Project 2:
Moor Othello?

Project due April 21, 2015

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.

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.

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 row and column only; 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/from string objects for output and input.
  - OthelloMove objects will be initialized via the operator(=string) assignment operator, which takes a move in string form and initializes the object's row and column. If the string is not in the correct format, or the row/column requested are out of bounds, the assignment operator will throw an exception. (See section titled "Exceptions".)
  - 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 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`.
- OthelloView: overloads C++ operators to support printing a board to the console.
  - Has friend access to an `OthelloBoard` object in order to access the board's matrix for printing.

  - Overloads operator<< to print a `View` to the console.

- FlipSet: this small private inner class of `OthelloMove` represents a set of "flips" made when a move was applied to a board. A `FlipSet` consists of three member variables: a `rowDelta` and `colDelta` representing the direction of the flips, plus a count of the number of flips made. 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. When undoing a move, you will check the `OthelloMove`'s vector of `FlipSets`, and use the information in each to reverse the move.

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 on the stack, and an `OthelloView` object using a pointer to the board. You will also need local variables for the user's command choice (a `string`) and a `vector` of `OthelloMove` pointers for use with `OthelloBoard::GetPossibleMoves` (below).
2. **Main loop**: repeatedly do the following:
   (a) **Print** the game board as in Project 1, using your `OthelloView` object. Output whose turn it is.
   (b) **Print** a list of all possible moves for the current player. Start by calling `OthelloBoard::GetPossibleMoves`. (This will fill in your local vector object with pointers to all valid `OthelloMove` objects for the current board.) Print out all of the moves in the vector, since the moves were placed on the heap and are now unneeded. Delete the pointers and clear the vector when you are done.
   (c) **Ask** the user to input a command. Use the `getline` function (look it up) 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)`*: create an `OthelloMove` object using `OthelloBoard::CreateMove`. (Note that you get back a pointer to something on the heap.) Using the second half of the input string, use the operator=(string) operator to initialize the OthelloMove object using the row and column from the user's input. If the assignment does not throw an exception, then ask the Board for the list of all possible moves using `OthelloBoard::GetPossibleMoves`. (This fills in the local `vector` object with pointers to possible `OthelloMoves`.) Make sure that among that list of possible moves, there exists an `OthelloMove` object that is "equal" to the move the user suggested, by using the `operator==` function from `OthelloMove`. If true, this means the move is valid and should be applied; if you cannot find an equivalent move, the move is invalid and you should inform the user. Delete the pointers and clear the vector when you are done.
      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 boad 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

You will be passing a lot of pointers in this project, and keeping a clear idea of who owns those pointers will be very important. OthelloMoves in particular will generally be put onto the heap and then passed by pointers; depending on if the move is applied or not, "someone" will be responsible for deleting those moves when they are no longer needed.

A summary of pointer issues regarding OthelloMoves:

- OthelloMove objects are **only created by the** CreateMove **function of** OthelloBoard. You should never ever "new" an OthelloMove, or declare a full OthelloMove object on the stack, except in OthelloBoard::CreateMove.
- Whoever calls CreateMove is responsible for deleting the Move when it is no longer needed, or for passing that responsibility to someone else.
- The OthelloBoard::GetPossibleMoves function fills a vector with new OthelloMove objects. Whoever called the GetPossibleMoves function is responsible for deleting those moves.
- Any move passed to ApplyMove becomes the property of the OthelloBoard object, and will be pushed onto its mHistory vector during the ApplyMove routine. Do not delete a move that you passed into ApplyMove – it is now managed by the board.
- Any move that is undone by OthelloBoard::UndoLastMove will be deleted at the end of that method.

Exceptions

The file OthelloExceptions.h contains a single class called OthelloException which you will use in your code to report certain errors. For now, you will only need to throw an exception in OthelloMove's operator=(const std::string &) if the user's requested move is not in-bounds. That exception should be caught **by reference** in the main command loop for the "move" case. Throw the exception by simply using throw OthelloException("error message goes here") (DO NOT "new" the exception like in Java.)

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 is one function you must implement in the .h file, then implement these functions 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 mNextPlayer to appropriate initial values.
   (b) Write ApplyMove, translating your code from Project 1 to work with OthelloBoard's member variables and an OthelloMove* parameter. Note that the board now keeps track of the current player and should update that during the function.
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.
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: 10 lines
- `ApplyMove`: 30 lines
- `GetPossibleMoves`: 30 lines
- `UndoLastMove`: 22 lines

`OthelloMove.cpp`:

- `operator=(const std::string &)`: 11 lines
- `operator std::string()`: 5 lines

`OthelloView.cpp`:

- `operator<<(ostream &lhs, const OthelloView &rhs)`: 2 lines

`int main`: 80 lines

Easy to Overlook

Summary of things that are easy to overlook:

- `main` has to use the `CreateMove` function of `OthelloBoard` to get a `OthelloMove` pointer for the user's move. It cannot use the `OthelloMove` constructors because they are private.
- You do not `cout` the `OthelloBoard` itself; you `cout` the `OthelloView` object.
- 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.