CSE 332 Lab 3:Tic-Tac-Toe Game

Due by Tuesday October 22nd at 11:59 pm
Final grade percentage: 12 percent

Objective:

This lab is intended to extend your use of basic C++ language features from the previous labs, and to give you experience using object-oriented programming techniques in C++, including:

- encapsulating game data within a class,
- prompting for and receiving user input for game moves,
- determining whether or not a move is valid, and if it is
- updating and displaying the new state of the game

To do this, you will develop a C++ program with a game class that can
(1) repeatedly display the current state of a game
(2) prompt the user and receive input for the next move in the game,
(3) check whether or not each input move is valid, and if it is update the game state; and
(4) detect the following three conditions: successful completion of the game, if the game has reached a point where no further valid moves are possible, or if there are still remaining valid moves available.

In this lab, you are again encouraged to re-use and modify code from your previous lab solution(s) - if you choose to do this, please put a completely new copy of any header and source files from your previous lab(s) in the new lab3 repository.

Assignment:

Part I - Resources and Guidelines:

1. The following readings in the textbook may be useful while working on this lab assignment, and selectively looking up topics and/or reading ahead (on demand as you encounter issues while working on the lab) in the textbook, lecture slides, and studio exercises is also encouraged:
    - Functions: LLM Chapter 6
    - Classes: LLM Chapter 7
    - The IO Library: LLM Chapter 8
    - Sequential Containers: LLM Chapter 9

2. The C++ Strings, Arrays, Vectors, and IO lecture slides, the C++ classes lecture slides, and the IO Library lecture slides, also may be helpful when implementing parts this lab.

3. Please read the following CSE 332 Programming Guidelines which are relevant to the previous labs and please follow them as you implement your solution: **A.1-A.7,**

A.10-A.11, A.17, B.1-B.3, B.13-B.17, B.20-B.24, B.29, B.31-B.33, and C.1-C.8.

Please also review and follow these additional guidelines that may be relevant to this lab: **A.8-A.9** (exceptions), **A.12-A.15** (pointers, arrays, and buffers), **A.16** (debugging), **B.10-B.12** (constructors), **B.31** first part (inclusion guards), and **C.5** (pointer and array index arithmetic).

Part II - Program Design and Implementation:

Note: the details of this assignment are again intentionally somewhat under-specified, leaving you some room to choose what you think is the best way to implement them, as long as what you do is reasonable and you explain your design decisions in comments in the code and in your Readme.txt file.

4. Accept the lab3 assignment from the invitation link here.

5. Open up Visual Studio 2019, connect to GitHub, and clone your newly created repository to create a local working copy on your h: drive. Create a new C++ Console Application within your repository, make sure to name it something descriptive such as *Lab3*.

6. Declare and define a class that implements a version of the Tic-tac-toe game (e.g., of type TicTacToeGame) whose member variables are all private, as follows:

    - The game board is five squares wide and five squares tall. The lower left square has coordinate 0,0 (as in the previous lab) and the upper right square has coordinate 4,4.

    - Each piece in this game is either an X or an O, and should be represented as such on the terminal screen when the board is printed.

    - When the game begins, the inner nine squares of the board (coordinates 1,1 through 3,3) should be empty and waiting for user input, and the outer sixteen squares of the board (each of which has a 0 or 4 in the horizontal or vertical portion of their coordinate) should be empty and stay empty.

    - Outside your game class, declare and define an insertion operator (operator<<) that takes a reference to an ostream and a reference to a const game class object as parameters, and returns the ostream reference that was passed in (this allows the operator to be called repeatedly on the same ostream object as in cout << tictactoe_game << endl;). Inside your game class, declare the insertion operator to be a friend so that it can access the private member variables of the game object. The operator should print out the current state of the game board with 'X' or 'O' characters for squares that have pieces on them, and spaces for empty squares, with the horizontal and vertical coordinate axes labeled. For example, the initial board might be printed out as:

```
4
3
2
1
0
 0 1 2 3 4
```

- Declare and define a public non-static done() method in the game class that takes no parameters (other than the this pointer that's passed implicitly to all non-static methods and operators) and has a bool return type. This method should return true if 3-Xs or Os are in a vertical, horizontal or diagonal line; otherwise the method should return false.

- Declare and define a public non-static draw() method in the game class that takes no parameters (other than the this pointer that's passed implicitly to all non-static methods and operators) and has a bool return type. This method should return false if there are moves remaining in the game (as defined below) or if the done() method returns true; otherwise (if there are no more moves remaining and the game has not been completed successfully) the draw() method should return true.

- Declare and define a public non-static prompt() method in the game class that takes references to two unsigned integers. The method should (repeatedly if necessary) use cout to prompt the user to type a line that either is "quit" to end the game or a string representing a valid coordinate of a square on the board as a comma-separated pair of unsigned decimal integers (for example "0,0" indicates the coordinate of the square at the lower left corner of the board).

  Exactly how you handle the cases where the user inputs an invalid coordinate, a badly formed input string, etc. is up to you, but essentially the user should be re-prompted to input either "quit" or a valid coordinate until she or he does so.

  If the user inputs "quit" the method should return a value indicating the user asked to quit (or you could design it to throw an exception that the program's main function or the play function should catch).

  If the user inputs a string with the coordinate of a valid square on the board, the method should extract the value to the left of the comma into the first (horizontal) unsigned integer parameter that was passed to the method, and extract the value to the right of the comma into the second (vertical) unsigned integer parameter that was passed to the method.

  Hint: one way to implement this method is to store the user's input in a C++-style string, find the comma character in that string and replace it with a space character, wrap the string in an istringstream, and then extract out the values from the istringstream into the two unsigned integer variables (while of course testing whether each extraction succeeded or not).

