Project 1:
The Tragedy of Othello

Project due February 28

Overview

In this project, you will develop a simple implementation of the game *othello*, also called *reversi*. You will program the general rules of the game and allow two human players to compete in a text-based version of the game. Your game will keep track of the current player, allow that player to select a move, then apply that move to the game board and output the resulting board. The game ends when both players have to "pass" their turn in succession, signaling that neither can select a valid move and thus the game is over.

Game Rules

See the Wikipedia article titled **Reversi** for an overview of how the game is played. We will use the following specific rules:

1. The game board is an 8x8 grid of squares. Each square can have a single piece that is either black or white on that square, or it can be empty. The **rows** of the grid are numbered 0 through 7, as are the **columns**. Thus the upper left corner is position (0, 0), and the upper right is position (0, 7).
2. At the start of the game, the center four squares are already taken by two black and two white pieces; see the Wikipedia article **Rules** section for a picture.
3. On his/her turn, a player must input a square coordinate for their move, in the format `(r, c)`. Alternatively, the user may choose to *pass* by entering a move of `(-1, -1)`.
4. If the board is full or the player cannot make a valid move, they must pass.
5. **The game ends when both players pass in succession.**
6. Black is the first player to move.
7. When the game is over, you will output the winner by counting the value of the board. For each black piece, the value goes up by 1; for each white piece, it goes down by 1. Thus a value of 0 indicates a tie, a positive value indicates a black win, and a negative indicates a white win. Output the winner, even if the board is not completely filled.

Project Organization

Your project will consist of three files: `main.cpp`, `OthelloGame.cpp`, and `OthelloGame.h`. `OthelloGame.h` will contain declarations for functions needed to update the game state and apply moves; `OthelloGame.cpp` will contain the implementation of those functions (see below). `main.cpp` will be the driver of the program, gathering all user input and displaying output after each move. Main will also keep track of whose turn it is and is responsible for detecting the end of the game.

Designing the Game State

You will need variables for tracking the following information. Be intelligent in your choice of types for these variables – **think long and hard about the most efficient implementation for each**. For example, do not use a double to keep track of whose turn it is. These variables should be declared in your main, **not as global values**.

Variables are needed for:

- The game board: an 8x8 multidimensional array (matrix).

  - A 0 value in the array indicates an empty square on the board. A 1 value in the array indicates a Black piece. A -1 value in the array indicates a White piece.
  - You will need to initialize the array with 0's in every square, then fill in the starting positions for the two players.

- The current player: a data type appropriate to knowing whose turn it is.
- The row and column of a player's move: these will be passed as references to a `GetMove` function.
- End of game: you will need some way to determine when the game is over according to the rules above.

Functions

The following functions will be needed and **must be implemented** for the game. Each will be declared in `OthelloGame.h` (this file is supplied on BeachBoard) and implemented in `OthelloGame.cpp`. You may not write any other functions except for these.

- `PrintBoard`: this function takes the game board array as a parameter and prints it to the screen. First, print a header row of the numbers 0 through 7 with spaces inbetween, showing the indices of each column. Then print one row of the board at a time: start with the row index, then print a **period** if the space is empty; a **B** if the black player has a piece there; a **W** for a white piece. Example for the initial board setup:

```
- 0 1 2 3 4 5 6 7
0 . . . . . . . .
1 . . . . . . . .
2 . . . . . . . .
3 . . . W B . . .
4 . . . B W . . .
5 . . . . . . . .
6 . . . . . . . .
7 . . . . . . . .
```

