Lab 2 : A Tic-Tac-Toe Game

1 Objectives

The main objective of this assignment is to introduce you to the use of classes, objects, and methods, thus applying the related concepts presented in the lectures. You will do so by building a simple Tic-Tac-Toe game. You will first implement a class that represents the state of the game and a function that implements the logic of the game. You will then write a main function that uses these classes to run a game.

2 Tic-Tac-Toe

Tic-tac-toe is a simple game commonly played by children. Two players, `X` and `O`, take turns marking the spaces in a 3×3 grid or board. The player who succeeds in placing three of their marks in a horizontal, vertical, or diagonal line wins the game. Player `X` is always the first to place a mark. The following example shows the progression of a game won by player `X`.

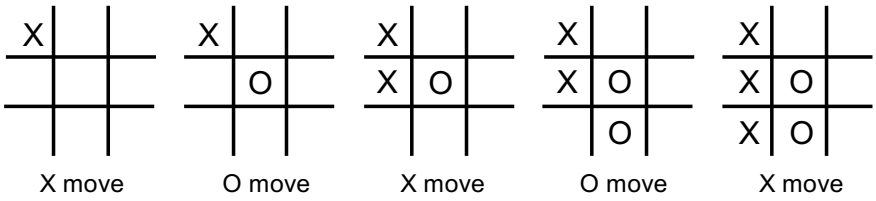

Figure 1: An example Tic-tac-toe game

The simplicity of the game makes it possible for each player to make a perfect move. Thus, the game often ends in a draw. A player wins only if the opponent makes a mistake (which is why the game is played only by children).

3 Game Overview

The game consists of two main components: the *game controller* or simply the *controller* and the *game logic*. This is shown pictorially in Figure 2. The controller is responsible for collecting players' input to determine the grid (or board) location a player wishes to place a mark in. It is also responsible for displaying the game board in the text format described later in this handout. The controller is implemented by the `main` function of the game. The game logic contains a single function called `playMove`. This function determines, for each player input, if the move represented by the selected board coordinates are valid or not, if the game is over or not and accordingly "plays" the move.

The controller and the game logic interact using an object of the type `GameState`. This object stores the state of the game, including the board coordinates selected by a player, the marks at each board location (i.e., `X` or `O`) , whether the move is valid or not, whose turn it is, whether the game is over or not, etc.

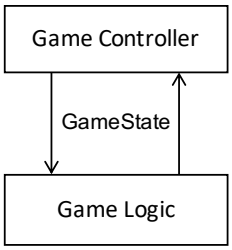

Figure 2: The game components

The game operation is simple. The controller prompts a player to enter two integers, representing the coordinates of a board's location. If the coordinates are legal (see below), it updates the `GameState` object with these coordinates and invokes the `playMove` function of the game logic, passing to it (by reference) the `GameState` object. This function updates the `GameState` object based on the selected coordinates. When the function returns to the controller, it prints the updated `GameState`. The process repeats until there is a win or draw.

4 Problem Statement

You will implement the methods of the class `GameState`, as defined in `GameState.h`. You will also implement a function `playMove`, which "plays" the move indicated by a player's input. The remainder of this section describes: (1) the `GameState` class, (2) the methods of which you must implement, (3) the `playMove` function that implements the game logic, (3) the controller functionality implemented in `main` and (4) other key files that make up the game.

4.1 The `GameState` Class

The state of a tic-tac-toe game is represented by an object of the `GameState` class. This object is created, initialized, and eventually destroyed by the controller. The definition of this class appears in the file `GameState.h`, which is released with the assignment. **You may NOT modify this file** to add to or delete from its content. Modifying the file often results in a mark of zero for the assignment. The class contains the following data members:

- `gameBoard` is a `boardSize` $\times$ `boardSize` (3×3 in this assignment) two-dimensional array that represents the game board. It stores the marks of each player, and thus the state of the game. Each element of the array can be one of `Empty`, `X` or `O`. The elements of the array are initialized by the controller to `Empty`. The definitions of `boardSize`, `Empty`, `X` and `O` appear in the file `globals.h`, which is also part of the assignment release. **You may NOT modify this file** to add to or delete from its content. Modifying the file often results in a mark of zero for the assignment.

  In the array, `gameBoard[0][0]` represents the top-left corner cell of the game grid. Similarly, `gameBoard[boardSize-1][boardSize-1]` represents the bottom-right corner cell of the board. This effectively defines the row and column coordinates of each cell in the grid.

- `selectedRow` and `selectedColumn` are two integers that store the grid coordinates that are selected by the player.

- `moveValid` is a Boolean variable that is set to `true` when the selected coordinates represent a valid move for the current game. That is, it is set to `true` when the grid cell at `selectedRow` and `selectedColumn` is empty and is set to `false` otherwise.

- `gameOver` is a Boolean variable that should be set to `true` if the game is over as a result of the last selected coordinates (i.e., win or draw) and to `false` otherwise.

