ```

(5 pts) Each Pair: Implement Your Group Design
We need to split back into pairs, and each of the 5 pairs in a group need to implement the design using **create_array(), initialize_array(), and print_array()** functions. Different pairs will finish at different times. After you get checked off by your project manager, please help the other pairs in your group. After everyone in the group has completed the program, then let your project manager know. We are curious how design influences the time spent on implementation.

Show your project manager (TA) how your program works with and without the –test option. Your program should print the array to confirm that the program works correctly.

Continue: If you complete the create_array(), initialize_array(), and print_array() functionality for the –test option, then write the function to find the greatest product of 4 adjacent numbers in box. For example:

Your matrix is:
```
00 01 01
00 01 01
00 00 00
```

Max Prod: 1