Indian Institute of Technology Bombay, Mumbai
Department of CSE, Kanwal Rekhi Building
CS101 – Computer Programming
Autumn Semester 2014-2015

Lab Handout for Week 5 - 25/08/2014 to 29/08/2014

General Instructions: *Please read the instructions carefully before proceeding further*
There are 7 programs given in this handout. You need to write and execute them. All the programs that you write and execute need to be uploaded today before leaving the lab. Take help from your TA to perform the tasks given below:

1. Go to ***http://www.cse.iitb.ac.in/~cs101***
2. Click on '**Lab Assignment Submissions**' link
3. Write your Roll Number in the text box
4. Click '**Choose File**' button
5. Browse through your directory by navigating to the folder in which you have created the project. Select the program i.e. '**.cpp file**' from your project directory.
6. Click '**Submit**' button
7. A new page will open with the message: '***Upload successful. Click here to go back.***'
8. Perform these steps (1 to 7) for all the programs that you have written.

Objective: In this lab, you are required to solve the practice problems discussed last week during the lectures on functions.

Program 1:
In this program, we will write a C++ function intSqRoot that takes a non-negative double input parameter and returns a double value such that

- fractional part of intSqRoot(x) is always 0.0
- intSqRoot(x) is always non-negative
- $(\text{intSqRoot}(x))^2 <= x < (\text{intSqRoot}(x) + 1)^2$

As examples, intSqRoot(4.0) should return 2.0, intSqRoot(6.0) should return 2.0 and intSqRoot(10.0) should return 3.0

The skeleton of the function along with the main program is given below. You are required to fill in the missing code, compile your program and test it by running it and providing as many inputs as you can. Be sure to test for the corner cases: when x is the square of an integer, when it is one less than the square of an integer, and when it is one more than the square of an integer.

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

//   Write your pre-condition here
double intSqRoot(double x) {
   double result;

   // Code to compute intSqRoot using a simple loop, as discussed in class
      Your code comes here

   return result;
}
```

```
//   Write your post-condition here
int main(){
   double x, result;
   cout << "This is my program for testing the intSqRoot function." << endl;
   cout << "-----------------------------------------------------------------" << endl;
   do { // start of do-while loop
    cout << "Enter non-negative number x (-1 to quit) :" ;
    cin >> x;
    if (x == -1.0) break;
    if (x < 0) {
       cout << "Can't compute square root of negative number " << x << endl;
       continue;
    }
    result = intSqRoot(x);

    // Check if answer is correct
    unsigned int intPartOfResult = result;  // Store the integer part of result in intPartOfResult;
    if ((result < 0) || (result > intPartOfResult) ||
        (result * result > x) || ((result + 1)*(result + 1) <= x)) {
          cout << "Oops! My intSqRoot function is not current!" << endl;
          cout << "It gave the wrong answer for intSqroot(" << x << ")" << endl;
          break;
    }
    else {
          cout << "Yeah! My intSqRoot function computed the right answer for " << x << endl;
    }
  } while (true);  // end of do-while loop – is this an infinite loop?
}
```

Program 2:

Write a program that takes the co-ordinates of 3 points , each in 3-dimensional space, and finds the point which is (a) farthest from the origin and (b) nearest to the origin. The program should print the nearest and farthest points, along with the integral part of their distances from the origin. You can use the intSqRoot function written in the previous problem (only after you have tested it well enough and convinced yourself that it's working fine).

Each point is represented by a triple of floating-point coordinates (x, y, z). The distance of the point (x, y, z) from the origin is the positive square root of $x^2 + y^2 + z^2$

As an example, if we provide as inputs the following three points (each point represented by a triple of coordinates), we should get the output as shown below.