- `turn` is a Boolean variable that indicates whose turn it is, `X` (`true`) or `O` (`false`) for the current selected coordinates. If the move is valid, then the value of `turn` should be changed by the game logic from `true` to `false` or from `false` to `true` to reflect the change in turn. The controller does not use this variable at all.

- `winCode` is integer variable is set to a code that indicates which cells on the board have marks that form a line, as shown in Table 1. If `gameOver` is `false`, the code should be set to 0. If `gameOver` is `true` and the game is a draw, then `winCode` should also be set to 0. if `gameOver` is `true` and one of the players won, the code should be set to one of the integer values as indicated in the table.

Code	Sequence
0	No win
1	Row 0 of the grid, cell (0,0) to cell (0,2)
2	Row 1 of the grid, cell (1,0) to cell (1,2)
3	Row 2 of the grid, cell (2,0) to cell (2,2)
4	Column 0 of the grid, cell (0,0) to cell (2,0)
5	Column 1 of the grid, cell (0,1) to cell (2,1)
6	Column 2 of the grid, cell (0,2) to cell (2,2)
7	Left to right diagonal, cell (0,0) to cell (2,2)
8	Right to left diagonal, cell (2,0) to cell (0,2)

Table 1: `winCode` values

The `GameState` class provides accessors and mutators to the respective class data members. Please read the comments in the `GameState.h` file to find out what these methods do.

4.2 The `playMove` Function

The game logic implements a single function:

```
void playMove(GameState& game_state)
```

This function is to be implemented in the file `playMove.cpp` and it is called every time a player makes a move. Its goal is to "play" the move and update the `GameState` object that is passed by reference to function. Upon completion, the function must update the game state object by updating:

- The game board at the appropriate location by either `X` or `O`.

- The `turn` value to reflect that the turn has changed.

- The Boolean variable `validMove`, described earlier.

- The Boolean variable `gameOver` to either `true` or `false` to reflect if the move ends the game in either a win or a draw.
- The variable `winCode` to either 0 if the game is not over or to the winning code (as described above) if the game is over.

The two game grid examples shown in Figure 3 are used to demonstrate the expected updates to the game state object.

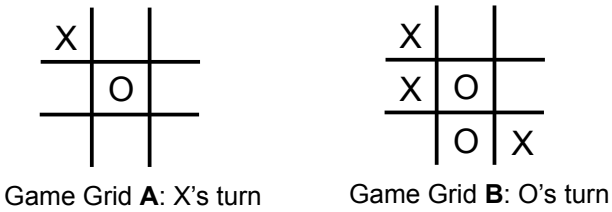

Figure 3: Input game boards

For game grid **A** on the left, the `turn` variable is `true`, indicating that it is X's turn to play. If `selectedRow = 1` and `selectedColumn = 2` when the `playMove` function is called, then an X is placed in the second row and third column of the game board. The move is valid (`validMove` is set to `true`), but the game is not over (`gameOver` is set to `false`). The variable `winCode` is set to 0. The variable `turn` is changed from `true` to `false` to indicate that it is now O's turn to play.

However, if `selectedRow = 1` and `selectedCol = 1` when the `playMove` function is called by the controller, then the move is not valid (O's mark already occupies the cell). The game board is not updated. Further, `validMove` is set to `false`, `gameOver` is set to `false` and `winCode` is set to 0. The variable `turn` is **not** changed to indicate that it is remains X's turn to play with a valid move.

Similarly, for game grid **B** on the right, upon entry to `playMove`, `turn` is `false` (or `0`) indicating that it is O's turn to play. Thus, if `selectedRow = 0` and `selectedCol = 1`, the game state object should be updated to have `validMove` as `true`, `gameOver` as `true`, the game board updated with an `O` in row 0 and column 1, and `winCode` set to 5.

4.3 The Game Controller

The game controller is implemented in the `main` function of the game, contained in the file `tictactoe.cpp`. Its purpose is to: (1) create and initialize the game board, (2) prompt the player for a pair of integers representing the row and column coordinates of the grid cell the player wishes to mark (where row 0 and column 0 represent the upper left corner of the grid and row 2 and column 2 represent the lower right corner), (2) check that the player enters integers that are in the range of 0–2, (3) set the `selectedRow` and `selectedCol` members of the `GameState` object to the entered values, (4) call the `playMove` function, and (5) print the `GameState` object.

You may assume that a player always enters integer input (i.e., not float or string, for example). Thus, you need not handle such errors in your code. The printing of the `GameState` object is in the following format:

- The selected row and column values
- The text "Game state after playMove:" on a line by itself.
- The text "Board:" on a line by itself.

- The grid cell values, printed as `X`, `O` or `B` (for blank). They are printed one row at a time starting with the top row (i.e., row 0). Each row is indented from the left by exactly three spaces and its values are separated by exactly a single space.
- The value of `moveValid`.
- The value of `gameOver`.
- The value of `winCode`.
- A blank line.

