Imperial College London
Department of Computing

MSc C++ Programming – Assessed Exercise No. 3

Due: See CATE[1]

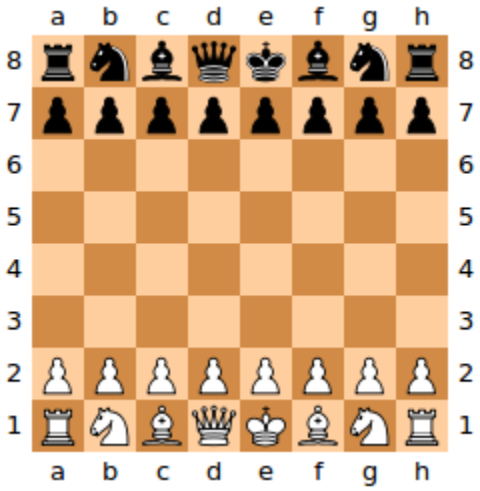
The Exercise

You are required to write a program with the name `chess`, to simulate and manage chess games. Your program should accept moves in the format `(source_square, destination_square)`, where a square is referred to using a letter in the range `A-H` and a number in the range `1-8`. It should validate each move and keep track of the state of the game, detecting when the game is over and producing appropriate output to the user.

Write appropriate class definitions in C++ which correctly interface with the main program given in the file `ChessMain.cpp`, and simulate the game found therein. This main program also contains code which tries to give various kinds of erroneous input which your solution should handle as indicated below in the sample output.

Chess Rules (A Reminder)[2]

The chess board is initially set up as per the following diagram:



Each row (numbered 1 - 8) is called a *rank*, and each column (labelled A to H) is called a *file*. White moves first, and thereafter players take it in turns to make moves. Pieces are moved to either an unoccupied square or one occupied by an opponent's piece, which is captured and removed from play. Each piece moves according to the following rules:

[1] `cate.doc.ic.ac.uk`
[2] The information in this section is taken from the Wikipedia page on Chess: `http://en.wikipedia.org/wiki/Chess`

The **king** moves one square in any direction.

The **rook** (or **castle**) can move any number of squares along any rank or file, but may not leap over other pieces.

The **bishop** can move any number of squares diagonally, but may not leap over other pieces.

The **queen** combines the power of the rook and bishop and can move any number of squares along rank, file, or diagonal, but it may not leap over other pieces.

The **knight** moves to any of the closest squares that are not on the same rank, file, or diagonal, thus the move forms an "L"-shape: two squares vertically and one square horizontally, or two squares horizontally and one square vertically. The knight is the only piece that can leap over other pieces.

The **pawn** may move forward to the unoccupied square immediately in front of it on the same file; or on its first move it may advance two squares along the same file provided both squares are unoccupied; or it may move to a square occupied by an opponent's piece which is diagonally in front of it on an adjacent file, capturing that piece.

When the king is under immediate attack by one or more of the opponent's pieces, it is said to be *in check*. A response to a check is a legal move if it results in a position where the king is no longer under direct attack (i.e., not in check). This can involve capturing the checking piece; interposing a piece between the checking piece and the king (which is possible only if the attacking piece is not a knight and there is a square between it and the king); or moving the king to a square where it is not under attack. It is illegal for a player to make a move that would put or leave his own king in check.

The object of the game is to checkmate the opponent; this occurs when the opponent's king is in check, and there is no legal way to remove it from attack. When checkmate occurs, the game is over and the player whose king is checkmated loses. The game may also end in a *stalemate*: this is where the player whose turn it is to move is not in check but has no legal move that they can make.

N.B. You are not required to implement any of the the chess rules of *castling*, *en passant*, or *pawn promotion*.

The Game To Simulate

The specific game which is simulated in the `main` function found in `ChessMain.cpp` is a short game played between Alexander Alekhine, and his opponent Vasic in 1931. In algebraic chess notation, the game played out as follows:

1. e4 e6
2. d4 d5
3. Nc3 Bb4
4. Bd3 Bxc3+

5. bxc3 h6
6. Ba3 Nd7
7. Qe2 dxe4
8. Bxe4 Ngf6
9. Bd3 b6
10. Qxe6+ fxe6
11. Bg6#

The main program also tests that your solution can detect and handle erroneous input (e.g. trying to move a non-existent piece, trying to move a piece of the wrong colour, making illegal moves, etc.).

The Output

When run against the provided main function, your program should produce the following output.

```
============================
Testing the Chess Engine
============================

A new chess game is started!

It is not Black's turn to move!

There is no piece at position D4!

White's Pawn moves from D2 to D4

Black's Bishop cannot move to B4!

============================
Alekhine vs. Vasic (1931)
============================

A new chess game is started!

White's Pawn moves from E2 to E4
Black's Pawn moves from E7 to E6

White's Pawn moves from D2 to D4
Black's Pawn moves from D7 to D5

White's Knight moves from B1 to C3
Black's Bishop moves from F8 to B4

White's Bishop moves from F1 to D3
Black's Bishop moves from B4 to C3 taking White's Knight
White is in check
```

```
White's Pawn moves from B2 to C3 taking Black's Bishop
Black's Pawn moves from H7 to H6

White's Bishop moves from C1 to A3
Black's Knight moves from B8 to D7

White's Queen moves from D1 to E2
Black's Pawn moves from D5 to E4 taking White's Pawn

White's Bishop moves from D3 to E4 taking Black's Pawn
Black's Knight moves from G8 to F6

White's Bishop moves from E4 to D3
Black's Pawn moves from B7 to B6

White's Queen moves from E2 to E6 taking Black's Pawn
Black is in check
Black's Pawn moves from F7 to E6 taking White's Queen

White's Bishop moves from D3 to G6
Black is in checkmate
```

Approach

A suitable way to approach the exercise could be as follows:

1. Create classes for each different kind of chess piece - each class will be responsible for determining which moves are valid for itself.
2. Create an *abstract* superclass for chess pieces, which provides the interface through which the `ChessBoard` class interacts with its pieces.
3. If you wish, you may use the C++ STL (Standard Template Library), and any other libraries that you think will be useful (e.g. you could use the `map` class to store the chess pieces on the board, indexed by their positions).

You might like to ask yourself the following questions:

- What information does the `ChessBoard` class require from each if its pieces to be able to validate the moves that are submitted to it?
- What information does a chess piece need to know in order to determine which moves it can make?
- What conditions must be checked after each piece has moved?