Object-Oriented Programming in C++
Assignment 2 (20%)
Board Game Software

Objective

The objective of this assignment is to provide a hands-on experience in applying object-oriented design and C++ implementation principles in building a non-trivial application. A good OO-design is important for this assignment. There is, however, no one “right” design and you have to make various (possibly) compromising design decisions along the way.

You can work on this assignment in a group of up to three persons (recommended). Hand in the code as well as a short report (2-3 pages) where you discuss your main design decisions.

Description

You have accepted a software-engineering job at a newly created startup company called *Brilliant Board Games Inc*. The company specializes in writing board game software. It is important for the company to get a product to the market as soon as possible, so they have decided to concentrate their effort initially on only two-person turn-taking board games, for example, chess-like games.

Task 1

Your first task is to design and implement **base classes** for a generic board game that provide both a well-defined interface and default behaviors as appropriate. Other teams within the company will write different board games by inheriting from your classes.

You may assume the following restrictions on the games to be implemented:

- A game will be a two-player turn-taking game, played on an arbitrary sized rectangular board of squares. You may assume a maximum board size of 9x9.
- All pieces are initially placed on the board in an arrangement determined by the game rules. Each piece belongs to exactly one player. The setup is not necessary symmetrical for both players.
- On a turn a player can move one of his or her pieces from one square to another one, in a patterned determined by the game rules. If another piece, opponent’s or own, is on the destination square it will be removed from the board. Thus, there can only be one piece on any given square at once.

- Legal moves and terminal values depend only on the current game state, not the history of how that state was reached. This, for example, implies that a three-fold-repetition (e.g., as in chess) are not considered.
- The outcome of the game will be a win, loss, or a tie, as determined by the game rules. The games are zero-sum, that is, if one player wins the other loses (or both tie). If a game reaches some preset maximum number of moves the game is declared as a tie.

Think well about your design before you begin the coding. There are both concrete and conceptual items that you may (or may not!) choose to model using classes, including: piece, board, position (or game state), game, etc. Also, read carefully the description of task two below as it provides important information about the functional requirements the base classes should provide. Document your code appropriately using *DoxyGen*.

Task 2

Your team has also been given the task of writing a console-based program capable of playing different games (another team within the company is writing a fancy HTML5-based graphical web-interface that will communicate with your console-based program). Also, to showcase the generality and usability of your design from task one, your team is furthermore required to fully implement three different board games.

The first game you should implement is *Fox and Hounds*:
- https://en.wikipedia.org/wiki/Fox_games#Fox_and_Hounds
- http://www.lutanho.net/play/foxanddogs.html

the second game is *Breakthrough:*
- https://en.wikipedia.org/wiki/Breakthrough_(board_game)

and the third game is s variant of Breakthrough, which we call *Mega-Breakthrough*. Each player now has two types of pawns: the former behaves as in the standard game and the second, so called mega-pawns, can advance two spaces vertically forward (if both squares are empty) in addition to the standard moves. The initial board setup is the same as before, but with mega-pawns on the bottom and top rows.

Your program *must* (minimally) support the commands listed below (you can add more commands if you like). The commands are read from standard input. Make sure the command names and number of arguments are *exactly* as asked for (the HTML team will expect that). *Play close attention to the commands, as they also give hints about the functionality expected from the board games your design must encompass*.

You can create as many .h/.cpp files as needed.

Commands:

- ***list***
  - Outputs the game list, that is, names of supported games (in our case the three above mentioned games).

- ***game*** *n*
  - Instructs the program to play game number *n* in the game list.

- ***start***
  - Start a new match game. The current game state (position) should be the start position of the game being played.

- **legal**
  - Outputs all legal moves in the position (for the player to move).

- ***move*** *from_square to_square*
  - Play a given move (e.g., *move b2 b3*). The current position (game state) will be updated accordingly. An error message "Illegal move" will be displayed if the move specified is illegal (and no game state update).

- ***retract***
  - Retracts the last move played. The current game state will be updated accordingly. If there is no move to retract, the command does nothing.

- ***display***
  - Outputs the current game state in a standard format; '.' means an empty square, but other letters indicate pieces (upper-case for the first player, and lower-case for the second). Columns are indexed by a letter in an increasing alphabetic order, whereas rows are indexed by the row number in a decreasing order. The bottom-leftmost square has the coordinate *a1* and the top-rightmost one *h8* (on an 8x8 board). The number in the line after the board shows the player to move (0 for player one, 1 for player two), and the last line shows the piece-counts for player 0 and 1, respectively.

```
8 mmmmmmmm
7 pppppppp
6 ........
5 ........
4 ........
3 ........
2 PPPPPPPP
1 MMMMMMMM
  abcdefgh
0
16 16
```

- ***evaluate***
    - Display the evaluation value of the current board state (from the perspective of the side to move) using the appropriate evaluation function decided by the difficulty level.

- ***go***
    - The computer plays for the side-to-move using the current difficulty level. The program should output the move it makes as:

      move from_square to_square

      If after making the move the game is over, the outcome of the game should be displayed (i.e., "*Player to move wins*", "*Player to move ties*", or "*Player to move loses*" depending on the outcome).

- ***level*** *[random | easy | medium | hard]*
    - random: play a random legal move.
    - easy: play the best available move using a one-ply look-a-head and piece-count evaluation.
    - medium: play the best available move using a three-ply minimax search and piece-count evaluation.
    - hard: play the best available move using a three-ply minimax search and an advanced evaluation.

- **debug**
    - Toggle debug mode on or off. In debug-mode your software can display to standard output any additional information it wants. However, when debug mode is off **only** the asked for output should be displayed. By default, the debug should be off.

- **quit**
    - Quit the program.

Good luck! ☺