Project 2:
Moor Othello?

Project due April 24, 2020

Overview

In this project, you will convert and expand your Project 1 to an object-oriented C++ solution following the well-known *model-view-controller* (MVC) software design pattern. In addition to refactoring your original solution to an object-oriented model, you will expand the Othello game with new functionality and options, including the ability to "undo" moves. You will also require the players to input moves that are truly valid and will result in at least one enemy piece flipped. Finally, you will manage dynamic memory requirements through the use of unique pointers and transferring ownership of memory between several functions and objects.

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.

Project Organization

Your project will consist of the following files: OthelloMove.h/cpp, a class for encapsulating a single move on the game board; OthelloBoard.h/cpp, a class representing the state of an Othello game board; OthelloView.h/cpp, a class that prints the current state of a board to the console; and main.cpp, the controller/driver of the game, responsible for handling user input and executing functions on the game model.

You will also need to import your BoardPosition and BoardDirection files from your lab assignment.

Class Overview

Your project will consist of these classes. Note that some classes have some functions that have been implemented for you. You may not change these functions, or add any member variables/functions to the classes.

- OthelloMove: represents a single move on an othello board.
  - This class encapsulates information associated with a single move on the board. Moves are identified by their `BoardPosition`; they do not keep track of whose turn it is/was when the move is/was applied. Moves can report whether they are a Pass or not, and can be converted to string objects for output.
  - OthelloMove objects can be created with two constructors. Most typically, they will be placed on the heap and managed with `unique_ptr` wrappers.
  - To support undo functionality, OthelloMove objects will also maintain a list of pieces that were flipped when the move was applied. This will be detailed below.
- OthelloBoard: collects all information needed to represent the state of the game board.
  - Contains the 8x8 board game array, plus member variables for the current player, move history, and board value. The value is now a **running total**: it is a member variable that is updated every time a new piece is placed on the board or an existing piece is flipped; it **does not and cannot** use a **loop** to walk through the entire board in order to recalculate the board value.
  - Public functions for manipulating the game state: `ApplyMove`, `UndoLastMove`, `GetPossibleMoves`.
  - The board can now report the list of all legal moves through `GetPossibleMoves`, which returns a vector of `OthelloMove` objects that are legal on the current board state for the current player.

  - When applying a move, the board updates its internal state, including saving the applied move in its move history.
  - The board can also undo the most recently applied move, restoring the board state to prior to that move's application.

- OthelloView: handles all input and output of othello-related objects.
  - Has shared ownership of an `OthelloBoard` object in order to access the board's matrix for printing.
  - Overloads operator<< to print a board to the console.
  - Has a method `ParseMove` that takes a string representing an othello move, then constructs a corresponding `OthelloMove` object on the heap and returns ownership of it.

- FlipSet: in order to undo an OthelloMove, the move object must "remember" the set of board positions that switched players as a result of the move's application. To do this, each Move object will record a vector of "flips". A `FlipSet` consists of two member variables: a `BoardDirection` and a count; together, these represent a direction on the board in which the given number of enemy pieces were switch to friendly. Since a `FlipSet` only describes a *single direction of flips* and a move can flip enemy pieces in multiple directions, an `OthelloMove` maintains an entire `vector` of `FlipSet` objects so it can remember the directions and numbers of all enemy pieces flipped by the move. To undo a move, we start at its board position and then, for each FlipSet object, walk the given number of squares in the given direction, restoring the enemy to those positions.

New Functionality

You will re-write your main to not only use your new object-oriented game design, but also support a host of new game options. Instead of the previous "show board, ask for move, verify move, apply move" loop, you will ask the user to input a *command*, and then parse that command to drive the game objects. Your main function then looks like this:

1. **Initialization**: initialize an `OthelloBoard` object in dynamic storage using a shared pointer, which you will share with an `OthelloView` object you construct in automatic storage. You will also need a local variable for the user's command choice (a `string`).
2. **Main loop**: repeatedly do the following:
   (a) **Print** the game board as in Project 1, using your `OthelloView` object. The View should print whose turn it is.
   (b) **Print** a list of all possible moves for the current player. Start by calling `OthelloBoard::GetPossibleMoves`. Print out all of the moves in the vector using `operator<<`.
   (c) **Ask** the user to input a command. Use the `getline` function to read in an entire line of text as the user's input. Perform one of the following commands depending on the choice:
      i. `move` *`(r, c)`*: use your View to parse just the `(r, c)` portion of this string into a Move object. If the parse does not fail, then use your vector of possible moves to look for a move that equals the user's input. If you find one, this means the entered move is valid and should be applied, by transferring ownership of the move to the board's `ApplyMove` method; if you cannot find an equivalent move, the move is invalid and you should inform the user.
      ii. `undo` *`n`*: undo the last *n* moves by repeatedly calling `OthelloBoard::UndoLastMove`. Stop calling `UndoLastMove` if you reach the start of the game. (If asked to `undo 100000`, you should not end up going into "negative" move counts.)
      iii. `showValue`: show the current value of the board by calling `OthelloBoard::GetValue`.
      iv. `showHistory`: show the history of moves applied by the players, with the most recent move shown first. One move per line, each line starting with the color of the player that applied that move. See `OthelloBoard::GetMoveHistory()`.
      v. `quit`: quit the game immediately.