```
Inputs:  Point 1: -1.0 -1.0 -1.0
         Point 2: -1.0 -2.0 -1.0
         Point 3: -0.5 0.5 0.5

Output: Nearest point from origin:  -0.5 0.5 0.5
        Integral part of distance from origin:  0.0 (= intSqRoot(0.75))
        Farthest point from origin:  -1.0 -2.0 -1.0
        Integral part of distance from origin: 2.0 (= intSqRoot(6.0))
```

Using the idea of a test program as shown for the previous problem, write a program that reads in three number, finds the nearest and farthest from the origin and prints out the appropriate message.

Use the ideas discussed in class. As already mentioned, you can use the intSqRoot function written as part of the previous problem to solve this problem.

Exercise 3:
We want to write a C++ function that can be eventually used to play a game of tic-tac-toe (using 0's and 1's), as discussed in class.

x1	x2	x3
x4	x5	x6
x7	x8	x9

We will use nine integer valued variables x1, x2, x3, x4, x5, x6, x7, x8, x9 to denote a configuration of the game. The square that each variable represents is as shown in the diagram above. We will use 0, 1 and -1 for the values of each of these variables. A 0 for x1 means that there is a 0 in the corresponding square. A 1 for x1 means that there is a 1 in the corresponding square. A -1 for x1 means that the corresponding square is empty.

0		
1	0	

Thus, the above configuration of the game is represented by the following values of the integers: x1 = 0, x1 = -1, x2 = -1, x4 = 1, x5 = 0, x6 = -1, x7 = -1, x8 = -1, x9 = -1

The following function can be used to print the tic-tac-toe grid on your screen, given the nine integers encoding its configuration. You are required to type this in, and test it out with a few sample configurations. You may find this useful when writing programs for the following questions.

```
void displayTTT(int x1, int x2, int x3, int x4, int x5, int x6, int x7, int x8, int x9) {
  char c1, c2, c3, c4, c5, c6, c7, c8, c9;  // Characters to be displayed in different squares
  // Validate inputs
  if ((x1 < -1) || (x1 > 1) || (x2 < -1) || (x2 > 1) || (x3 < -1) || (x3 > 1) || (x4 < -1) || (x4 > 1) ||
      (x5 < -1) || (x5 > 1) || (x6 < -1) || (x6 > 1) || (x7 < -1) || (x7  > 1) || (x8 < -1) || (x8 > 1) ||
      (x9 < -1) || (x9 > 1)) {
   cout << "Invalid configuration cannot be displayed !!!" << endl;
   return;
  }
  c1 = (x1 == 1) ? '1' : ((x1 == 0) ? '0': ' ');  // There is a blank between the two quotes in ' '
  c2 = (x2 == 1) ? '1' : ((x1 == 0) ? '0': ' ');
  c3 = (x3 == 1) ? '1' : ((x1 == 0) ? '0': ' ');
  c4 = (x4 == 1) ? '1' : ((x1 == 0) ? '0': ' ');
  c5 = (x5 == 1) ? '1' : ((x1 == 0) ? '0': ' ');
  c6 = (x6 == 1) ? '1' : ((x1 == 0) ? '0': ' ');
  c7 = (x7 == 1) ? '1' : ((x1 == 0) ? '0': ' ');
  c8 = (x8 == 1) ? '1' : ((x1 == 0) ? '0': ' ');
  c9 = (x9 == 1) ? '1' : ((x1 == 0) ? '0': ' ');

  cout << "Tic-tac-toe board: " << endl;
  cout << c1 << " | " << c2 << " | " << c3 << endl;
  cout << "-----------" << endl; // Exactly eleven '-'s
  cout << c4 << " | " << c5 << " | " << c6 << endl;
  cout << "-----------" << endl; // Exactly eleven '-'s
  cout << c7 << " | " << c8 << " | " << c9 << endl;
```

```
  cout << endl;
  return;
}
```

Program 4

A valid configuration has the following two properties:

1. Number of "0"s in the configuration is either equal to the number of "1"s or is more by one.
2. There can be at most one winning line (horizontal, vertical or diagnoal) of "0"s or "1"s.

Note that there can be 8 winning lines: three horizontal for each row, three vertical for each column and two diagonal.

Write a C++ function "tttCheckConfig" that checks if a configuration of the game is a valid one. Your function should take as inputs nine integers (x1, x2, … x9) and return a bool value. The function should check whether the configuration is valid. If so, it should return true, else it should return false.

You can use the following skeleton of the function provided for your convenience.

