Project 3:
Virtual Realities

Project due on your Final date[1]

Overview

In this project, you will implement Tic Tac Toe in the same pattern as how your Project 2 implemented Othello. With two games now completed (plus a third that I provide), you will change your main to let the user choose which game they wish to play. Both your games will derive from a generic set of base classes that use virtual methods to allow specific games to override generic game-related functions like getting possible moves, applying moves, etc.

You will be given a ZIP file containing starting points for most of the .h files in this project. Those .h files have hints and requirements inside of them; you must add any required functions to the .h files as noted in those files, and then implement every function in a corresponding .cpp file that you will write in its entirety.

Base Classes and Virtual Functions

The "big picture" point of this project is that the *controller* portion of project 2 (the main, which interacts with the user) really doesn't do anything specific to othello. The main prints a "board", shows some "possible moves", reads in a string representing a "move", verifies that move is "possible", and "applies" the move, then repeats. None of that has anyting to do with othello, and in fact the same logic could be used to run any two-player alternating-turns board game... like Tic Tac Toe!

If our main is going to interact solely with generic board games, then we need to design base classes with virtual functions that Othello and TicTacToe-specific classes can derive from. If we separate the "game-specific" logic from the "game-agnostic" logic in the othello classes, we come up with the following base classes representing a generic two-player game. Each class notes any functions/behaviors that all game-specific types should implement.

- `GameMove`: represents a single "move" that can be applied to a `GameBoard`.
  - moves can be converted to a string for printing (`operator string()`)
  - moves can compare themselves to other moves for equality (`operator==`).
- `GameBoard`: represents all the state needed to track and play a game.
  - boards can determine a set of valid possible moves for the current game state
  - boards can apply a move that represents a valid possible move
  - boards can undo the last applied move
  - boards can report if they are finished or not according to their own game rules
  - boards can share a history of moves in the order they were applied to the board
  - boards can report whether it is player 1 or 2's turn
  - boards can report a value that indicates whether player 1 or 2 is winning
- `GameView`: prints a `GameBoard` to an `ostream`
  - gets constructed with a shared pointer to a `GameBoard` object appropriate for the derived `View` type
  - has a function `PrintBoard` which does game-specific output of the view's board pointer, that gets called from a global `operator<<`.

[1] You *really* don't want to be working on this during finals week...

- can return a string to represent the "name" of player 1 or 2. `OthelloView` would return "Black" for a player value of 1, whereas `TicTacToeView` would return "X".
- can parse a string into a `GameMove`-derived object appropriate to the game being played

I have given you .h files for these three classes. You do not need to implement any functions for these classes specifically; all the code you will modify or create will be for Othello or TicTacToe classes. **Start by reviewing these files**.

Connect Four

I am providing you with an implementation of Connect Four. You should add the six files I provided to your Project 3 solution, and then review them briefly – the comments within will explain many of the tasks you will have to perform, such as performing down casts of unique pointers, reworking `GetCurrentPlayer`, and implementing `GetPlayerString`. You might also choose to represent the "value" of your future Tic Tac Toe boards in a way similar to my Connect Four implementation.

Note: you may need to change some of the Connect Four code if you named some of your `BoardPosition`/`BoardDirection` methods differently than I did.

Changes to Othello Classes

You will need to update your **OthelloMove.h, OthelloBoard.h, and OthelloView.h** from Project 2 to match the interfaces of the GameX.h files that I have provided to you. Start by adding an inheritance statement to each of your classes, then work on updating the signatures of the methods to *exactly match* the methods from GameBoard/GameMove/GameView.

Example: `OthelloBoard::ApplyMove` used to take `unique_ptr<OthelloMove>` as a parameter, but `GameBoard::ApplyMove` takes `unique_ptr<GameMove>`. `OthelloBoard` must be updated to match this signature.

The changes from Othello-specific types to Game-generic types are necessary because the virtual functions defined in the Game base classes cannot take parameters specific to one individual game type. Since derived classes must match the declaration of virtual functions *exactly as declared in the base class*, Othello-specific classes must use functions which take generic Game types.

