The Text based Chess Game

2017-11-20
Junpyo Hong (jp7.hong@gmail.com)

- Table of Contents -

1. Software requirements
2. Software architecture design
3. Implementation
4. Verification
5. How to build
6. References

This document contains a description of the simple text-based chess game for two players.

you can see how I designed the software architecture and data structures, and how I implemented functions to satisfy software requirements.

1. Software requirements

The simple chess game to be implemented which has the following requirements:

- Managing a game for two players
- This chess is played on the same 64-square board as regular chess.
- The only pieces are kings, pawns, rooks, and bishops, No knights or queens.
- En Passant, castling, and promotion are not valid moves.
- When pawns reach the end of the board, they cannot move any further.
- Edge case conditions for game termination such as the threefold repetition and fifty-move rule do not apply.
- The game is only ended when a player is in checkmate, or has no legal moves left. In both cases, they lose. This chess is a game of psychological endurance and some games will never terminate.

- In all cases where the rules leave ambiguity, the game mechanics operate in the same way as they do in old-chess.
- Initial board layout is as follows:

```
  -----------------------
8|BR .. BB .. BK BB .. BR|
7|BP BP BP BP BP BP BP BP|
6|.. .. .. .. .. .. .. ..|
5|.. .. .. .. .. .. .. ..|
4|.. .. .. .. .. .. .. ..|
3|.. .. .. .. .. .. .. ..|
2|WP WP WP WP WP WP WP WP|
1|WR .. WB .. WK WB .. WR|
  -----------------------
  A  B  C  D  E  F  G  H
```

  Each alphabet means that R (Rook), B (Bishop), P (Pawn), K (King), W (White), and B (Black).

- Output is as follows:

```
==== BEGIN GAME OUTPUT ====
  -----------------------
8|BP BP BP BP .. BP BP BP|
7|.. .. .. .. .. .. .. ..|
6|.. .. .. .. .. .. .. ..|
5|.. .. BB .. BP .. .. ..|
4|.. .. .. .. WP .. .. ..|
3|.. .. .. .. .. WP .. ..|
2|WP WP WP WP .. .. WP WP|
1|WR .. WB .. WK WB .. WR|
  -----------------------
  A  B  C  D  E  F  G  H
In check: N
Move: W
====  END GAME OUTPUT  ====
```

  where 'In check' means that a king can be captured by enemy troops.

- Inputs are specified as source, destination coordinate pairs, followed by the system newline. If a move is invalid, no output is produced; the game should immediately prompt the player for another input. Players take turns playing chess using the same terminal.

```
==== BEGIN GAME INPUT ====
A2,A4
====  END GAME INPUT  ====
```

- When the game is won, lost, or tied, it should print a single line and exit:

```
==== BEGIN GAME OUTPUT ====
Winnner: W
====  END GAME OUTPUT  ====
```

  where valid winners are 'W' - White, 'B' - Black, or 'D' - Draw.

2. Software architecture design

A. Functionality (use case diagram)

The chess game is for two players and should have functions as follows:

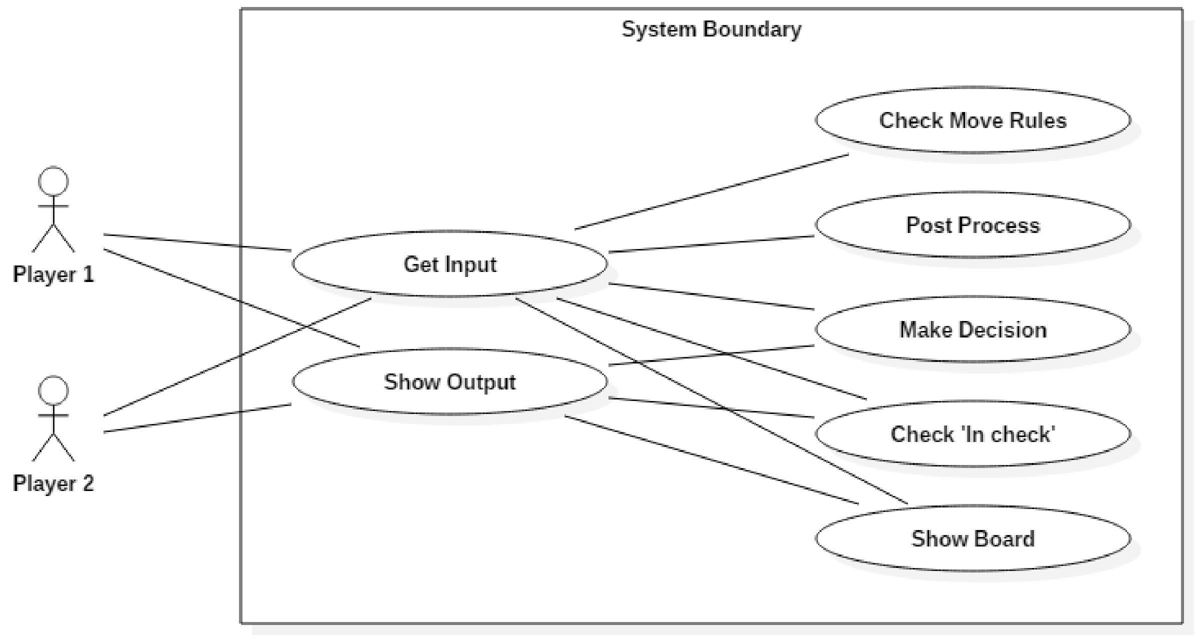


Fig. 1. Use case diagram for the chess game.

The functions in the *System Boundary* are as follows:

Use Case	Description
Get Input	• gets a player's input (source and destination positions).
Check Move Rule	• checks if a player entered a correct destination position according to the chess move rule.
Post Process	• removes a enemy if it is captured. • moves the player's piece. (updates position)
Make Decision	• determines the game result. (winner, draw, or unfinished state)
Check 'Check in'	• checks if 'In check' state has happed.
Show Board	• updates chessboard information according to states of pieces. • shows the chessboard
Show Output	• show result (winner, draw, or chessboard)

B. Date structure (class diagram)

Fig. 2 shows overall class information of the chess game. In the figure, main classes are *CChessBoard* and *CChessPiece*.

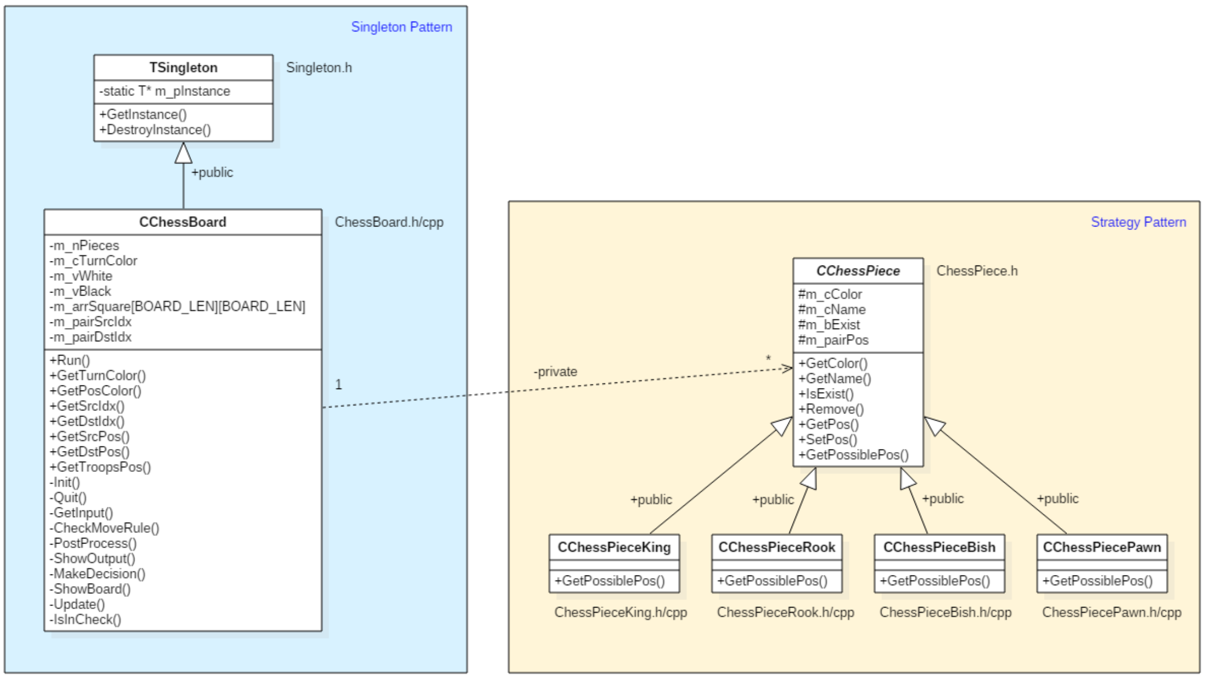


Fig. 2. Class diagram of the chess game.

The *CChessBoard* class has attributes for the interaction with players, managing white/black pieces, and operations for a game management. The implemented chess game doesn't support multiple games at the same time. It means that only one game is available in the program. For this reason, the *ChessBoard* class is derived from *TSingleton* template class which makes exactly one instance across the system. The singleton pattern is generalized to systems that operate more efficiently when only one object exists[1]. Also it enables to get the *CChessBoard* class instance pointer by calling CChessBoard::GetInstance() member function from *CChessPieceXXXX* classes which are derived from *CChessPiece* class.

There are four different chess piece types in this chess game. Those piece types have common attributes such as color, name, existence, and position. Therefore it is suitable to use strategy pattern[2]. The strategy pattern enables selecting a proper algorithm at runtime and encapsulating each algorithm. By using the strategy pattern, *CChessBoard* class can access each chess piece classes by using abstract class pointer of *CChessPiece*. Each *CChessPieceXXXX* class has member function GetPossiblePos() of the same name. GetPossiblePos() member function of each *CChessPieceXXXX* has a role to find possible location according to the move rule of chess piece types.

C. Work flow (sequence diagram)

Fig. 3 shows the interaction between *CChessBoard*, *CChessPieceXXXX* and *CChessYYYY* to manage a game operation. The CChessPieceXXXX means that the chess piece type which is selected by the current player and *CChessYYYY* means that the chess piece type which can be captured by CChessPieceXXXX. The *CChessBoard* class accesses member functions of CChessPieceXXXX and CChessPieceYYYY classes to get or set attributes.

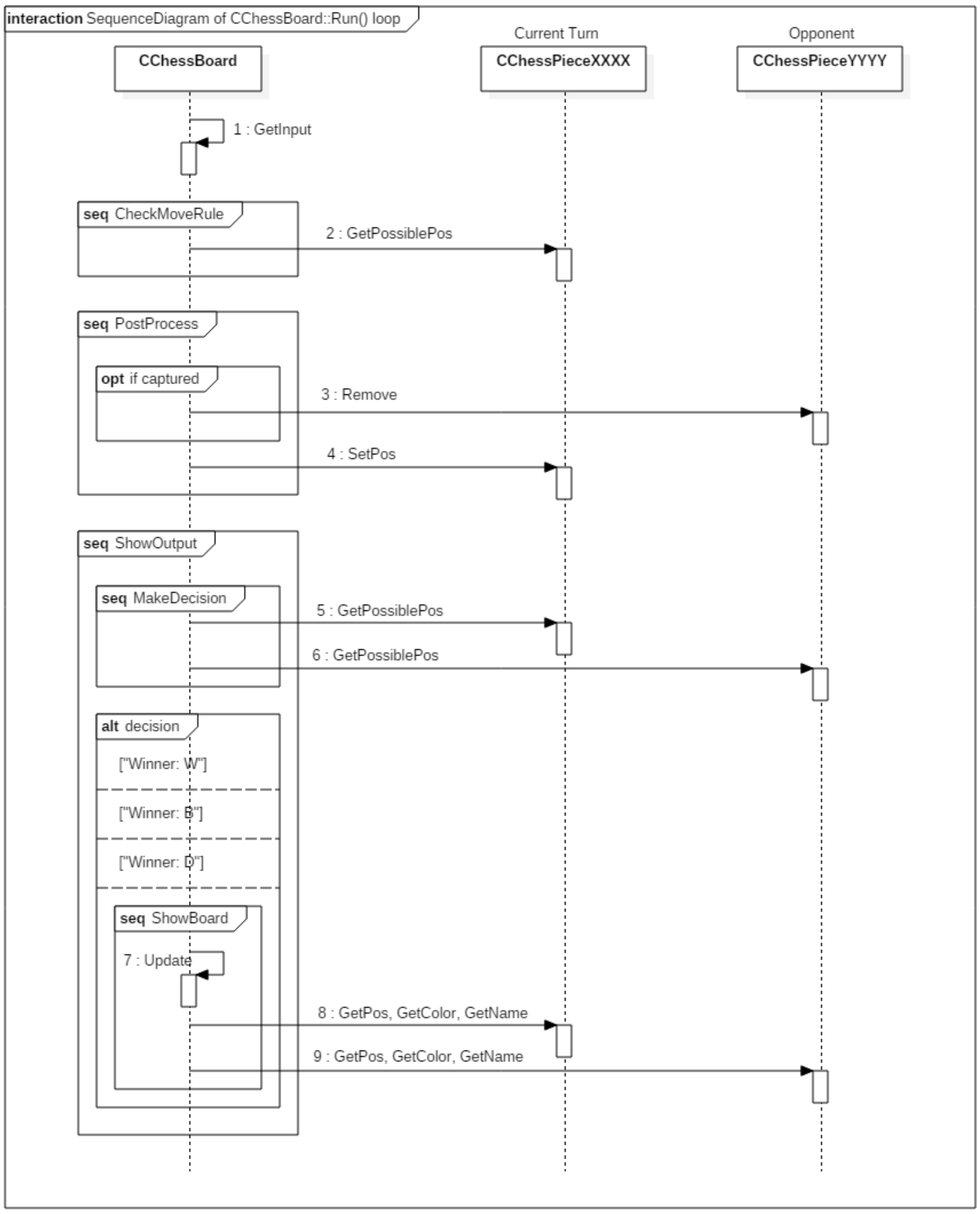


Fig. 3. Sequence diagam of the interaction between *CChessBoard* and *CChessPiece* classes.