```
// Write your pre-condition here
bool tttCheckConfig(int x1, int x2, int x3, int x4, int x5, int x6, int x7, int x8, int x9)  {
   int numZeros, numOnes;
   // Count number of 0's and 1's.
   // Check if number of 0's and number of 1's are fine.

   // Check how many winning lines of 0's exist.  You will have to check for each of the 8
   // possible winning lines.
   // Check if 0 has more than one winning line.

   // Check how many winning lines of 1's exist.
   // Check if 1 has more than one winning line.

   // Check if both 0 and 1 have at least one winning line each.

   // Return appropriate value
}
// Write your post-condition here
```

Optional: If the function finds that a configuration is not valid, it can print some diagnostic messages telling the user why the configuration is not valid.

Program 5:

Continuing with our game of tic-tac-toes, we say that "1 has a winning configuration" if the current configuration is a valid one, and if there is a winning line (horizontal, vertical or diagonal) of "1"s. The case of "0 has a winning configuration" is similarly defined.

Write a C++ function "tttReferee" that takes as inputs a sequence of 9 integers in {-1, 0, 1} representing a configuration of tic-tac-toe, and returns 1 if "1" has a winning configuration, 0 if "0" has a winning configuration, and returns 2 otherwise (neither "0" nor "1" has a winning configuration). If there is an error, you can have the function return -1. You can also use the function tttCheckConfig you wrote as part of the previous problem in this question.

```
//Write your pre-condition here
int tttReferee(int x1, int x2, int x3, int x4, int x5, int x6, int x7, int x8, int x9) {
  if (tttCheckConfig(x1, x2, x3, x4, x5, x6, x7, x8, x9) == false) {
    cout << "Invalid configuration argument of tttReferee(...)" << endl
    return -1;
  }

  // Check if "0" has a winning configuration.  You'll have to check all 8 possible winning lines.
  // If so, return 0;

  // Do the same as above for "1".

  // If neither "0" nor "1" has a winning configuration, return 2.
}
// Write your post-condition here
```

Program 6:
Write a C++ function that takes as input an input configuration and determines who ("0" or "1") should move next. Note that this can be easily determined by counting the number of 0's and 1's in the configuration. Your function should of course check if the input configuration is valid. If the next move is for "0", the function should return 0. If the next move is for "1", the function should return1. If one of the players has already won, the function should return 2. If an error condition is encountered, the function should return -1.

```
// Write your pre-condition here
 int nextTurn(int x1, int x2, int x3, int x4, int x5, int x6, int x7, int x8, int x9)
 {  // Check if configuration is valid
    // Check if somebody has already won in the current configuration.  If so, return 2;
    // Count no. of 0's and 1's to determine the player who should play next, and return
    // appropriate value.