   (d) **Loop** part (c) until the user enters a valid move or undoes a move, then **loop** back to (a).

3. **Quit** the game when the OthelloBoard notes that it is finished (`OthelloBoard::IsFinished`). The board keeps track of passes and reports that it is finished when 2 passes in a row are recorded. Passes are still applied to the board via `ApplyMove` and therefore can be undone with the `undo` command, but now the player can only pass if all other options are exhausted.

Pointer Management

This project used to be a lot harder, because students had to keep track of who "owned" each OthelloMove object that was created on the heap (and a *lot* of Move objects will be created on the heap). Thanks to C++11, this becomes much easier:

1. Anyone who calls `GetPossibleMoves` will receive a vector that owns a sequence of OthelloMove pointers. If that vector goes out of scope or is cleared, it will delete all of the Move objects automatically.
2. `ParseMove` also returns ownership of the Move that it creates. This object will be used to see if the user's selected move is among the valid possible moves. If the move is valid, ownership will be transferred to the OthelloBoard; if it is not valid, it will go out of scope and be deleted.
3. Any move that is applied will be owned by the OthelloBoard's `mHistory` vector. When a move is undone and popped from that vector, it will be automatically deleted. If a board is destroyed, its vector will be destroyed and that will destroy all the moves that it owns.

You just need to call `std::move` appropriately when transferring ownership of `unique_ptr` objects. Piece of cake!

Where to Begin?

I recommend the following order when approaching this project:

1. (This one should be obvious.) Read this entire specification and ask questions about things you don't understand.
2. Download the BeachBoard ZIP file containing starting points for the project.
3. Implement the `OthelloMove` class: add any functions required by the .h file, then write the .cpp file.
4. Implement pieces of the `OthelloBoard` class. There are several functions that must be written in a .cpp file:
   (a) First write the constructor, which initializes the `mBoard` matrix with the initial state of the board, and sets `mValue` and `mCurrentPlayer` to appropriate initial values.
   (b) Write `ApplyMove`, translating your code from Project 1 to work with `OthelloBoard`'s member variables and a `unique_ptr<OthelloMove>` parameter (which has an `mPosition` value instead of being given a row and column separately). Note that the board now keeps track of the current player and should update that during the function.
   (c) You may have trouble meeting the `ApplyMove` line limit unless you switch your "choose a direction" code to use the `CARDINAL_DIRECTIONS` array from your lab assignment, and take advantage of the "operator+" that adds a BoardPosition and a BoardDirection.
5. Implement the `OthelloView` class. Write the `PrintBoard` function, which is a rough translation of your `PrintBoard` from Project 1. Implement the `operator<<` to call the `PrintBoard` method of the `OthelloView` parameter. Write `ParseMove`, using an `istringstream` to read in the move's board position.
6. You can now run the **debugging code** in the main I gave you. This code applies three simple moves to a default board and prints the board after each move. Examine the code and the output to make sure you translated `ApplyMove` correctly. At this point, you have effectively translated your entire Project 1 solution to an object-oriented model, and can work on the new behavior

7. Implement the new behavior for the othello game:
   (a) Augment your code to perform the new behaviors expected of `ApplyMove`: updating the `mValue` member variable to reflect the changed board value, recording the `FlipSets` of the move, and saving the applied move to the board's history vector.
   (b) Write `GetPossibleMoves`, noting the requirements in the .h file for how to order the moves.
   (c) Write an empty function for `UndoLastMove` and save it for later.
8. You are now ready to start your *real* main. Comment out the debugging code. Implement the "`move`" command so you can play the game with your own moves. Implement `showValue` and `showHistory` next.
9. Now go back and work on `UndoLastMove` and the "`undo` *n*" command.
10. Easy :).

Line Limits

Yes, there are line limits.

`OthelloBoard.cpp`:

- Constructor: 5 lines
- `ApplyMove`: 18 lines
- `GetPossibleMoves`: 18 lines
- `UndoLastMove`: 12 lines

`OthelloMove.cpp`:

- `operator==(const OthelloMove &rhs)`: 1 line
- `operator std::string()`: 5 lines

`OthelloView.cpp`:

- `operator<<(ostream &lhs, const OthelloView &rhs)`: 3 lines
- `operator<<(ostream &lhs, const OthelloMove &rhs)`: 1 line

`int main`: 80 lines

Easy to Overlook

Summary of things that are easy to overlook:

- You do not `cout` the `OthelloBoard` itself; you `cout` the `OthelloView` object. You can use `cout` with `OthelloMove` objects.
- The board's history vector lists moves in the order they were applied. When `showHistory` is run, you have to print the moves in reverse order from how they appear in the history vector.
- When printing the history, you must indicate which player applied which move.
- Passes should be treated like any other move by `main`. You can only pass if `pass` is in the vector of possible moves. `pass` will only be in the vector if there are no other possible moves.
- You cannot use any additional loops to calculate the value of a board. Every time you place a piece or flip a piece, you must adjust the value by some small amount. (Think about it... what happens to the value of the board when you place a new black piece vs. flip a white to a black?)
- The game is over when `IsFinished()` returns true.