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, 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 member variables that all game-specific types should have, as well as any functions/behaviors that all game-specific types should implement. The member variables will be protected; the functions will be pure virtual.

- `GameMove`: represents a single move that can be applied to a `GameBoard`.
  - moves can be assigned to with a string (`operator=(const string &rhs)`)
  - moves can be converted to a string (`operator string()`)
  - moves can compare themselves to other moves for equality (a new function, `bool Equals(const GameMove &other)`)
- `GameBoard`: represents all the state needed to track and play a game.
  - boards have member variables for the history of applied moves (a vector of `GameMove` pointers), the value (indicating who is winning/won), and who the next player is
    - boards have simple inherited functions for returning the value, history, and next player
  - boards can create a `GameMove` pointer appropriate to the `GameBoard`, e.g., `OthelloBoard` can create `OthelloMove`s and `TicTacToeBoard` can create `TicTacToeMove`s
  - boards can determine which spaces are valid possible moves for the current board 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 return a custom string to represent the "name" of player 1 or 2. `OthelloBoard` would return "Black" for a player value of 1, whereas `TicTacToeBoard` would return "X".

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

- `GameView`: prints a `GameBoard` to an `ostream`
  - gets constructed with a pointer to a `GameBoard` object appropriate for the derived `View` type
  - has a single function `PrintBoard` which does game-specific output of the view's board pointer

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.

Changes to Othello Classes

You will need to use **new versions of OthelloMove.h, OthelloBoard.h, and OthelloView.h** that I have provided to you. These files have slight tweaks to the parameters certain methods take. **Any changes to function declarations in a .h file must be matched in your .cpp files**.

In `OthelloBoard`, any Othello-specific return type or parameter to a function has been replaced by the generic Game-type classes. That means `ApplyMove` no longer takes `OthelloMove*`; it takes `GameMove*`. Likewise, `GetPossibleMoves` will take a `vector<GameMove*>` instead of `vector<OthelloMove*>`. You will need to update the functions' definitions in your .cpp files to match the new parameters and return types.

In `OthelloMove`, `operator=` is now virtual and returns a `GameMove&` instead of `OthelloMove&`. There is a new method for testing equality: `bool Equals(const GameMove &other)`, which returns true if the context Move object is equal to the parameter. OthelloMove's operator== is no more.

Finally, `OthelloBoard`/`Move`/`View` must all inherit from the appropriate generic `Game` classes. This gives them access to the protected members of those classes, notably `mValue`, `mHistory`, and `mNextPlayer` from `GameBoard`.

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.

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 `GameMove*`, those functions won't have access to the private Game-specific members like `mRow` and `mCol` 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** use a `dynamic_cast` here, and you **must** check your cast to make sure it succeeded.

Example:

```
void OthelloBoard::ApplyMove(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);
   if (m == nullptr)
      throw OthelloException("Tried to apply a non-OthelloMove.");
   // work with m->mRow, m->mCol, 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)`.

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.
- `TicTacToeBoard` should implement its own version of `GameBoard's GetPlayerString` so it can return "X" or "O" for player 1/-1 instead of "Black" or "White".
- You will need to update `mValue` 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.

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:

```
GameBoard *board; // now a pointer!!
GameView *v; // likewise!!
cout << "What game do you want to play?  1) Othello; 2) Tic Tac Toe; 3) Exit";

if (__ user wants to play othello __) {
   board = new OthelloBoard();
   v = new OthelloView(board);
}
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.)

You must clean up any heap variables when a game ends. In particular, note that `GameView *v` and `GameBoard *board` are initialized on the heap, and need to be deleted before playing a new game. **Also note that a GameBoard variable will have moves in its history that are saved on the heap**. If you simply `delete board`, then the moves in the board's history will be leaked. You **must** modify the `GameBoard` destructor in `GameBoard.h` to go through the `mHistory` vector and free the moves that are in the history.

Extra Credit Opportunity: Connect Four

For 3 extra credit points on your project – not enough to radically change your grade, but enough to make up for turning in the first two projects late – you may implement **Connect Four** instead of TicTacToe. All

of the other comments above still apply to your project, including the changes to main, Othello, and the inheritance rules.

The design of Connect Four is up to you, but you must follow these game rules:

1. The game is played on a 6 rows by 7 columns grid.
2. The two players are Yellow and Red – Yellow makes the first move.
3. The game ends when one player connects four pieces in a row in any of the 8 directions. Hint: your Othello logic can be adapted to this task.
4. The game's value is up to you, so long as it follows the rules used by other games: when the game is over, the value is positive if player 1 wins and negative if player 2 wins.
5. Connect Four can tie (value 0) if the entire board fills up.
6. Although Connect Four has the "walk in a direction looking for pieces" logic in common with Othello, it is more similar to Tic Tac Toe in terms of implementation. (No flip sets, applying and undoing is simple, no passing, etc.)

Easy to Overlook

Summary of things that are easy to overlook:

- `main` must first ask the user what game they want to play.
- `OthelloBoard board` and `OthelloView v` should now be `GameBoard *board` and `GameView *v`, and initialized to point to game-specific variables on the heap.
- 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++ casts in your code (`static_cast` and `dynamic_cast`), and **no C-style casts.**
- **Test your Tic Tac Toe / Connect Four implementations!** Make sure both players can win, and make sure the game can tie.