3. Implementation

CChessPieceXXXX::GetPossiblePos() function has a role to collect possible positions to move according to the move rule of chess game.

A. Move rule - King

The king piece has following move rules[3]:

(a) The king moves exactly one square horizontally, vertically, or diagonally.

(b) The king cannot move where the square is already occupied by a friendly piece.

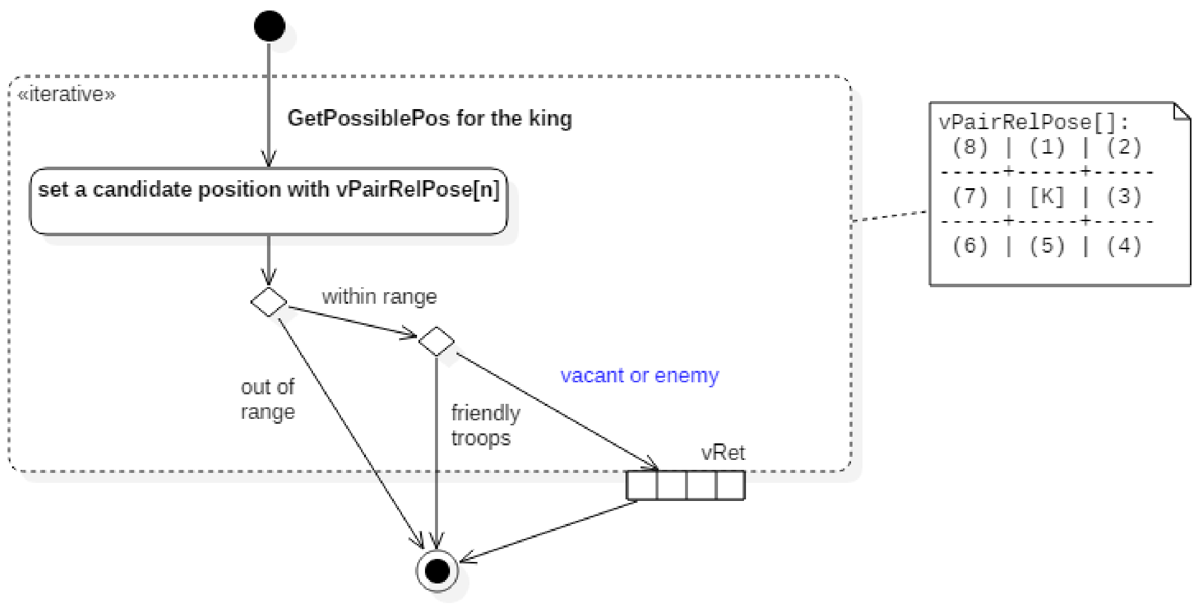
(c) The king captures an enemy piece by moving onto its square.

(d) The king cannot move where it would place itself in check.

(e) The king cannot capture the opposing king.

In this program, it doesn't consider the rules (d), (e), and other exceptional rules.

Fig. 4 shows the algorithm to collect possible positions to move for the king piece.


«iterative»
GetPossiblePos for the king
set a candidate position with vPairRelPose[n]
within range
out of range
friendly troops
vacant or enemy
vRet

```
vPairRelPose[]:
 (8) | (1) | (2)
-----+-----+-----
 (7) | [K] | (3)
-----+-----+-----
 (6) | (5) | (4)
```

Fig. 4. The algorithm to collect possible positions to move for the king piece.

B. Move rule – Rook

A rook piece has following move rules[4]:

(a) A rook moves any number of vacant squares in a horizontal or vertical direction.

(b) A rook cannot move where the square is already occupied by a friendly piece.

(c) A rook can capture an enemy piece by moving onto its square.

(d) A rook cannot leap over other pieces.

Fig. 5 shows the algorithm to collect possible positions to move for a rook.

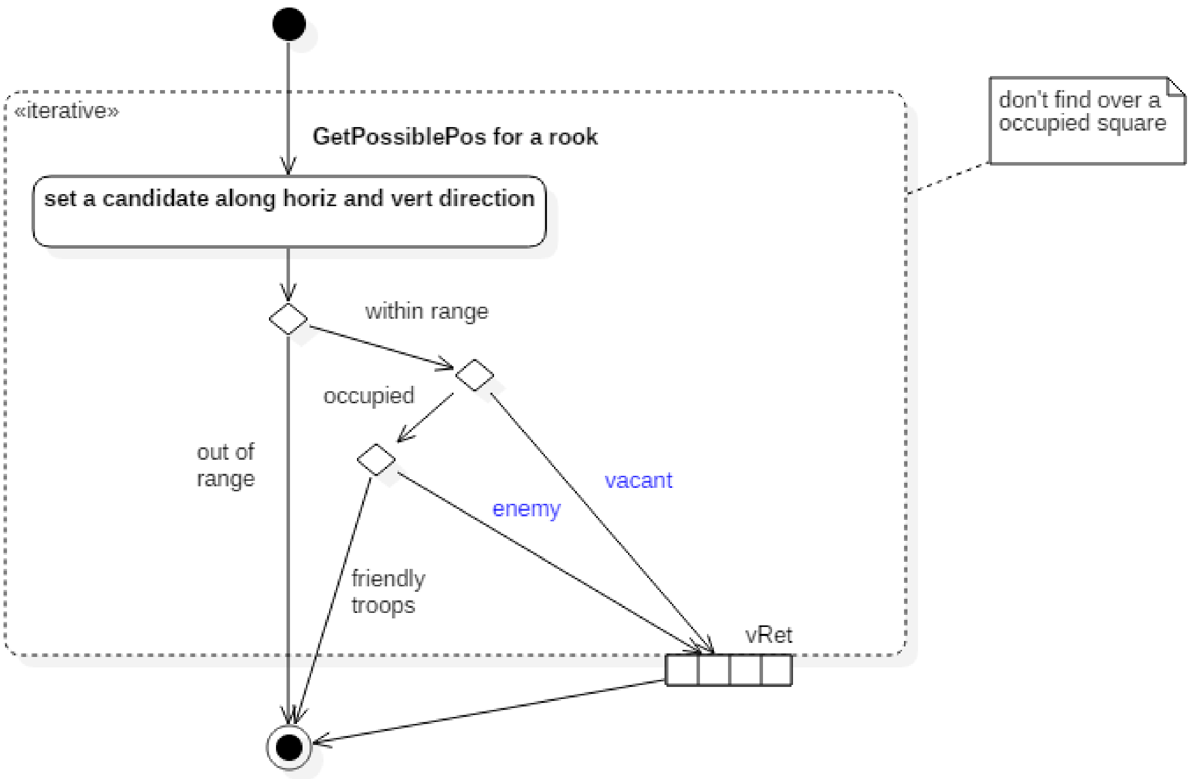

«iterative»
GetPossiblePos for a rook
set a candidate along horiz and vert direction
within range
occupied
out of range
vacant
enemy
friendly troops
vRet
don't find over a occupied square

Fig. 5. The algorithm to collect possible positions to move for a rook piece.

C. Move rule – Bishop

A bishop piece has following move rules[5]:

(a) A bishop moves any number of vacant squares in any diagonal direction.

(b) A bishop cannot move where the square is already occupied by a friendly piece.

(c) A bishop can capture an enemy piece by moving onto its square.

(d) A bishop cannot leap over other pieces.

Fig. 6 shows the algorithm to collect possible positions to move for a bishop.

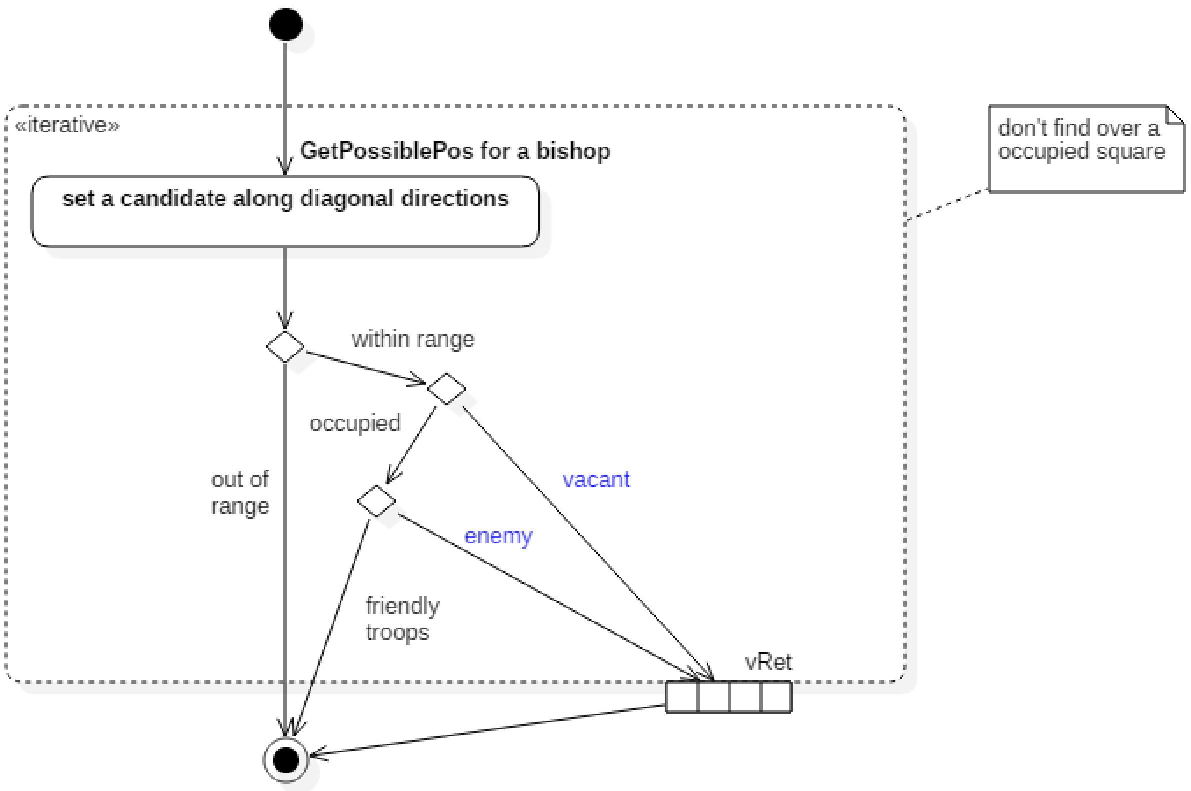

«iterative»
GetPossiblePos for a bishop
set a candidate along diagonal directions
don't find over a occupied square
within range
occupied
out of range
vacant
enemy
friendly troops
vRet

Fig. 6. The algorithm to collect possible positions to move for a bishop piece.