- `GetMove`: this function uses `cin` to read a line of input from the user representing their move, in the format *`(row, col)`*; then parses the line and returns a row and column from the function. Since the function needs to return two pieces of information, it will need to take **reference** parameters from the main, and then use those to store the information read from `cin`. This function should **not** output anything; the main is responsible for *asking* the user to type in a move.
- `InBounds`: this function takes a row and column, and returns true if the position specified is "in bounds" of the board: that is, if both are at least 0 and less than the size of the board.
- `IsValidMove`: this function takes the game board, and integer values for a row and column that the player would like to move to. This function **only checks** that the destination is in-bounds (use `InBounds`) and does not have a piece already at the location; OR alternatively, if the move is a pass (both row and column are -1). This function **does not** check to make sure the player will actually surround enemy pieces with the move.
- `ApplyMove`: this is the hardest function and contains the bulk of the game logic. Given a game board array, a row, a column, and the current player, this function applies the move requested by the current player by turning the requested square to the player's color, and then searches all eight directions from that square looking to see if there is a run of the opponent's pieces to flip according to the game rules. Because you can't count on the user moving to a specific game location, you must use loops to iterate through all 8 directions and continue moving in each direction looking for a run of enemy pieces.

  You can assume that the given move is a valid selection for the current player. You do not need to validate moves.

  We will go over some of the logic in class, but much of it will be up to you to determine. Start

early, get feedback from me early, and let it "stew" in your head for a while. You won't nail the logic in your first go, so don't put it off to the last minute.

- `GetValue`: given the board, this function returns an integer for the value of the board as defined above. **Your main may only call this function once.**

Line Limits

I am putting a cap on the number of countable lines that each of the following functions can contain. A line is countable if it contains at least 3 characters; in practice, this means that empty lines and lines with just a closing brace } are not countable, but everything else is. You may not declare new functions and move part of your function there in order to reduce your line count. **You may not compact multiple statements onto one line**. You must follow the style guide for variable names while still conforming with the following line limits:

- `main` function: 45 lines
- `PrintBoard`: 15 lines
- `IsValidMove`: 3 lines
- `InBounds`: 2 lines
- `GetMove`: 2 lines
- `ApplyMove`: 20 lines
- `GetValue`: 5 lines

Demoing Your Project

As noted in the syllabus, you will be demoing your project to me **in-person** during the lab period of the course. You may request a demo during any lab period, and I will gladly give you feedback or pass you ahead of the due date depending on the quality of your work. During the demo, I will run your program with inputs designed to test its functionality and grace. I will not tell you what these inputs are ahead of time. Should your code not meet my standard of performance, your demo will be rejected and you will need to arrange for another demonstration after fixing your code. **Your project may also be rejected for egregious violations of the course style guide**.

Important: I can only *guarantee* a single demo for each individual on the day the project is due. If you fail your demo on the due date, then you may not get a second chance that day. You will still need to fix and re-demo your code to get a grade on the project. Remember, you must get a passing grade on each project, and you may resubmit as many times as it takes to pass the demo. The only penalty for failing a demo is potentially losing more points to lateness.

Example Output

User input is in *italics*. These do not cover every possible test case. It is up to you to thoroughly test your program before submitting it.

```
Welcome to Othello!

- 0 1 2 3 4 5 6 7
0 . . . . . . . .
1 . . . . . . . .
2 . . . . . . . .
3 . . . W B . . .
4 . . . B W . . .
5 . . . . . . . .
```

```
6 . . . . . .
7 . . . . . .

Black's turn.  Please choose a move:
(3, 4)
Invalid move.  Please choose a move:
(3, 2)

- 0 1 2 3 4 5 6 7
0 . . . . . .
1 . . . . . .
2 . . . . . .
3 . . B B B . . .
4 . . . B W . . .
5 . . . . . .
6 . . . . . .
7 . . . . . .

White's turn.  Please choose a move:
(2, 2)

- 0 1 2 3 4 5 6 7
0 . . . . . .
1 . . . . . .
2 . . W . . . .
3 . . B W B . . .
4 . . . B W . . .
5 . . . . . .
6 . . . . . .
7 . . . . . .

Black's turn.  Please choose a move:
(-1, -1)

- 0 1 2 3 4 5 6 7
0 . . . . . .
1 . . . . . .
2 . . W . . . .
3 . . B W B . . .
4 . . . B W . . .
5 . . . . . .
6 . . . . . .
7 . . . . . .

White's turn.  Please choose a move:
(-1, -1)

Game over.  We have a tie!
```