The following is a sample run showing the output produced in response to player inputs.

```
Enter row and column of a grid cell: 1 1
Selected row 1 and column 1
Game state after playMove:
Board:
   B B B
   B X B
   B B B
moveValid: true
gameOver: false
winCode: 0

Enter row and column of a grid cell: 0 1
Selected row 0 and column 1
Game state after playMove:
Board:
   B O B
   B X B
   B B B
moveValid: true
gameOver: false
winCode: 0

Enter row and column of a grid cell: 0 1
Selected row 0 and column 1
Game state after playMove:
Board:
   B O B
   B X B
   B B B
moveValid: false
gameOver: false
winCode: 0

Enter row and column of a grid cell: 0 0
Selected row 0 and column 0
Game state after playMove:
```

```
Board:
   X O B
   B X B
   B B B
moveValid: true
gameOver: false
winCode: 0

Enter row and column of a grid cell: 7 0
Invalid board coordinates 7 0

Enter row and column of a grid cell: 2 0
Selected row 2 and column 0
Game state after playMove:
Board:
   X O B
   B X B
   O B B
moveValid: true
gameOver: false
winCode: 0

Enter row and column of a grid cell: 2 2
Selected row 2 and column 2
Game state after playMove:
Board:
   X O B
   B X B
   O B X
moveValid: true
gameOver: true
winCode: 7
```

4.4 Include Files

In addition to the `GameState.h` file described above, the file `globals.h` has global definitions for `X`, `O` and `boardSize`. You should include these files in your `tictactoe.cpp` and `playMove.cpp`. Again, **you may NOT modify these files** to add to or delete from their content. Modifying the files commonly results in a mark of zero for the assignment.

4.5 Reference Executable

In order to help you observe the behaviour of the game, a reference executable called `tictactoe-ref.exe` is released with the assignment. When in doubt about what the game should do or print to the output, you can use this reference to play the game.

Please note that this reference executable works only on ECF machines and on ECE244VM. It will not work in Window machines or Macs. Thus, to use it, you must connect to ECF, as is

described in the "Remote Connection to ECF" handout, or must have ECE244VM installed on your home machine.

5 Procedure

Create directory called `lab2` in your `ece244` directory. Make sure that the permissions of this new directory are such that it is readable by none other than you (refer to lab assignment 1 for how to do so). Download the `zip` file containing the assignment release files and place it in this `lab2` directory. Unzip the file, which will create the assignment files in the directory. You can move the zip file out of the directory or remove it after this step.

There are six assignment files. You will add your code in three files: `GameState.cpp`, `playMove.cpp` and `tictactoe.cpp`, The first contains the implementation of the methods of the class `GameState`. The second contains the "logic" of the game in the `playMove` function. The third contains the `main` function that implements the game controller. All three files contain some skeletal code to get you started, particularly in `tictactoe.cpp`.

In addition, there are two include files `globals.h` and `GameState.h`. Remember that **you may NOT modify these files** to add to or delete from their content. Modifying the files commonly results in a mark of zero for the assignment. Finally, there is a `Makefile` that is used by `NetBeans` to separately compile your project. Do not modify this file either.

Use the assignment files to build a C++ project for `NetBeans`. Start `NetBeans` and create a new C++ project with existing sources. In the resulting dialog window, browse to your `ece244/lab2` directory and click "Select". Ignore the warning message and click "Finish". This creates and builds your project in directory `nbproject` inside your `ece244/lab2` directory. You can now start to add code into your `.cpp` files.

The `~ece244i/public/exercise` command is helpful in testing your code. The executable (i.e., `tictactoe.exe`) is in your `ece244/lab2` directory. Run exercise as follows:

```
~ece244i/public/exercise 2 tictactoe.exe
```

The `exercise` command will let you know if your code has errors by providing it with several test cases. Please note that some of the `exercise` test cases will be used by the autotester during marking of your assignment. However, we will not provide all the autotester test cases in `exercise`, so you should create additional test cases yourself and ensure you fully meet the specification listed above.

Please note that there is behaviour of the game that is not completely specified in the handout (i.e., "corner cases"). You must think of test inputs for such cases. The reference solution can be used to find out what the output is for these cases.

6 Marking and Deliverables

You must submit your code for autotesting. The autotester will be used to check the correctness of your `GameState` implementation, your `playMove` function and your `main` function. Thus, you need to submit only the three files: `GameState.cpp`, `playMove.cpp` and `tictactoe.cpp`. Your copies of the `.h` files are ignored, since you are not allowed to modify them.

To submit your code, *make sure you are in the directory that contains the source files, i.e., your* `ece244/lab2`. Submit your `GameState.cpp`, `playMove.cpp` and `tictactoe.cpp` files as lab 2 using the command:

```
~ece244i/public/submit 2
```