D. Move rule – Pawn

A pawn piece has following move rules[6]:

(a) A pawn moves straight forward one square where there's no piece.

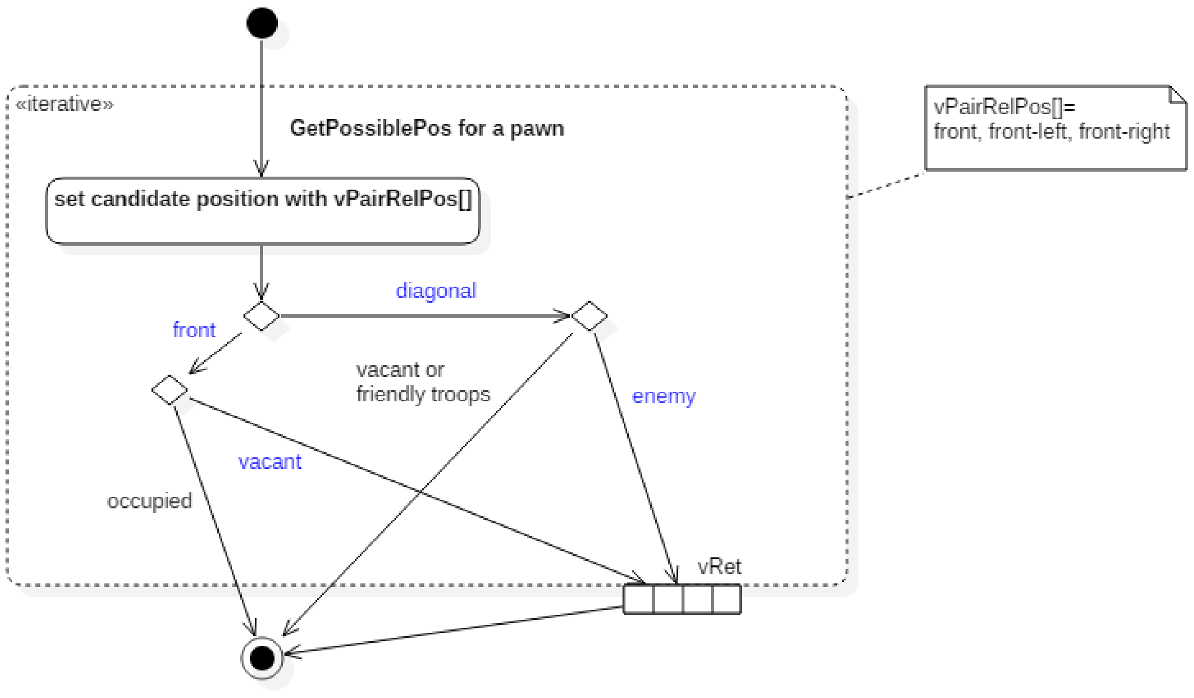
(b) A pawn cannot move backward or lateral side.

(c) A pawn can capture an opponent's piece on a square diagonally.

(d) On first move, a pawn can advance two squares along the same file, provided both square are unoccupied.

In this program, it doesn't consider the rule (d) and other exceptional rules.

Fig. 7 shows the algorithm to collect possible positions to move for a pawn.



«iterative»
GetPossiblePos for a pawn
set candidate position with vPairRelPos[]
vPairRelPos[]=
front, front-left, front-right
diagonal
front
vacant or
friendly troops
enemy
vacant
occupied
vRet

Fig. 7. The algorithm to collect possible positions to move for a pawn piece.

E. Decision of 'In check' state

The 'check' is a condition that occurs when a player's king is under threat of capture on their opponent's next turn[7].

To check whether 'In check' state has happened, it needs to find all possible positions of opponent's pieces. If the king's position is equal to one of opponent's possible position to move, it is 'In check' state.

In this game, a player can move intentionally the king to the opponent's attackable position. Therefore it is necessary to find out 'In check' state of both kings. It means that it needs to investigate all possible positions of all chess pieces except two kings and to compare those positions with the opponent's king position.

F. Decision of winner/draw

The conditions in which the game ends are when a winner occurs or a draw is made.

- Finding a winner

  It is simple to figure out who is a winner. It only needs to check which colored king is missing.

- Judgement of draw

  There are several ways games can end in a draw (stalemate, insufficient materials, threefold repetition of position, fifty-move rule, fivefold repetition of position, draw on time, draw by agreement). Only 'stalemate' is considered in this game.

  When the king has no place to avoid and there's no piece to move, it becomes the stalemate state[8][9]. Therefore, it needs two comparisons. One is to compare possible positions of the king and possible positions of all enemy pieces. The other is to compare possible positions of other pieces.

4. Verification

A. Move rule – King

The king can move exactly one square horizontally, vertically, or diagonally.

```
==== BEGIN GAME OUTPUT ====             ==== BEGIN GAME OUTPUT ====
  -----------------------                 -----------------------
8|BR .. BB .. BK BB .. BR|              8|BR .. BB .. BK BB .. BR|
7|BP BP BP BP BP BP BP BP|              7|BP BP BP BP BP BP BP BP|
6|.. .. .. .. .. .. .. ..|              6|.. .. .. .. .. .. .. ..|
5|.. .. .. .. .. .. .. ..|              5|.. .. .. .. .. .. .. ..|
4|.. .. .. .. .. .. .. ..|              4|.. .. .. .. .. .. .. ..|
3|.. .. .. .. .. .. .. ..|              3|.. .. .. .. .. .. .. ..|
2|WP WP WP WP WP WP WP WP|              2|WP WP WP WP WP WP WP WP|
1|WR .. WB .. WK WB .. WR|      ➔       1|WR .. WB WK .. WB .. WR|
  -----------------------                 -----------------------
  A  B  C  D  E  F  G  H                  A  B  C  D  E  F  G  H
In check: N                             In check: N
Next move: W                            Next move: B
====  END GAME OUTPUT  ====             ====  END GAME OUTPUT  ====

==== BEGIN GAME INPUT ====              ==== BEGIN GAME INPUT ====
E1,D1
====  END GAME INPUT  ====
```

The king can move exactly one square horizontally, vertically, or diagonally.

```
==== BEGIN GAME OUTPUT ====             ==== BEGIN GAME OUTPUT ====
  -----------------------                 -----------------------
8|.. .. BB .. BK BB .. BR|              8|.. .. BB .. BK BB .. BR|
7|.. .. BP BP BP BP BP ..|              7|.. .. BP BP BP BP BP ..|
6|.. BP .. .. .. .. .. BP|              6|.. BP .. .. .. .. .. BP|
5|.. .. .. .. WP .. .. ..|              5|.. .. .. .. WP .. .. ..|
4|BP .. .. .. .. .. .. ..|              4|BP .. .. .. .. .. .. ..|
3|WP .. WP .. WK .. .. ..|              3|WP .. WP .. .. .. .. ..|
2|.. WP .. .. .. WP WP WP|              2|.. WP .. .. WK WP WP WP|
1|WR .. WB .. .. WB .. WR|      ➔       1|WR .. WB .. .. WB .. WR|
  -----------------------                 -----------------------
  A  B  C  D  E  F  G  H                  A  B  C  D  E  F  G  H
In check: N                             In check: N
Next move: W                            Next move: B
====  END GAME OUTPUT  ====             ====  END GAME OUTPUT  ====

==== BEGIN GAME INPUT ====              ==== BEGIN GAME INPUT ====
E3,E2
====  END GAME INPUT  ====
```

The king cannot move where the square is already occupied by a friendly piece.

```
==== BEGIN GAME OUTPUT ====                ==== BEGIN GAME OUTPUT ====
  -----------------------                    -----------------------
8|BR .. BB .. BK BB .. BR|                 8|BR .. BB .. BK BB .. BR|
7|BP BP BP BP BP BP BP BP|                 7|BP BP BP BP BP BP BP BP|
6|.. .. .. .. .. .. .. ..|                 6|.. .. .. .. .. .. .. ..|
5|.. .. .. .. .. .. .. ..|                 5|.. .. .. .. .. .. .. ..|
4|.. .. .. .. .. .. .. ..|                 4|.. .. .. .. .. .. .. ..|
3|.. .. .. .. .. .. .. ..|                 3|.. .. .. .. .. .. .. ..|
2|WP WP WP WP WP WP WP WP|                 2|WP WP WP WP WP WP WP WP|
1|WR .. WB WK .. WB .. WR|        ➔        1|WR .. WB WK .. WB .. WR|
  -----------------------                    -----------------------
  A  B  C  D  E  F  G  H                     A  B  C  D  E  F  G  H
In check: N                                In check: N
Next move: B                               Next move: B
====  END GAME OUTPUT  ====                ====  END GAME OUTPUT  ====

==== BEGIN GAME INPUT ====                 ==== BEGIN GAME INPUT ====
D1,D2
====  END GAME INPUT  ====
```

The king can capture an enemy piece by moving onto its square.

```
==== BEGIN GAME OUTPUT ====                ==== BEGIN GAME OUTPUT ====
  -----------------------                    -----------------------
8|.. .. BB .. BK BB .. BR|                 8|.. .. BB .. BK BB .. BR|
7|.. BP BP BP BP BP BP ..|                 7|.. BP BP BP BP BP BP ..|
6|.. .. .. .. .. .. .. BP|                 6|.. .. .. .. .. .. .. BP|
5|.. .. .. .. WP .. .. ..|                 5|.. .. .. .. WP .. .. ..|
4|BP .. .. .. .. .. .. ..|                 4|BP .. .. .. .. .. .. ..|
3|WP .. WP .. BR .. .. ..|                 3|WP .. WP .. WK .. .. ..|
2|.. WP .. WK .. WP WP WP|                 2|.. WP .. .. .. WP WP WP|
1|WR .. WB .. .. WB .. WR|        ➔        1|WR .. WB .. .. WB .. WR|
  -----------------------                    -----------------------
  A  B  C  D  E  F  G  H                     A  B  C  D  E  F  G  H
In check: N                                In check: N
Next move: W                               Next move: B
====  END GAME OUTPUT  ====                ====  END GAME OUTPUT  ====

==== BEGIN GAME INPUT ====                 ==== BEGIN GAME INPUT ====
D2,E3
====  END GAME INPUT  ====
```

B. Move rule – Rook

A rook can move any number of vacant squares in a horizontal or vertical direction.

```
==== BEGIN GAME OUTPUT ====
  -----------------------
8|.. .. BB .. BK BB .. BR|
7|.. .. BP BP BP BP BP ..|
6|.. BP .. .. .. .. .. ..|
5|.. .. .. .. WP .. .. BP|
4|BP .. .. .. .. .. .. ..|
3|WP .. WP .. .. .. .. WP|
2|.. WP .. .. WK WP WP ..|
1|WR .. WB .. .. WB .. WR|
  -----------------------
  A  B  C  D  E  F  G  H
In check: N
Next move: B
====  END GAME OUTPUT  ====

==== BEGIN GAME INPUT ====
H8,H6
====  END GAME INPUT  ====
```

➔

```
==== BEGIN GAME OUTPUT ====
  -----------------------
8|.. .. BB .. BK BB .. ..|
7|.. .. BP BP BP BP BP ..|
6|.. BP .. .. .. .. .. BR|
5|.. .. .. .. WP .. .. BP|
4|BP .. .. .. .. .. .. ..|
3|WP .. WP .. .. .. .. WP|
2|.. WP .. .. WK WP WP ..|
1|WR .. WB .. .. WB .. WR|
  -----------------------
  A  B  C  D  E  F  G  H
In check: N
Next move: W
====  END GAME OUTPUT  ====

==== BEGIN GAME INPUT ====
```

A rook cannot move where the square is already occupied by a friendly piece.

```
==== BEGIN GAME OUTPUT ====
  -----------------------
8|.. .. BB .. BK BB .. ..|
7|.. .. BP BP BP BP BP ..|
6|.. BP .. .. .. .. .. BR|
5|.. .. .. .. WP .. .. BP|
4|BP .. .. .. .. .. .. ..|
3|WP .. WP .. .. .. .. WP|
2|.. WP .. .. WK WP WP ..|
1|WR .. WB .. .. WB .. WR|
  -----------------------
  A  B  C  D  E  F  G  H
In check: N
Next move: W
====  END GAME OUTPUT  ====

==== BEGIN GAME INPUT ====
H1,F1
====  END GAME INPUT  ====
```