 }
// Write your post-condition here
```

Program 7:
Given a configuration of tic-tac-toe, we want to determine if there is a winning/losing move of the current player.
Winning Move of "0": A move of "0" (an empty position in the tic-tac-toe grid where one can place a 0) such that after this move, there is at least one way for "0" to win *no matter how "1" plays.* For example, if a move of "0" is such that the resulting configuration results in tttReferee returning 0, then this is a winning move of "0".
Winning Move of 1: This is defined similarly to a winning move of "0".
Losing Move of 0: A move of "0" (an empty position in the tic-tac-toe grid where one can place a 0) such that after this move, there is at least one winning move of "1" from the configuration resulting from "0"'s move.
Losing Move of 1: This is defined similarly to a losing move of "0".

Useful observation to formulate a recursive solution:
A move of "0" from a given configuration is a winning move if and only if the configuration resulting from this move is such that every move of "1" from the resulting configuration is a losing move for "1".
Similarly, a move of "1" from a given configuration is a winning move if and only if the

configuration resulting from this move is such that every move of “0” from the resulting configuration is a losing move for “0”.

A move of “0” from a given configuration is a losing move if and only if the configuration resulting from this move is such that there is at least one winning move of “1” from the resulting configuration.
Similarly, a move of “1” from a given configuration is a losing move if and only if the configuration resulting from this move is such that there is at least one winning move of “0” from the resulting configuration.

Write mutually recursive C++ functions 'winMove' and 'loseMove', such that each takes as inputs (i) a configuration (i.e. nine integers), (ii) next player (0 or 1) and the proposed next move (position on the grid), and determines if the proposed next move of the next player is a winning move or a losing move for that player.

You are free to use the functions developed in the previous problems in this problem. We have provided below skeletons of the functions winMove and loseMove that you are required to complete.

```
// Write your pre-condition here
int winMove(int x1, int x2, int x3, int x4, int x5, int x6, int x7, int x8, int x9, int nextPlayer, int move)
{   // Check if configuration is valid.  Use the function tttCheckConfig designed in Program 3.
    //  Check if one of the players has already won.  Use the function tttReferee designed in Program
    //  4.  In this case, return -1 (since the next move cannot be taken if one of the players has already
    //  won.
    //  Check if nextPlayer matches with what function nextTurn (see Program 5) returns.
    //  Check if the position given by the input parameter “move” is empty
    //  If so, check if the configuration resulting from the proposed “move” of “nextPlayer” is valid.
    //  Use the function tttCheckConfig for this.
    //  Check if “nextPlayer “ wins as a result of the proposed move.  Use the function tttReferee for
    //  this.  If so, return true – this is a winning move for “nextPlayer”.
    // Otherwise, starting from the configuration resulting from the proposed “move” of “nextPlayer”,
    // check if all possible moves of the opponent (1 - “nextPlayer”) is a losing move.  You will need
    // determine which are the empty positions remaining after the proposed “move” of “nextPlayer”,
    // and check whether the opponent's move in all of these positions is a losing move.  You must
    // use the function “loseMove” for this purpose.  This is where mutual recursion comes in.
}
// Write your post-condition here.  What were the termination cases of the mutual recursion?

// Write your pre-condition here
int loseMove(int x1, int x2, int x3, int x4, int x5, int x6, int x7, int x8, int x9, int nextPlayer, int move)
{  // Check if configuration is valid.  Use the function tttCheckConfig designed in Program 3.
    //  Check if one of the players has already won.  Use the function tttReferee designed in Program
    //  4.  In this case, return -1 (since the next move cannot be taken if one of the players has already
    //  won.
    //  Check if nextPlayer matches with what function nextTurn (see Program 5) returns.
    //  Check if the position given by the input parameter “move” is empty
    //  If so, check if the configuration resulting from the proposed “move” of “nextPlayer” is valid.
    //  Use the function tttCheckConfig for this.
```

```
    //  Starting from the configuration resulting from the proposed “move” of “nextPlayer”,
    //  check if there is at least one  moves of the opponent (1 - “nextPlayer”) that is a winning move.
    // You will need to determine which are the empty positions remaining after the proposed “move”
    // of “nextPlayer”,  and check whether the opponent's move in at least one of these positions is a
    // winning move.  You must use the function “winMove” for this purpose.
    // This is where mutual recursion comes in.
}
// Write your post-condition here.  Ensure that the mutual recursion terminates
```

Optional Program 8.
Use the winMove and loseMove functions developed in Program 6 to write a program that can interactively play a game of tic-tac-toe with the user, and is guaranteed to never lose.

Play a game of tic-tac-toe with your program.