Next add any new methods. There are only two:

- `GameBoard::IsFinished`: returns true iff the game is over. In othello, the game is over if 60 moves have been applied, or if two passes have been applied in a row.
- `GameView::GetPlayerString`: given an integer value of 1 or 2, returns a string representing the name given to that player in a specific game. In othello, player 1 is "Black" and 2 is "White".

Finally, change your `mHistory` variable to be of type `vector<unique_ptr<`

Pointers and Down-Casting

Recall the use of pointers with class inheritance: "a base-class pointer can point to a derived-class object, but you can't access the derived-class functions through the base class pointer." Since functions like `ApplyMove` now take base-class pointers to a `unique_ptr<GameMove>`, those functions won't have access to the private Game-specific members like `mPosition` and `mFlips` of `OthelloMove`.

In order to access information specific to a type of game, the first thing you should do in game-specific functions is **cast** any generic Game-type pointers to specific derived-type pointers. In order to be safe, you **must** perform a `dynamic_cast` here, using the raw pointer owned by the `unique_ptr` parameter.

Example:

```
void OthelloBoard::ApplyMove(unique_ptr<GameMove> move) {
   // note:  the parameter is now GameMove, but we need it to only point to
   // OthelloMove, so we dynamic_cast it down and check the result.
```

```
   OthelloMove *m = dynamic_cast<OthelloMove*>(move.get());
   // work with m->mPosition, m->mFlips, etc.
}
```

This rule applies anywhere you have a base-class pointer that actually points to a specific derived type. It also works with references: if you have a variable of type `const GameMove &other` and you need to treat it like an `OthelloMove`, you can cast it as `const OthelloMove &casted = dynamic_cast<const OthelloMove&>(other)`.

Changes to Main

Your main from project 2 creates a few Othello-specific variables: the Board, a View, and (eventually) a Move pointer. These need to be converted to generic Game-type **pointers**, and initialized with appropriate heap-created variables based on the user's choice of game to play. (By the way... you need to ask the user which game they want to play!) Example:

```
shared_ptr<GameBoard> board; // now a base type!!
unique_ptr<GameView> v; // likewise!!
cout << "What game do you want to play?  1) Othello; 2) Tic Tac Toe; 3) Connect Four; 4) Exit";

if (__ user wants to play othello __) {
   auto oBoard = make_shared<OthelloBoard>();
   v = make_unique<OthelloView>(oBoard);
   board = oBoard;
}
else // guess!
```

Playing Again:

When a game is over, OR when the user enters the command "**quit**", the program does not terminate. Instead, you will ask the user for a new game to play, and only terminate the program if they select "Exit". (See above.)

Implementing Tic Tac Toe

Here are some general thoughts:

- You will need to make your own .h files for TicTacToe classes. Base them on Othello's files. These need to derive from appropriate Game-type base classes.
- Much of the work will be copying and tweaking your Lab 2 assignment.
- TicTacToe moves should be similar to Othello moves, except they don't need to remember FlipSets.
- TicTacToe boards require very little work to apply or undo moves.
- You will need to update `mCurrentValue` each time `ApplyMove`/`UndoLastMove` are called. You can make the final call on how to represent a `TicTacToeBoard's` "value", but whatever you choose should be consistent with `OthelloBoard`: when the game is over, a positive value means X won, and a negative value means O won.
- You may write your own functions that aren't present in `OthelloBoard`. Perhaps a helper function to see if anyone has connected 3 in a row...
- As a reminder, a derived class must **re-declare** all virtual functions that it inherits from its parent, but using `override` instead.

Easy to Overlook

Summary of things that are easy to overlook:

- `main` must first ask the user what game they want to play.
- Your program must be able to play full games of othello, Tic Tac Toe, or Connect Four.
- When a game `IsFinished` or when the user types `quit`, the program does not terminate. The user returns to the main menu and asked what game they want to play.
- You must only use modern C++ polymorphic down-casts (`dynamic_cast`), and **no C-style casts.**
- **Test your Tic Tac Toe implementation!** Make sure both players can win, and make sure the game can tie.