➔

```
==== BEGIN GAME OUTPUT ====
  -----------------------
8|.. .. BB .. BK BB .. ..|
7|.. .. BP BP BP BP BP ..|
6|.. BP .. .. .. .. .. BR|
5|.. .. .. .. WP .. .. BP|
4|BP .. .. .. .. .. .. ..|
3|WP .. WP .. .. .. .. WP|
2|.. WP .. .. WK WP WP ..|
1|WR .. WB .. .. WB .. WR|
  -----------------------
  A  B  C  D  E  F  G  H
In check: N
Next move: W
====  END GAME OUTPUT  ====

==== BEGIN GAME INPUT ====
```

A rook can capture an enemy piece by moving onto its square.

```
==== BEGIN GAME OUTPUT ====            ==== BEGIN GAME OUTPUT ====
  -----------------------                -----------------------
8|.. .. BB .. BK BB .. ..|             8|.. .. BB .. BK BB .. ..|
7|.. .. BP .. BP BP BP ..|             7|.. .. BP .. BP BP BP ..|
6|.. BP .. WP .. .. .. BR|             6|.. BP .. BR .. .. .. ..|
5|.. .. .. .. .. .. .. BP|             5|.. .. .. .. .. .. .. BP|
4|BP .. .. .. .. .. .. ..|             4|BP .. .. .. .. .. .. ..|
3|WP WP WP .. .. .. .. WP|             3|WP WP WP .. .. .. .. WP|
2|.. .. .. .. WK WP WP ..|             2|.. .. .. .. WK WP WP ..|
1|WR .. WB .. .. WB .. WR|      ➔      1|WR .. WB .. .. WB .. WR|
  -----------------------                -----------------------
  A  B  C  D  E  F  G  H                 A  B  C  D  E  F  G  H
In check: N                            In check: N
Next move: B                           Next move: W
====  END GAME OUTPUT  ====            ====  END GAME OUTPUT  ====

==== BEGIN GAME INPUT ====             ==== BEGIN GAME INPUT ====
H6,D6
====  END GAME INPUT  ====
```

A rook cannot leap over other pieces.

```
==== BEGIN GAME OUTPUT ====            ==== BEGIN GAME OUTPUT ====
  -----------------------                -----------------------
8|.. .. BB .. BK BB .. ..|             8|.. .. BB .. BK BB .. ..|
7|.. .. BP .. BP BP BP ..|             7|.. .. BP .. BP BP BP ..|
6|.. BP .. BR .. .. .. ..|             6|.. BP .. BR .. .. .. ..|
5|.. .. .. .. .. .. .. BP|             5|.. .. .. .. .. .. .. BP|
4|BP .. .. .. .. .. .. ..|             4|BP .. .. .. .. .. .. ..|
3|WP WP WP .. .. .. .. WP|             3|WP WP WP .. .. .. .. WP|
2|.. .. .. .. WK WP WP ..|             2|.. .. .. .. WK WP WP ..|
1|WR .. WB .. .. WB .. WR|      ➔      1|WR .. WB .. .. WB .. WR|
  -----------------------                -----------------------
  A  B  C  D  E  F  G  H                 A  B  C  D  E  F  G  H
In check: N                            In check: N
Next move: W                           Next move: W
====  END GAME OUTPUT  ====            ====  END GAME OUTPUT  ====

==== BEGIN GAME INPUT ====             ==== BEGIN GAME INPUT ====
D6,A6
====  END GAME INPUT  ====
```

C. Move rule – Bishop

A bishop moves any number of vacant squares in any diagonal direction.

```
==== BEGIN GAME OUTPUT ====              ==== BEGIN GAME OUTPUT ====
  -----------------------                  -----------------------
8|.. .. BB .. BK BB .. ..|               8|.. .. BB .. BK BB .. ..|
7|.. .. BP .. BP BP BP ..|               7|.. .. BP .. BP BP BP ..|
6|.. BP .. BR .. .. .. ..|               6|.. BP .. BR .. .. .. WB|
5|.. .. .. .. .. .. .. BP|               5|.. .. .. .. .. .. .. BP|
4|BP .. .. .. .. .. .. ..|               4|BP .. .. .. .. .. .. ..|
3|WP WP WP .. .. .. .. WP|               3|WP WP WP .. .. .. .. WP|
2|.. .. .. .. WK WP WP ..|               2|.. .. .. .. WK WP WP ..|
1|WR .. WB .. .. WB .. WR|       ➔       1|WR .. .. .. .. WB .. WR|
  -----------------------                  -----------------------
  A  B  C  D  E  F  G  H                   A  B  C  D  E  F  G  H
In check: N                              In check: N
Next move: W                             Next move: B
====  END GAME OUTPUT  ====              ====  END GAME OUTPUT  ====

==== BEGIN GAME INPUT ====               ==== BEGIN GAME INPUT ====
C1,H6
====  END GAME INPUT  ====
```

A bishop cannot move where the square is already occupied by a friendly piece.

```
==== BEGIN GAME OUTPUT ====              ==== BEGIN GAME OUTPUT ====
  -----------------------                  -----------------------
8|.. .. BB .. BK BB .. ..|               8|.. .. BB .. BK BB .. ..|
7|.. .. BP .. BP .. BP ..|               7|.. .. BP .. BP .. BP ..|
6|.. .. .. BR .. BP .. WB|               6|.. .. .. BR .. BP .. WB|
5|.. BP .. .. .. .. .. BP|               5|.. BP .. .. .. .. .. BP|
4|BP .. .. .. .. .. .. ..|               4|BP .. .. .. .. .. .. ..|
3|WP WP WP .. .. .. .. WP|               3|WP WP WP .. .. .. .. WP|
2|.. .. .. WK .. WP WP ..|               2|.. .. .. WK .. WP WP ..|
1|WR .. .. .. .. WB .. WR|       ➔       1|WR .. .. .. .. WB .. WR|
  -----------------------                  -----------------------
  A  B  C  D  E  F  G  H                   A  B  C  D  E  F  G  H
In check: Y                              In check: Y
Next move: W                             Next move: W
====  END GAME OUTPUT  ====              ====  END GAME OUTPUT  ====

==== BEGIN GAME INPUT ====               ==== BEGIN GAME INPUT ====
H6,D2
====  END GAME INPUT  ====
```

A bishop can capture an enemy piece by moving onto its square.

```
==== BEGIN GAME OUTPUT ====          ==== BEGIN GAME OUTPUT ====
  ------------------------             ------------------------
8|.. .. BB .. BK BB .. ..|           8|.. .. BB .. BK BB .. ..|
7|.. .. BP .. BP .. BP ..|           7|.. .. BP .. BP .. BP ..|
6|.. .. .. BR .. BP .. WB|           6|.. .. .. BR .. BP .. WB|
5|.. BP .. .. .. .. .. BP|           5|.. WB .. .. .. .. .. BP|
4|BP .. .. .. .. .. .. ..|           4|BP .. .. .. .. .. .. ..|
3|WP WP WP .. .. .. .. WP|           3|WP WP WP .. .. .. .. WP|
2|.. .. .. WK .. WP WP ..|           2|.. .. .. WK .. WP WP ..|
1|WR .. .. .. .. WB .. WR|      ➔    1|WR .. .. .. .. .. .. WR|
  ------------------------             ------------------------
  A  B  C  D  E  F  G  H               A  B  C  D  E  F  G  H
In check: Y                          In check: Y
Next move: W                         Next move: B
====  END GAME OUTPUT  ====          ====  END GAME OUTPUT  ====

==== BEGIN GAME INPUT ====           ==== BEGIN GAME INPUT ====
F1,B5
====  END GAME INPUT  ====
```

A bishop cannot leap over other pieces.

```
==== BEGIN GAME OUTPUT ====          ==== BEGIN GAME OUTPUT ====
  ------------------------             ------------------------
8|.. .. BB .. BK BB .. ..|           8|.. .. BB .. BK BB .. ..|
7|.. .. BP .. BP .. BP ..|           7|.. .. BP .. BP .. BP ..|
6|.. .. BR .. .. BP .. WB|           6|.. .. BR .. .. BP .. WB|
5|.. WB .. .. .. .. .. BP|           5|.. WB .. .. .. .. .. BP|
4|BP .. .. .. .. .. .. ..|           4|BP .. .. .. .. .. .. ..|
3|WP WP WP .. .. .. .. WP|           3|WP WP WP .. .. .. .. WP|
2|.. .. .. WK .. WP WP ..|           2|.. .. .. WK .. WP WP ..|
1|WR .. .. .. .. .. .. WR|      ➔    1|WR .. .. .. .. .. .. WR|
  ------------------------             ------------------------
  A  B  C  D  E  F  G  H               A  B  C  D  E  F  G  H
In check: N                          In check: N
Next move: W                         Next move: W
====  END GAME OUTPUT  ====          ====  END GAME OUTPUT  ====

==== BEGIN GAME INPUT ====           ==== BEGIN GAME INPUT ====
B5,D7
====  END GAME INPUT  ====
```

D. Move rule – Pawn

A pawn moves straight forward one square where there's no piece.

```
==== BEGIN GAME OUTPUT ====
  ------------------------
8|.. .. BB .. BK BB .. ..|
7|.. .. BP .. BP .. BP ..|
6|.. .. BR .. .. BP .. WB|
5|.. WB .. .. .. .. .. BP|
4|BP .. .. .. .. .. .. ..|
3|WP WP WP .. .. .. .. WP|
2|.. .. .. WK .. WP WP ..|
1|WR .. .. .. .. .. .. WR|
  ------------------------
  A  B  C  D  E  F  G  H
In check: N
Next move: W
====  END GAME OUTPUT  ====

==== BEGIN GAME INPUT ====
G2,G3
====  END GAME INPUT  ====
```

➔

```
==== BEGIN GAME OUTPUT ====
  ------------------------
8|.. .. BB .. BK BB .. ..|
7|.. .. BP .. BP .. BP ..|
6|.. .. BR .. .. BP .. WB|
5|.. WB .. .. .. .. .. BP|
4|BP .. .. .. .. .. .. ..|
3|WP WP WP .. .. .. WP WP|
2|.. .. .. WK .. WP .. ..|
1|WR .. .. .. .. .. .. WR|
  ------------------------
  A  B  C  D  E  F  G  H
In check: N
Next move: B
====  END GAME OUTPUT  ====

==== BEGIN GAME INPUT ====
```

A pawn cannot move backward or lateral side.

```
==== BEGIN GAME OUTPUT ====
  ------------------------
8|.. .. BB .. BK BB .. ..|
7|.. .. BP .. BP .. BP ..|
6|.. .. BR .. .. BP .. WB|
5|.. WB .. .. .. .. .. BP|
4|BP .. .. .. .. .. .. ..|
3|WP WP WP .. .. .. WP WP|
2|.. .. .. WK .. WP .. ..|
1|WR .. .. .. .. .. .. WR|
  ------------------------
  A  B  C  D  E  F  G  H
In check: N
Next move: B
====  END GAME OUTPUT  ====

==== BEGIN GAME INPUT ====
F6,F7
====  END GAME INPUT  ====
```

➔

```
==== BEGIN GAME OUTPUT ====
  ------------------------
8|.. .. BB .. BK BB .. ..|
7|.. .. BP .. BP .. BP ..|
6|.. .. BR .. .. BP .. WB|
5|.. WB .. .. .. .. .. BP|
4|BP .. .. .. .. .. .. ..|
3|WP WP WP .. .. .. WP WP|
2|.. .. .. WK .. WP .. ..|
1|WR .. .. .. .. .. .. WR|
  ------------------------
  A  B  C  D  E  F  G  H
In check: N
Next move: B
====  END GAME OUTPUT  ====

==== BEGIN GAME INPUT ====
```

A pawn cannot move backward or lateral side.

```
==== BEGIN GAME OUTPUT ====
  ------------------------
8|.. .. BB .. BK BB .. ..|
7|.. .. BP .. BP .. BP ..|
6|.. .. BR .. .. BP .. WB|
5|.. WB .. .. .. .. .. BP|
4|BP .. .. .. .. .. .. ..|
3|WP WP WP .. .. .. WP WP|
2|.. .. .. WK .. WP .. ..|
1|WR .. .. .. .. .. .. WR|
  ------------------------
   A  B  C  D  E  F  G  H
In check: N
Next move: B
====  END GAME OUTPUT  ====

==== BEGIN GAME INPUT ====
F6,E6
====  END GAME INPUT  ====
```