- Declare and define a public non-static turn() method in the game class that takes no parameters. The game should remember which player's turn it is. The method should print out a message to that effect (player X, or player O), and the method should alternate whose turn it is each time it is called. The method should (repeatedly if necessary) call the prompt() method to obtain coordinates for a valid move in the game, or to determine that the user has quit the game (if the user quits, the turn() method should immediately return an appropriate failure value, instead of doing any further processing).

  A move is *valid* if and only if it moves a piece to an empty square. Specifically, a valid move is within the 9 inner squares of the game board.

  If a valid move has been made, the method should move the piece to the square. Unless the user quits, the method should require one valid move to be made before returning

  After each valid move, the method should print out the updated game board (using the ostream operator described above) and then after an intervening blank line, print out a single line showing the valid moves that have been made as in:

  ```
  4
  3    X O
  2    X O
  1
  0
    0 1 2 3 4

  Player O: 3, 2; 3, 3
  ```

- Declare and define a public non-static play() method that takes no parameters, prints the game board (using the ostream operator described above) and then repeatedly calls the turn() method and then the done() and draw() methods until either: (1) the done() method returns true in which case the program should print out a message indicating 'X' or 'O' won the game and then return a success code (with value 0); or (2) the draw() method returns true in which case the program should print out a message indicating how many turns were played and saying that no winning moves remain (the game is a draw) and then return a unique non-zero failure code; or (3) the user quits, in which case the program should print out a message indicating how many turns were played and saying that the user quit and then return a different unique non-zero failure code.

7. In your program's main function:
   - Check that exactly one argument (or two if you did the extra credit portion) has been passed to the program (in addition to the program's name), and if so that the passed argument is "TicTacToe". If not, your program should pass the program name and a second string showing the format of the command line arguments that need to be given, into a call to the usage function you implemented in the previous lab, and then return the (unique non-zero) value that was returned by that function.

- Declare an object of the TicTacToe game class (e.g., of type TicTacToeGame).

- Call the object's play() method, and use the result returned by that method as the program's return value.

- Check for non-zero values returned by (and catch any exceptions thrown by) the object's methods, and return an appropriate unique non-zero value from the program for each different failure case.

  Note that in this lab (and in the remaining labs this semester) your main program should only return 0 if the game being played was successfully completed, and otherwise (if the user quit or the game reached a point where successful completion was no longer possible) should return a unique non-zero value for each different failure case.

8. Build your project, and fix any errors or warnings that occur. Please be sure to note in your ReadMe.txt file all of the different kinds of errors or warnings you ran into as you developed your lab solution (though you don't have to list every instance of each kind) in your ReadMe.txt file. **If you were fortunate enough not to run into any errors or warnings, please note that instead, where you would have listed errors or warnings in your ReadMe.txt file.**

9. Open up a Windows console (terminal) window and change to the directory where your program's executable file was created by Visual Studio.

10. Run the executable program through a series of trials that test it with good coverage of cases involving both well formed and badly formed inputs on the command line and from the user.

    In your ReadMe.txt file please document which cases you ran (i.e., what the command lines were and what additional cases you added to the files) and summarize what your program did and whether that is correct behavior, in each case.

    You should first make sure the program behaves correctly when run with a well-formed command line, as in:

    lab3.exe TicTacToe

    Make sure your program (1) can handle valid and invalid move inputs from the user and keep on playing, (2) allows the user to quit when they choose, and identifies (3) successful completion of the game, or (4) when no further winning moves remain.

    Then test your program's handling of badly formed command lines (e.g., without a game name, or with a game name other than "TicTacToe").

11. In your ReadMe.txt file, make sure that your name and the lab number (lab 3) are at the top of the file, and that you've documented whether you ran into any errors or warnings

while developing your solution (and if so what they were) and what the executable program did for each of the trials you ran. Be sure to save your changes to your ReadMe.txt file.

12. Stage and Commit your changes with a useful commit message. Once committed, push your commit to your remote repository on GitHub. Log on to GitHub and ensure your code was pushed successfully. **If you forget to push your code, it cannot be graded!**

Part III - Extra Credit: (optional, worth up to 5% extra credit)

The goal of this optional part of the assignment is to make further use of the class methods and operator you have defined, to play the game automatically. Doing this exercise will encourage you to factor out portions of your code that are relevant to both automatic and user-driven play (e.g., the parts that evaluate moves in the game) from portions that are specific to user-driven play (e.g., the part that prompts for input), which is likely to produce good modularity overall.

For extra credit, please do the following:

13. In your TicTacToe game class, declare and define an auto_player() method that takes no parameters, *determines* (i.e., hard-coding a sequence will not earn extra credit) the move for that turn to try to win the game

    Hint: generating all possible valid moves for each game board state, and being able to backtrack in the sequence of turns (and possibly within the moves made within each turn) and reset the state of the game board to an earlier state can help significantly with this.

14. Modify your main function so that it accepts either one argument or two (in addition to the program name), and if two arguments are given that the first of those arguments is "TicTacToe" and that the second of them is "auto_player" and if so calls the auto_player() method instead of the prompt() method when it is the auto_player's turn (also update your usage message so that the user can be told how to run the program correctly, with or without the additional option).

15. Add a section marked "Extra Credit" at the end of the ReadMe.txt file you submit with your solution, and in that section summarize how you designed, implemented, and tested the extra credit features. In that section also show the output produced by your program when run with the extra credit option.

    Please ask for help from your professors or teaching assistants if you are uncertain about how to do this extra credit part or if you run into any difficulty with it.

Project Designed by Chris Gill