➔

```
==== BEGIN GAME OUTPUT ====
  ------------------------
8|.. .. BB .. BK BB .. ..|
7|.. .. BP .. BP .. BP ..|
6|.. .. BR .. .. BP .. WB|
5|.. WB .. .. .. .. .. BP|
4|BP .. .. .. .. .. .. ..|
3|WP WP WP .. .. .. WP WP|
2|.. .. .. WK .. WP .. ..|
1|WR .. .. .. .. .. .. WR|
  ------------------------
   A  B  C  D  E  F  G  H
In check: N
Next move: B
====  END GAME OUTPUT  ====

==== BEGIN GAME INPUT ====
```

A pawn can capture an opponent's piece on a square diagonally.

```
==== BEGIN GAME OUTPUT ====
  ------------------------
8|.. .. BB .. BK BB .. ..|
7|.. .. BP .. .. .. BP ..|
6|.. .. BR .. BP BP .. ..|
5|.. WB .. .. .. .. WB BP|
4|BP .. .. .. .. .. .. ..|
3|WP WP WP .. .. .. WP WP|
2|.. .. .. WK .. WP .. ..|
1|WR .. .. .. .. .. .. WR|
  ------------------------
   A  B  C  D  E  F  G  H
In check: N
Next move: B
====  END GAME OUTPUT  ====

==== BEGIN GAME INPUT ====
F6,G5
====  END GAME INPUT  ====
```

➔

```
==== BEGIN GAME OUTPUT ====
  ------------------------
8|.. .. BB .. BK BB .. ..|
7|.. .. BP .. .. .. BP ..|
6|.. .. BR .. BP .. .. ..|
5|.. WB .. .. .. .. BP BP|
4|BP .. .. .. .. .. .. ..|
3|WP WP WP .. .. .. WP WP|
2|.. .. .. WK .. WP .. ..|
1|WR .. .. .. .. .. .. WR|
  ------------------------
   A  B  C  D  E  F  G  H
In check: N
Next move: W
====  END GAME OUTPUT  ====

==== BEGIN GAME INPUT ====
```

E. 'In check' decision

'In check' state

```
==== BEGIN GAME OUTPUT ====            ==== BEGIN GAME OUTPUT ====
  -----------------------                -----------------------
8|.. .. BB BK .. .. .. ..|             8|.. .. BB BK .. .. .. ..|
7|.. .. BP .. .. .. BP ..|             7|.. .. BP .. .. .. BP ..|
6|.. .. BR .. .. .. .. ..|             6|.. .. BR .. .. .. .. ..|
5|.. WB BB .. .. .. BP BP|             5|.. WB BB .. .. .. BP BP|
4|BP WP .. .. WK .. .. ..|             4|BP WP .. .. WK .. .. ..|
3|WP .. WP .. .. .. WP WP|             3|WP .. WP .. .. .. WP WP|
2|.. .. .. .. .. WP .. ..|             2|.. .. .. .. .. WP .. ..|
1|WR .. .. .. .. .. .. WR|      ➔      1|WR .. .. WR .. .. .. ..|
  -----------------------                -----------------------
  A  B  C  D  E  F  G  H                 A  B  C  D  E  F  G  H
In check: N                            In check: Y
Next move: W                           Next move: B
====  END GAME OUTPUT  ====            ====  END GAME OUTPUT  ====

==== BEGIN GAME INPUT ====             ==== BEGIN GAME INPUT ====
H1,D1
====  END GAME INPUT  ====
```

'In check' state

```
==== BEGIN GAME OUTPUT ====            ==== BEGIN GAME OUTPUT ====
  -----------------------                -----------------------
8|.. .. BB BK .. .. .. ..|             8|.. .. .. BK .. .. .. ..|
7|.. .. BP .. .. .. .. ..|             7|.. .. BP .. .. .. .. ..|
6|.. .. BR .. .. .. BP ..|             6|.. .. BR .. .. .. BP ..|
5|.. WB BB .. .. .. BP BP|             5|.. WB BB .. .. BB BP BP|
4|BP WP .. .. WK .. .. ..|             4|BP WP .. .. WK .. .. ..|
3|WP .. WP .. .. .. WP WP|             3|WP .. WP .. .. .. WP WP|
2|.. .. .. .. .. WP .. ..|             2|.. .. .. .. .. WP .. ..|
1|WR .. WR .. .. .. .. ..|      ➔      1|WR .. WR .. .. .. .. ..|
  -----------------------                -----------------------
  A  B  C  D  E  F  G  H                 A  B  C  D  E  F  G  H
In check: N                            In check: Y
Next move: B                           Next move: W
====  END GAME OUTPUT  ====            ====  END GAME OUTPUT  ====

==== BEGIN GAME INPUT ====             ==== BEGIN GAME INPUT ====
C8,F5
====  END GAME INPUT  ====
```

F. Winner/draw decision

Winner

```
==== BEGIN GAME OUTPUT ====
  -----------------------
8|.. .. .. BK .. .. .. ..|
7|.. .. BP .. .. .. .. ..|
6|.. .. BR .. .. .. BP ..|
5|.. WB BB .. .. BB BP BP|
4|BP WP .. .. WK .. .. ..|
3|WP .. WP .. .. .. WP WP|
2|.. .. .. .. .. WP .. ..|
1|WR .. .. WR .. .. .. ..|
  -----------------------
  A  B  C  D  E  F  G  H
In check: Y
Next move: B
====  END GAME OUTPUT  ====

==== BEGIN GAME INPUT ====
F5,E4
====  END GAME INPUT  ====
```

➔

```
==== BEGIN GAME OUTPUT ====
Winner: B
====  END GAME OUTPUT  ====
```

Winner

```
==== BEGIN GAME OUTPUT ====
  -----------------------
8|.. .. BB .. BK BB .. BR|
7|.. BP .. BP .. .. .. ..|
6|.. BR BP .. .. BP .. BP|
5|BP WP .. .. WR .. BP ..|
4|WP .. .. .. .. .. .. ..|
3|WB .. .. WP .. .. .. WP|
2|.. .. WP .. WP WP WP ..|
1|.. .. .. .. WK WB .. WR|
  -----------------------
  A  B  C  D  E  F  G  H
In check: Y
Next move: W
====  END GAME OUTPUT  ====

==== BEGIN GAME INPUT ====
E5,E8
====  END GAME INPUT  ====
```

➔

```
==== BEGIN GAME OUTPUT ====
Winner: W
====  END GAME OUTPUT  ====
```

Draw

```
==== BEGIN GAME OUTPUT ====
  -----------------------
8|BK .. .. .. WR .. .. ..|
7|.. .. .. .. .. .. .. ..|
6|.. .. .. .. WB .. .. ..|
5|WR .. .. .. .. .. .. ..|
4|.. .. .. .. .. .. .. ..|
3|.. .. .. .. .. .. .. ..|
2|.. .. .. .. .. .. .. ..|
1|.. .. .. .. WK .. .. ..|
  -----------------------
  A  B  C  D  E  F  G  H
In check: Y
Next move: W
==== END GAME OUTPUT ====

==== BEGIN GAME INPUT ====
E6,D5
==== END GAME INPUT ====
```

➔

```
==== BEGIN GAME OUTPUT ====
Winner: D
==== END GAME OUTPUT ====
```

Draw

```
==== BEGIN GAME OUTPUT ====
  -----------------------
8|.. .. BK .. .. .. .. ..|
7|.. .. .. .. .. .. .. ..|
6|.. .. .. .. BP .. .. ..|
5|.. BP .. .. WP .. .. ..|
4|.. WP .. .. .. .. .. ..|
3|.. .. .. .. .. .. .. BR|
2|.. .. .. .. .. .. .. ..|
1|.. BR .. .. WK .. .. ..|
  -----------------------
  A  B  C  D  E  F  G  H
In check: Y
Next move: B
==== END GAME OUTPUT ====

==== BEGIN GAME INPUT ====
H3,H2
==== END GAME INPUT ====
```

➔

```
==== BEGIN GAME OUTPUT ====
Winner: D
==== END GAME OUTPUT ====
```

5. How to build

```
$ cmake . (make 'Makefile' file)
$ make (build)
```

6. References

[1] https://en.wikipedia.org/wiki/Singleton_pattern

[2] https://en.wikipedia.org/wiki/Strategy_pattern

[3] https://en.wikipedia.org/wiki/King_(chess)

[4] https://en.wikipedia.org/wiki/Rook_(chess)

[5] https://en.wikipedia.org/wiki/Bishop_(chess)

[6] https://en.wikipedia.org/wiki/Pawn_(chess)

[7] https://en.wikipedia.org/wiki/Check_(chess)

[8] https://en.wikipedia.org/wiki/Chess

[9] https://en.wikipedia.org/wiki/Stalemate

- End of Document -

Appendix - Source Codes

Filename	Description
main.cpp	- Entry point function, main()
ChessPiece.h	- Abstract class declaration for *CChessPiece* class (Strategy pattern)
ChessPieceKing.h ChessPieceKing cpp	- *CChessPieceKing* class declaration and implementation - Derived from *CChessPiece* class - Collects next possible positions of the given king piece.
ChessPieceRook.h ChessPieceRook.cpp	- *CChessPieceRook* class declaration and implementation - derived from *CChessPiece* class - Collects next possible positions of the given rook piece.
ChessPieceBish.h ChessPieceBish.cpp	- *CChessPieceBish* class declaration and implementation - Derived from *CChessPiece* class - Collects next possible positions of the given bishop piece.
ChessPiecePawn.h ChessPiecePawn.cpp	- *CChessPiecePawn* class declaration and implementation - Derived from *CChessPiece* class - Collects next possible positions of the given pawn piece.
ChessBoard.h ChessBoard.cpp	- *CChessBoard* class declaration and implementation - Derived from TSingleton template class (Singleton pattern) - Manages a game, manipulates all pieces, and decides winner/draw.

● main.cpp

```
///
/// @file       main.cpp
/// @author     Junpyo Hong (jp7.hong@gmail.com)
/// @date       Nov. 14, 2017
/// @version    1.0
/// @brief      entry point function of the program
/// @remark     Tab size: 4
///

#include "ChessBoard.h"

/// @brief      entry point function of the program
/// @param      argc [in] the number of arguments being passed into this program
/// @param      argv [in] character array of arguments
/// @return     0 on success
int main(int argc, char *argv[])
{
    CChessBoard::GetInstance()->Run();

    return 0;
}
```

● ChessPiece.h

```
///
/// @file       ChessPiece.h
/// @author     Junpyo Hong (jp7.hong@gmail.com)
/// @date       Nov. 14, 2017
/// @version    1.0
/// @brief      abstract class for chess piece types (strategy pattern)
/// @remark     Tab size: 4
///

#ifndef _CHESS_PIECE_H_
#define _CHESS_PIECE_H_

#include <utility>      // std::pair
#include <vector>       // std::vector (used at derived class)
#include <cassert>      // assert (used at derived class)

/// length of column and row
#ifndef BOARD_LEN
#define BOARD_LEN   (8)
#endif

/// check if a given position is valid
#define IS_POS_WITHIN_RANGE(x,y) \
    ((x) >= 0 && (x) <= BOARD_LEN - 1 && (y) >= 0 && (y) <= BOARD_LEN - 1)

/// @brief      abstract class for chess piece types (strategy pattern)
class CChessPiece
{
public:
    explicit CChessPiece(const int cColor, const char cName, \
        const char col, const char row)
    : m_cColor(cColor)
    , m_cName(cName)
    , m_bExist(true)
    , m_pairPos(std::make_pair(col - 'A', row - '1')) {}
    virtual ~CChessPiece() {}

    char GetColor() { return m_cColor; }
    char GetName() { return m_cName; }
    bool IsExist() { return m_bExist; }
    void Remove() { m_bExist = false; }
    std::pair<int, int> GetPos() { return m_pairPos; }
    void SetPos(const int x, const int y) { m_pairPos = std::make_pair(x, y); }

    virtual std::vector<std::pair<int, int>> GetPossiblePos() = 0;

private:
    /// non construction-copyable
    CChessPiece(const CChessPiece&);

    /// non copyable
    const CChessPiece& operator=(const CChessPiece&);

protected:
    char m_cColor;                  ///< 'W' or 'B'
    char m_cName;                   ///< 'K', 'R', 'B', or 'P'
    bool m_bExist;                  ///< true or false
    std::pair<int, int> m_pairPos;  ///< x(col), y(row) (integer!)
};

#endif // _CHESS_PIECE_H_
```

● ChessPieceKing.h

```
///
/// @file       ChessPieceKing.h
/// @author     Junpyo Hong (jp7.hong@gmail.com)
/// @date       Nov. 14, 2017
/// @version    1.0
/// @brief      King piece class (derived from CChessPiece class)
/// @remark     Tab size: 4
///

#ifndef _CHESS_PIECE_KING_H_
#define  CHESS PIECE KING H

#include "ChessPiece.h"

/// @brief      King piece class (derived from CChessPiece class)
class CChessPieceKing : public CChessPiece
{
public:
    explicit CChessPieceKing(const char color, const char name, \
        const char col, const char row)
        : CChessPiece(color, name, col, row) {}
    virtual ~CChessPieceKing() {}

    virtual std::vector<std::pair<int, int>> GetPossiblePos();

private:
    /// non construction-copyable
    CChessPieceKing(const CChessPieceKing&);

    /// non copyable
    const CChessPieceKing& operator=(const CChessPieceKing&);
};

#endif //  CHESS PIECE KING H
```

● ChessPieceKing.cpp

```
///
/// @file       ChessPieceKing.cpp
/// @author     Junpyo Hong (jp7.hong@gmail.com)
/// @date       Nov. 14, 2017
/// @version    1.0
/// @brief      King piece class (derived from CChessPiece class)
/// @remark     Tab size: 4
///

#include "ChessPieceKing.h"
#include "ChessBoard.h"     // CChessBoard

/// @brief      get possible positions of this piece
/// @param      N/A
/// @return     position vector which contains possible positions
std::vector<std::pair<int, int>>
CChessPieceKing::GetPossiblePos()
{
    std::vector<std::pair<int, int>> vRet;

    // my position
    int nSrcX = GetPos().first;
    int nSrcY = GetPos().second;

    // check whether it's within normal range
    assert(IS_POS_WITHIN_RANGE(nSrcX, nSrcY));

    // (a) The king moves exactly one square horizontally, vertically, or diagonally.
    // (b) The king cannot move where the square is already occupied by a friendly piece.
    // (c) The king captures an enemy piece by moving onto its square.
    // (d) The king cannot move where it would place itself in check.
    // (e) The king cannot capture the opposing king.
    // *** In this program, it doesn't consider the rule (d), (e), and other exceptional
    // rules.

    // rule (a): can move exactly one square horizontally, vertically, or diagonally.
    std::vector<std::pair<int, int>> vPairRelPos =
    {
        std::make_pair( 0, +1), // (1)   //    +1   (8) | (1) | (2)
        std::make_pair(+1, +1), // (2)   //        -----+-----+-----
        std::make_pair(+1,  0), // (3)   // y   0   (7) | [K] | (3)
        std::make_pair(+1, -1), // (4)   //        -----+-----+-----
        std::make_pair( 0, -1), // (5)   //    -1   (6) | (5) | (4)
        std::make_pair(-1, -1), // (6)   //
        std::make_pair(-1,  0), // (7)   //         -1     0    +1
        std::make_pair(-1, +1), // (8)   //                x
    };

    // check move rules for the king
    std::vector<std::pair<int, int>>::iterator it;
    for (it = vPairRelPos.begin(); it != vPairRelPos.end(); ++it)
    {
        // candidate position
        int nCndX = nSrcX + it->first;
        int nCndY = nSrcY + it->second;

        // if candidate position is out of range, just skip
        if (!IS_POS_WITHIN_RANGE(nCndX, nCndY))
```

```
                continue;

            // get a color at the candidate position
            char cCndColor = CChessBoard::GetInstance()->GetPosColor(nCndX, nCndY);

            // if candidate square is vacant or a enemy, then can move (rule (b) and (c))
            if (cCndColor != GetColor())
            {
                // add to possible position
                vRet.push_back(std::make_pair(nCndX, nCndY));
            }
        }

        return vRet;
    }
```

- ChessPieceRook.h

```
///
/// @file       ChessPieceRook.h
/// @author     Junpyo Hong (jp7.hong@gmail.com)
/// @date       Nov. 14, 2017
/// @version    1.0
/// @brief      Rook piece class (derived from CChessPiece class)
/// @remark     Tab size: 4
///

#ifndef _CHESS_PIECE_ROOK_H_
#define _CHESS_PIECE_ROOK_H_

#include "ChessPiece.h"

/// @brief      Rook piece class (derived from CChessPiece class)
class CChessPieceRook : public CChessPiece
{
public:
    explicit CChessPieceRook(const char color, const char name, \
        const char col, const char row)
        : CChessPiece(color, name, col, row) {}
    virtual ~CChessPieceRook() {}

    virtual std::vector<std::pair<int, int>> GetPossiblePos();

private:
    /// non construction-copyable
    CChessPieceRook(const CChessPieceRook&);

    /// non copyable
    const CChessPieceRook& operator=(const CChessPieceRook&);
};

#endif // _CHESS_PIECE_ROOK_H_
```

- ChessPieceRook.cpp

```
///
/// @file       ChessPieceRook.cpp
/// @author     Junpyo Hong (jp7.hong@gmail.com)
/// @date       Nov. 14, 2017
/// @version    1.0
/// @brief      Rook piece class (derived from CChessPiece class)
/// @remark     Tab size: 4
///

#include "ChessPieceRook.h"
#include "ChessBoard.h"     // CChessBoard

/// @brief      get possible positions of this piece
/// @param      N/A
/// @return     position vector which contains possible positions
std::vector<std::pair<int, int>>
CChessPieceRook::GetPossiblePos()
{
    std::vector<std::pair<int, int>> vRet;

    // my position
    int nSrcX = GetPos().first;
    int nSrcY = GetPos().second;

    // check whether it's within normal range
    assert(IS_POS_WITHIN_RANGE(nSrcX, nSrcY));

    // (a) A rook moves any number of vacant squares in a horizontal or vertical direction.
    // (b) A rook cannot move where the square is already occupied by a friendly piece.
    // (c) A rook can capture an enemy piece by moving onto its square.
    // (d) A rook cannot leap over other pieces.

    // comparing step
    std::vector<std::pair<int, int>> vDeltaXY = \
        { { -1, 0 }, { +1, 0 }, { 0, -1 }, { 0, +1 } };

    // check move rules for a rook
    for (size_t i = 0; i < vDeltaXY.size(); i++)
    {
        for (size_t j = 1; j < BOARD_LEN; j++)  // [1..7]
        {
            // rule (a): can move any squares in horiz/vert direction.
            int nCndX = nSrcX + (vDeltaXY[i].first * j);
            int nCndY = nSrcY + (vDeltaXY[i].second * j);

            // if it is out of range, do not check this direction anymore
            if (!IS_POS_WITHIN_RANGE(nCndX, nCndY))
                break;

            // get a color at the candidate position
            char cCndColor = CChessBoard::GetInstance()->GetPosColor(nCndX, nCndY);

            // rule (a): can move any squares in horiz/vert direction.
            if (cCndColor == CChessBoard::EPieceColor::NO_COLOR)
            {
                // add (nCndX, nCndY) to possible positions
                vRet.push_back(std::make_pair(nCndX, nCndY));
            }
            // rule (b): cannot move to the position of our troops.
            else if (cCndColor == GetColor())    // friendly troops
            {
                // rule (d): cannot leap over other pieces.
                break;
            }
            // rule (c): can capture an enemy piece by moving onto its square.
            else    // enemy
            {
                // add (nCndX, nCndY) to possible positions and break
                vRet.push_back(std::make_pair(nCndX, nCndY));

                // rule (d): cannot leap over other pieces.
                break;
            }
        }
    }

    return vRet;
}
```

- ChessPieceBish.h

```
///
/// @file       ChessPieceBish.h
/// @author     Junpyo Hong (jp7.hong@gmail.com)
/// @date       Nov. 14, 2017
/// @version    1.0
/// @brief      Bishop piece class (derived from CChessPiece class)
/// @remark     Tab size: 4
///

#ifndef _CHESS_PIECE_BISH_H_
#define _CHESS_PIECE_BISH_H_

#include "ChessPiece.h"

/// @brief      Bishop piece class (derived from CChessPiece class)
class CChessPieceBish : public CChessPiece
{
public:
    explicit CChessPieceBish(const char color, const char name, \
        const char col, const char row)
        : CChessPiece(color, name, col, row) {}
    virtual ~CChessPieceBish() {}

    virtual std::vector<std::pair<int, int>> GetPossiblePos();

private:
    /// non construction-copyable
    CChessPieceBish(const CChessPieceBish&);

    /// non copyable
    const CChessPieceBish& operator=(const CChessPieceBish&);
};

#endif // _CHESS_PIECE_BISH_H_
```

- ChessPieceBish.cpp

```
///
/// @file       ChessPieceBish.cpp
/// @author     Junpyo Hong (jp7.hong@gmail.com)
/// @date       Nov. 14, 2017
/// @version    1.0
/// @brief      Bishop piece class (derived from CChessPiece class)
/// @remark     Tab size: 4
///

#include "ChessPieceBish.h"
#include "ChessBoard.h"     // CChessBoard

/// @brief      get possible positions of this piece
/// @param      N/A
/// @return     position vector which contains possible positions
std::vector<std::pair<int, int>>
CChessPieceBish::GetPossiblePos()
{
    std::vector<std::pair<int, int>> vRet;

    // my position
    int nSrcX = GetPos().first;
    int nSrcY = GetPos().second;

    // check whether it's within normal range
    assert(IS_POS_WITHIN_RANGE(nSrcX, nSrcY));

    // (a) A bishop moves any number of vacant squares in any diagonal direction.
    // (b) A bishop cannot move where the square is already occupied by a friendly piece.
    // (c) A bishop can capture an enemy piece by moving onto its square.
    // (d) A bishop cannot leap over other pieces.

    // comparing step
    std::vector<std::pair<int, int>> vDeltaXY = \
        { { -1, -1 }, { -1, +1 }, { +1, -1 }, { +1, +1 } };

    // check move rules for a bishop
    for (size_t i = 0; i < vDeltaXY.size(); i++)
    {
        for (size_t j = 1; j < BOARD_LEN; j++)  // [1..7]
        {
            // rule (a): can move any squares in any diagonal direction.
            int nCndX = nSrcX + (vDeltaXY[i].first * j);
            int nCndY = nSrcY + (vDeltaXY[i].second * j);

            // if it is out of range, do not check this direction anymore
            if (!IS_POS_WITHIN_RANGE(nCndX, nCndY))
                break;

            // get a color at the candidate position
            char cCndColor = CChessBoard::GetInstance()->GetPosColor(nCndX, nCndY);

            // rule (a): can move any squares in horiz/vert direction
            if (cCndColor == CChessBoard::EPieceColor::NO_COLOR)
            {
                // add (nCndX, nCndY) to possible positions
                vRet.push_back(std::make_pair(nCndX, nCndY));
            }
            // rule (b): cannot move to the position of our troops
            else if (cCndColor == GetColor())    // friendly troops
            {
                // rule (d): cannot leap over other pieces.
                break;
            }
            // rule (c): can capture an enemy piece by moving onto its square.
            else    // enemy
            {
                // add (nCndX, nCndY) to possible positions and break
                vRet.push_back(std::make_pair(nCndX, nCndY));

                // rule (d): cannot leap over other pieces.
                break;
            }
        }
    }

    return vRet;
}
```

- ChessPiecePawn.h

```
///
/// @file       ChessPiecePawn.h
/// @author     Junpyo Hong (jp7.hong@gmail.com)
/// @date       Nov. 14, 2017
/// @version    1.0
/// @brief      Pawn piece class (derived from CChessPiece class)
/// @remark     Tab size: 4
///

#ifndef _CHESS_PIECE_PAWN_H_
#define _CHESS_PIECE_PAWN_H_

#include "ChessPiece.h"

/// @brief      Pawn piece class (derived from CChessPiece class)
class CChessPiecePawn : public CChessPiece
{
public:
    explicit CChessPiecePawn(const char color, const char name, \
        const char col, const char row)
        : CChessPiece(color, name, col, row) {}
    virtual ~CChessPiecePawn() {}

    virtual std::vector<std::pair<int, int>> GetPossiblePos();

private:
    /// non construction-copyable
    CChessPiecePawn(const CChessPiecePawn&);

    /// non copyable
    const CChessPiecePawn& operator=(const CChessPiecePawn&);
};

#endif // _CHESS_PIECE_PAWN_H_
```

- ChessPiecePawn.cpp

```
/// @file       ChessPiecePawn.cpp
/// @author     Junpyo Hong (jp7.hong@gmail.com)
/// @date       Nov. 14, 2017
/// @version    1.0
/// @brief      Pawn piece class (derived from CChessPiece class)
/// @remark     Tab size: 4
///

#include "ChessPiecePawn.h"
#include "ChessBoard.h"     // CChessBoard

/// @brief      get possible positions of this piece
/// @param      N/A
/// @return     position vector which contains possible positions
std::vector<std::pair<int, int>>
CChessPiecePawn::GetPossiblePos()
{
    std::vector<std::pair<int, int>> vRet;

    // my position
    int nSrcX = GetPos().first;
    int nSrcY = GetPos().second;

    // check whether it's within normal range
    assert(IS_POS_WITHIN_RANGE(nSrcX, nSrcY));

    // (a) A pawn moves straight forward one square where there's no piece.
    // (b) A pawn cannot move backward or lateral side.
    // (c) A pawn can capture an opponent's piece on a square diagonally.
    // (d) On first move, a pawn can advance two square along the same file,
    //     provided both square are unoccupied.
    // *** In this program, it doesn't consider the rule (d) and other exceptional rules.

    // rule (a): can move straight forward one square where there's no piece.
    // rule (b): cannot move backward or lateral side.
    std::vector<std::pair<int, int>> vPairRelPosWhite =
    {                                   //  +1   (2) | (1) | (3)
        std::make_pair(0, +1),  // (1)  //      -----+-----+-----
        std::make_pair(-1, +1), // (2)  // y 0    -  | [P] |  -
        std::make_pair(+1, +1), // (3)  //      -----+-----+-----
    };                                  //    x: -1     0    +1

    std::vector<std::pair<int, int>> vPairRelPosBlack =
    {                                   //      -----+-----+-----
        std::make_pair(0, -1),  // (1)  // y 0    -  | [P] |  -
        std::make_pair(-1, -1), // (2)  //      -----+-----+-----
        std::make_pair(+1, -1), // (3)  //   -1   (2) | (1) | (3)
    };                                  //    x: -1     0    +1

    // set pointer according to the current turn color
    std::vector<std::pair<int, int>> *pVecPairRelPos = \
        (CChessBoard::GetInstance()->GetTurnColor() == \
        CChessBoard::EPieceColor::WHITE) ? \
        &vPairRelPosWhite : &vPairRelPosBlack;

    // check move rules for a pawn
    for (size_t i = 0; i < pVecPairRelPos->size(); i++)
    {
        // candidate position
        int nCndX = nSrcX + pVecPairRelPos->at(i).first;
        int nCndY = nSrcY + pVecPairRelPos->at(i).second;

        // if candidate position is out of range, just skip
        if (!IS_POS_WITHIN_RANGE(nCndX, nCndY))
            continue;

        // get a color at the candidate position
        char cCndColor = CChessBoard::GetInstance()->GetPosColor(nCndX, nCndY);

        // straight forward direction:
        // can move only when there's no piece. (rule (a) and (b))
        if (i == 0)
        {
            // if candidate position is not occupied, then add to possible position
            if (cCndColor == CChessBoard::EPieceColor::NO_COLOR)
                vRet.push_back(std::make_pair(nCndX, nCndY));
        }
        // diagonal direction: can move only when there's a enemy (rule (c))
        else if (
            (cCndColor == CChessBoard::EPieceColor::WHITE &&
            GetColor() == CChessBoard::EPieceColor::BLACK) ||
            (cCndColor == CChessBoard::EPieceColor::BLACK &&
            GetColor() == CChessBoard::EPieceColor::WHITE))
        {
            // add to possible position
            vRet.push_back(std::make_pair(nCndX, nCndY));
        }
    }

    return vRet;
}
```

- ChessBoard.h

```
///
/// @file       ChessBoard.h
/// @author     Junpyo Hong (jp7.hong@gmail.com)
/// @date       Nov. 15, 2017
/// @version    1.0
/// @brief      Chessboard class (singleton pattern)
/// @remark     Tab size: 4
///

#ifndef _CHESS_BOARD_H_
#define _CHESS_BOARD_H_

#include <vector>       // std::vector
#include <utility>      // std::pair, std::make_pair

#include "Singleton.h"
#include "ChessPiece.h"

/// length of column and row
#ifndef BOARD_LEN
#define BOARD_LEN   (8)
#endif

/// @brief      Chessboard class (singleton pattern)
class CChessBoard : public TSingleton<CChessBoard>
{
public:
    enum EPieceColor { NO_COLOR = '.', WHITE = 'W', BLACK = 'B' };
    enum EDecision { CONTINUE = 0, WIN_W, WIN_B, DRAW };

public:
    explicit CChessBoard();
    virtual ~CChessBoard();

    void Run();

    char GetTurnColor() { return m_cTurnColor; }
    char GetPosColor(const int x, const int y) { return m_arrSquare[y][x].cColor; }
    std::pair<char, char> GetSrcIdx() { return m_pairSrcIdx; }
    std::pair<char, char> GetDstIdx() { return m_pairDstIdx; }
    std::pair<int, int> GetSrcPos()
    {
        return std::make_pair<int, int>( \
            int(m_pairSrcIdx.first - 'A'), int(m_pairSrcIdx.second - '1'));
    }
    std::pair<int, int> GetDstPos()
    {
        return std::make_pair<int, int>( \
            int(m_pairDstIdx.first - 'A'), int(m_pairDstIdx.second - '1'));
    }
    std::vector<std::pair<int, int>> GetTroopsPos( \
        const int color, \
        const std::pair<int, int>& pairPosIgnore = std::pair<int, int>(-1, -1));

private:
    void Init();
    void Quit();
    bool GetInput();
    bool CheckMoveRule();
    void PostProcess();
    bool ShowOutput();
    int  MakeDecision();
    void ShowBoard();
    void Update();
    bool IsInCheck();

private:
    /// non construction-copyable
    CChessBoard(const CChessBoard&);

    /// non copyable
    const CChessBoard& operator=(const CChessBoard&);

private:
    typedef struct _tagSBoardGrid
    {
        char cColor;                          ///< color of a piece ('W' or 'B')
        char cName;                           ///< name of a piece ('K', 'R', 'B', or 'P')
        CChessPiece *pChessPiece;             ///< pointer to access a chess piece
    } SBoardGrid;

    const int m_nPieces = 13;                 ///< the number of pieces of a team
    char m_cTurnColor = WHITE;                ///< current turn (WHITE('W') or BLACK('B'))
    std::vector<CChessPiece*> m_vWhite;       ///< vector to hold white pieces
    std::vector<CChessPiece*> m_vBlack;       ///< vector to hold black pieces
    SBoardGrid m_arrSquare[BOARD_LEN][BOARD_LEN];   ///< matrix to show the chessboard
    std::pair<char, char> m_pairSrcIdx;       ///< user's source index (eg. "A2")
    std::pair<char, char> m_pairDstIdx;       ///< user's destination index (eg. "A3")
};

#endif // _CHESS_BOARD_H_
```

- ChessBoard.cpp

```
///
/// @file       ChessBoard.cpp
/// @author     Junpyo Hong (jp7.hong@gmail.com)
/// @date       Nov. 14, 2017
/// @version    1.0
/// @brief      Chessboard class (singleton pattern)
/// @remark     Tab size: 4
///

#include <iostream>     // std::cout
#include <string>       // std::getline
#include <algorithm>    // std::for_each
#include <cctype>       // toupper, isalpha, isdigit
#include <cassert>      // assert

#include "ChessBoard.h"
#include "ChessPieceKing.h"
#include "ChessPieceRook.h"
#include "ChessPieceBish.h"
#include "ChessPiecePawn.h"

/// @brief      constructor
/// @param      N/A
/// @return     N/A
CChessBoard::CChessBoard()
{
    // m_nPieces = 13 (King 1, Rook 2, Bishop 2, and Pawn 8)
    m_vWhite.reserve(m_nPieces);
    m_vBlack.reserve(m_nPieces);

    // allocate memory and initialize color and position of each pieces
    Init();
}

/// @brief      destructor
/// @param      N/A
/// @return     N/A
CChessBoard::~CChessBoard()
{
    // deallocate memory for all pieces
    Quit();
}

/// @brief      initialize color and position of each pieces
/// @param      N/A
/// @return     void
void
CChessBoard::Init()
{
    // CChessPieceXXXX arguments:
    // argument 1: 'W'(white) or 'B'(black)
    // argument 2: 'K'(king), 'R'(rook), 'B'(bishop), or 'P'(pawn)
    // argument 3: column index ['A'..'H'] (it's a character, not a int)
    // argument 4: row index ['1'..'8'] (it's a character, not a int)

    // make instances for white pieces
    m_vWhite.push_back(new CChessPieceKing('W', 'K', 'E', '1'));
    m_vWhite.push_back(new CChessPieceRook('W', 'R', 'A', '1'));
    m_vWhite.push_back(new CChessPieceRook('W', 'R', 'H', '1'));
    m_vWhite.push_back(new CChessPieceBish('W', 'B', 'C', '1'));
    m_vWhite.push_back(new CChessPieceBish('W', 'B', 'F', '1'));
    m_vWhite.push_back(new CChessPiecePawn('W', 'P', 'A', '2'));
    m_vWhite.push_back(new CChessPiecePawn('W', 'P', 'B', '2'));
    m_vWhite.push_back(new CChessPiecePawn('W', 'P', 'C', '2'));
    m_vWhite.push_back(new CChessPiecePawn('W', 'P', 'D', '2'));
    m_vWhite.push_back(new CChessPiecePawn('W', 'P', 'E', '2'));
    m_vWhite.push_back(new CChessPiecePawn('W', 'P', 'F', '2'));
    m_vWhite.push_back(new CChessPiecePawn('W', 'P', 'G', '2'));
    m_vWhite.push_back(new CChessPiecePawn('W', 'P', 'H', '2'));

    // make instances for black pieces
    m_vBlack.push_back(new CChessPieceKing('B', 'K', 'E', '8'));
    m_vBlack.push_back(new CChessPieceRook('B', 'R', 'A', '8'));
    m_vBlack.push_back(new CChessPieceRook('B', 'R', 'H', '8'));
    m_vBlack.push_back(new CChessPieceBish('B', 'B', 'C', '8'));
    m_vBlack.push_back(new CChessPieceBish('B', 'B', 'F', '8'));
    m_vBlack.push_back(new CChessPiecePawn('B', 'P', 'A', '7'));
    m_vBlack.push_back(new CChessPiecePawn('B', 'P', 'B', '7'));
    m_vBlack.push_back(new CChessPiecePawn('B', 'P', 'C', '7'));
    m_vBlack.push_back(new CChessPiecePawn('B', 'P', 'D', '7'));
    m_vBlack.push_back(new CChessPiecePawn('B', 'P', 'E', '7'));
    m_vBlack.push_back(new CChessPiecePawn('B', 'P', 'F', '7'));
    m_vBlack.push_back(new CChessPiecePawn('B', 'P', 'G', '7'));
    m_vBlack.push_back(new CChessPiecePawn('B', 'P', 'H', '7'));

    // update the board status to show
    Update();
}

/// @brief      deallocate memory for all pieces
/// @param      N/A
/// @return     void
void
CChessBoard::Quit()
{
    // deallocate memory for white/black pieces
    for (int i = 0; i < m_nPieces; i++)
    {
        delete m_vWhite[i];
        delete m_vBlack[i];
    }

    m_vWhite.clear();
    m_vBlack.clear();
}

/// @brief      main interface function for main.cpp
/// @param      N/A
/// @return     void
void CChessBoard::Run()
{
    // show board at the beginning
    ShowOutput();

    while (1)
    {
        // get user's input
        GetInput();

        // change turn if user's command is valid
        if (CheckMoveRule() == true)
        {
            // update position of a piece and remove a enemy if it captured
            PostProcess();

            // change turn
            m_cTurnColor = (m_cTurnColor == 'W') ? 'B' : 'W';
        }

        // show board or decision result (return false if the game is terminated)
        if (ShowOutput() == false)
            break;
    }
}

/// @brief      get positions of given colored troops (w/o including 'pairPosIgnore')
/// @param      color [in] CChessBoard::WHITE or CChessBoard::BLACK
/// @param      pairPosIgnore [in] position which is not to be included
/// @return     position vector of given colored troops (w/o including 'pairPosIgnore')
/// @remark     don't use 'pairPosIgnore' if you don't need to ignore specific position
std::vector<std::pair<int, int>>
CChessBoard::GetTroopsPos(const int color, \
const std::pair<int, int>& pairPosIgnore)
{
    std::vector<std::pair<int, int>> vRet;
    std::vector<CChessPiece*> *p = 0;

    assert(color == CChessBoard::WHITE || color == CChessBoard::BLACK);

    // set a vector pointer to a proper colored troops
    p = (color == WHITE) ? &m_vWhite : &m_vBlack;

    std::vector<CChessPiece*>::iterator it;
    for (it = p->begin(); it != p->end(); ++it)
    {
        // if it is a live piece
        if ((*it)->IsExist())
        {
            int x = (*it)->GetPos().first;
            int y = (*it)->GetPos().second;

            // add only if it is not same position
            if (std::make_pair(x, y) != pairPosIgnore)
                vRet.push_back(std::make_pair(x, y));
        }
    }

    return vRet;
}

/// @brief      get a user's input
/// @param      N/A
/// @return     true if inputted col/row index range is valid, otherwise false
bool
CChessBoard::GetInput()
{
    std::string s;

    std::cout << std::endl;
    std::cout << "==== BEGIN GAME INPUT ====" << std::endl;
    std::getline(std::cin, s);
    std::cout << "====  END GAME INPUT  ====" << std::endl << std::endl;

    // get a character from inputted string
```

```cpp
    char col_curr = s[0];
    char row_curr = s[1];
    char col_next = s[3];
    char row_next = s[4];

    // check alphabet, number, and comma (eg. "C3,D4")
    if (!isalpha(int(s[0])) ||  // 'C'
        !isdigit(int(s[1])) ||  // '3'
        !isalpha(int(s[3])) ||  // 'D'
        !isdigit(int(s[4])) ||  // '4'
        s[2] != ',')            // ','
        return false;

    // convert to upper case (eg. "C3,D4")
    col_curr = char(toupper(s[0]));     // 'C'
    col_next = char(toupper(s[3]));     // 'D'

    // column range check (available range: ['A'..'H'])
    if (col_curr < 'A' || col_curr > 'H' || col_next < 'A' || col_next > 'H')
        return false;

    // row range check (available range: ['1'..'8'])
    if (row_curr < '1' || row_curr > '8' || row_next < '1' || row_next > '8')
        return false;

    // check piece color, existence of the given current position
    int x = col_curr - 'A';
    int y = row_curr - '1';
    if (m_arrSquare[y][x].cColor != m_cTurnColor)
        return false;

    // set current/desired positions
    m_pairSrcIdx = std::make_pair(col_curr, row_curr);
    m_pairDstIdx = std::make_pair(col_next, row_next);

    return true;
}

/// @brief      check whether a user entered the move rule
/// @param      N/A
/// @return     true if the move rule is satisfied, otherwise false
bool
CChessBoard::CheckMoveRule()
{
    // pointer of chess piece of user's source index
    CChessPiece* pChessPiece = \
        m_arrSquare[GetSrcPos().second][GetSrcPos().first].pChessPiece;

    // if there's no piece at the source position of user's input
    if (!pChessPiece)
        return false;

    // check whether piece color is matched
    if (m_cTurnColor != pChessPiece->GetColor())
        return false;

    // get possible positions of user's source piece position
    std::vector<std::pair<int, int>> vPairPossiblePos;
    vPairPossiblePos = pChessPiece->GetPossiblePos();
    std::vector<std::pair<int, int>>::iterator it;
    for (it = vPairPossiblePos.begin(); it != vPairPossiblePos.end(); ++it)
    {
        // if a user's desired position is matched according to the move rule
        if (GetDstPos() == (*it))
        {
            return true;
        }
    }

    return false;
}

/// @brief      remove a enemy if it's captured and move player's piece
/// @param      N/A
/// @return     void
/// @remark     this function must be called when user's move command is valid
void
CChessBoard::PostProcess()
{
    int srcPosX = GetSrcPos().first;
    int srcPosY = GetSrcPos().second;
    int dstPosX = GetDstPos().first;
    int dstPosY = GetDstPos().second;
    char cEnemyColor = (m_cTurnColor == 'W') ? 'B' : 'W';

    // if there's a enemy at the desired position, remove it
    if (m_arrSquare[dstPosY][dstPosX].cColor == cEnemyColor)
        m_arrSquare[dstPosY][dstPosX].pChessPiece->Remove();

    // move a piece to the user's desired position
    m_arrSquare[srcPosY][srcPosX].pChessPiece->SetPos(dstPosX, dstPosY);
}

/// @brief      show game output (display board or result)
/// @param      N/A
/// @return     false if the game is terminated, otherwise true
bool
CChessBoard::ShowOutput()
{
    bool bRet = false;

    std::cout << "==== BEGIN GAME OUTPUT ====" << std::endl;

    switch (MakeDecision())
    {
    case WIN_W:
        std::cout << "Winner: W" << std::endl;
        break;

    case WIN_B:
        std::cout << "Winner: B" << std::endl;
        break;

    case DRAW:
        std::cout << "Winner: D" << std::endl;
        break;

    case CONTINUE:
    default:
        ShowBoard();
        bRet = true;    // the game should be continued
    }

    std::cout << "====  END GAME OUTPUT  ====" << std::endl;

    return bRet;
```

```
}

/// @brief      make a decision
/// @param      N/A
/// @return     NONE (no dicision), WIN_W (white win), WIN_B (black win), or DRAW
int
CChessBoard::MakeDecision()
{
    // check whether two kings are alive
    assert(m_vWhite[0]->GetName() == 'K');
    assert(m_vBlack[0]->GetName() == 'K');

    // (1) if the white king is not exist, then black wins.
    if (!m_vWhite[0]->IsExist())
        return WIN_B;

    // (2) if the black king is not exist, then white wins.
    if (!m_vBlack[0]->IsExist())
        return WIN_W;

    ////////////////////////////////////////////////////////////////////////////
    // (3) check if the 'stalemate' has happened

    // set the troops pointers according to the turn
    std::vector<CChessPiece*> *pTroops[2] = { 0, 0 };
    pTroops[0] = (m_cTurnColor == WHITE) ? &m_vWhite : &m_vBlack;
    pTroops[1] = (m_cTurnColor == WHITE) ? &m_vBlack : &m_vWhite;

    // the pTroops[0] king's possible position
    std::vector<std::pair<int, int>> vPairPosKingPossiblePos = \
        pTroops[0]->at(0)->GetPossiblePos();

    // possible positions of all pTroops[1] pieces
    std::vector<std::pair<int, int>> vPairPosEnemiesPossiblePos;
    for (size_t i = 0; i < pTroops[1]->size(); i++)
    {
        if (!pTroops[1]->at(i)->IsExist())
            continue;

        // possible positions of a pTroops[1] piece
        std::vector<std::pair<int, int>> vPairPosEnemyPossiblePos = \
            pTroops[1]->at(i)->GetPossiblePos();

        // possible positions of all pTroops[1] pieces
        vPairPosEnemiesPossiblePos.insert(vPairPosEnemiesPossiblePos.end(), \
            vPairPosEnemyPossiblePos.begin(), vPairPosEnemyPossiblePos.end());
    }

    // check if the king can avoid
    bool bKingCanMove = true;
    for (size_t i = 0; i < vPairPosKingPossiblePos.size(); i++)
    {
        // compare with each enemy's possible positions
        for (size_t j = 0; j < vPairPosEnemiesPossiblePos.size(); j++)
        {
            // if this square is a dangerous square
            if (vPairPosKingPossiblePos[i] == vPairPosEnemiesPossiblePos[j])
            {
                bKingCanMove = false;
                break;
            }
        }
        if (bKingCanMove)
            return CONTINUE;
    }

    // if pTroop[0] king cannot move, then check if other pieces can move
    if (!bKingCanMove)
    {
        for (size_t i = 1; i < pTroops[0]->size(); i++)
        {
            std::vector<std::pair<int, int>> vPairPos = \
                pTroops[0]->at(i)->GetPossiblePos();

            // if it is possible to move at least one piece, it's not 'statemate'
            if (vPairPos.size() > 0)
                return CONTINUE;
        }
    }

    ///////////////////////////////////////////////////////////////////////////

    return DRAW;
}

/// @brief      update chessboard state and show the board
/// @param      N/A
/// @return     void
void
CChessBoard::ShowBoard()
{
    // update the board status to show
    Update();

    std::cout << "  -----------------------" << std::endl;

    for (int y = BOARD_LEN - 1; y >= 0; y--)
    {
        std::cout << y + 1 << "|";

        for (int x = 0; x < BOARD_LEN; x++)
        {
            std::cout << m_arrSquare[y][x].cColor << m_arrSquare[y][x].cName;

            // don't print rightmost space
            if (x != 7)
                std::cout << " ";
        }

        std::cout << "|" << std::endl;
    }

    std::cout << "  -----------------------" << std::endl;
    std::cout << "  A  B  C  D  E  F  G  H" << std::endl;
    std::cout << "In check: " << (IsInCheck() ? "Y" : "N") << std::endl;
    std::cout << "Next move: " << m_cTurnColor << std::endl;    // add "Next" not to confuse
}

/// @brief      update the chessboard status to show
/// @param      N/A
/// @return     void
void
CChessBoard::Update()
{
    // clear board array to get new position information of pieces
    for (int y = 0; y < BOARD_LEN; y++)
    {
        for (int x = 0; x < BOARD_LEN; x++)
        {
            m_arrSquare[y][x].cColor = '.';
            m_arrSquare[y][x].cName = '.';
            m_arrSquare[y][x].pChessPiece = 0;
        }
    }

    // search live white/black pieces and update chessboard state
    for (size_t i = 0; i < 2; i++)
    {
        std::vector<CChessPiece*> *pTroops = \
            (i == 0) ? &m_vWhite : &m_vBlack;

        std::vector<CChessPiece*>::iterator it;
        for (it = pTroops->begin(); it != pTroops->end(); ++it)
        {
            if ((*it)->IsExist())
            {
                int x = (*it)->GetPos().first;
                int y = (*it)->GetPos().second;

                // set color and name
                m_arrSquare[y][x].cColor = (*it)->GetColor();
                m_arrSquare[y][x].cName = (*it)->GetName();
                m_arrSquare[y][x].pChessPiece = *it;
            }
        }
    }
}

/// @brief      check whether 'In check' state has happened
/// @param      N/A
/// @return     true if 'In check' has happened, otherwise false
bool
CChessBoard::IsInCheck()
{
    // check whether the two kings are alive.
    assert(m_vWhite[0]->GetName() == 'K');
    assert(m_vBlack[0]->GetName() == 'K');

    // Because I don't consider following rules
    // "The king cannot move where it would place itself in check." and
    // "The king cannot capture the opposing king.",
    // so it needs to check 'In check' state for both sides.

    for (size_t i = 0; i < 2; i++)
    {
        // set pointer of friendly/enemy troops
        std::vector<CChessPiece*> *pFriendlyTroops = 0;
        std::vector<CChessPiece*> *pEnemyTroops = 0;
        pFriendlyTroops = (i == 0) ? &m_vWhite : &m_vBlack;
        pEnemyTroops = (i == 0) ? &m_vBlack : &m_vWhite;

        // the king's position to check
        //std::pair<int, int> pairKingPos = pFriendlyTroops[0].at(0)->GetPos();
        std::pair<int, int> pairKingPos = pFriendlyTroops->at(0)->GetPos();

        // check 'In check' possibility
        std::vector<CChessPiece*>::iterator itEnemy = pEnemyTroops->begin();
        for (; itEnemy < pEnemyTroops->end(); ++itEnemy)
        {
            if ((*itEnemy)->IsExist())
            {
                // possible positions where a enemy can move
                std::vector<std::pair<int, int>> vPairEnemyAttackablePos = \
                    (*itEnemy)->GetPossiblePos();

                // check whether the king can be caught
                // for each possible position by the enemy
                std::vector<std::pair<int, int>>::iterator it;
                it = vPairEnemyAttackablePos.begin();
                for (; it != vPairEnemyAttackablePos.end(); ++it)
                {
                    // if the king's current position is overlapped
                    // with a possible position
                    if (pairKingPos == *it)
                        return true;
                }
            }
        }
    }

    return false;
}
```