

# Lab 3: Chess

---

```
01234567
-----
0| ♖ |
1|   ♙ |
2| ♗ |
3|   |
4|   |
5| ♖   ♘ |
6|   |
7|   |
-----
```

## Goal

---

In this project, you will develop a simplified chess game.

If you are not familiar with chess, feel free to experiment with the game: <https://www.chess.com/>

The version you will implement is simplified in a number of ways:

You will implement 4 types of pieces: a Pawn ♙, Rook ♖, Bishop ♗, King ♔.

The pieces will move as per normal chess rules.

The chess board need not necessarily be 8x8. Do not hard code the standard dimensions.

Some simplifications have been made:

1. Pawns do not get promoted to other pieces.
2. Castling is not allowed.
3. En Passant pawn capture is not allowed.
4. White piece move first.

## Part 1

---

You need to implement the functionality for all the chess pieces except the King (both the base class and the derived class), and the code necessary to realize the `isValidMove()` function. This function is used to verify if a proposed move is valid. During runtime, the program will receive a series of steps to check for validity.


## Part 2

---

The program should be able to move pieces across the grid via `movePiece()` , provided that the move is valid and it is the turn of the color of the moving piece. During runtime, the program will parse a series of steps to execute, with the chess board being updated should a move be executed. You will also implement a special function called `isPieceUnderThreat()` . This function checks if the chess piece at a certain position can be captured by any opponent piece. You do not have to worry about the King for this part as well.

## Part 3

---

Implement the King  and make sure that moves resulting in the king being in a state of check cannot be performed. There are perhaps many ways to achieve this, and a key part of the project is coming up with a carefully thought out design to address it (not just the implementation). This final section is more challenging, and we plan to limit our help to bouncing ideas with you on the design.

## Submission

---

Upload all `.hh` and `.cc` files to Gradescope. Any tests that you pass will be in green, any that you fail will be in red.

## Restriction

---

- If you do not need to use a function declared in `.hh`, you may define it to do nothing.
- You may add functions/members to the provided `.hh` files but you aren't allowed to remove any that we give you.
- Submit a `.cc` file for every `.hh` file, implementing all functions declared in their `.hh` files.
- The only includes allowed in the `.cc` files are the `.hh` files provided to you.
- No other C++ libraries are allowed nor are they needed.
- Turn off any print statements or calls to `displayBoard()` when you submit.  
`displayBoard()` is a function provided to help visualize the board and will not be checked.

## Concepts you will use in this project

---

The concepts that you would be using in this project include:

- inheritance
- constructor member initialiser list
- abstract class

- namespace
- virtual methods and polymorphism
- C++ containers: specifically, list and vector

# Components

---

## ChessBoard

---

- The ChessBoard class defines the board which is a rectangle with M rows and N columns. M and N are 8 in standard chess, but your code should work for more general numbers.
- The rows and columns are 0-indexed (i.e., they run from 0 to N-1). See figure.
- The initial state of the board can be arbitrary, and is read from a configuration file. If the initial state is as per the start of a standard chess game, black pieces are initially in rows 0 and 1, while white pieces are initially in rows N-2 and N-1.
- Contains information about the positions of all chess pieces.
- Returns status of a position, i.e. whether it is occupied.
- Allows players to execute a valid move.

## ChessPiece

---

- This is an abstract class detailing which methods to implement in derived chess pieces.
- The variables and methods common for all chess pieces should be defined here.

## PawnPiece

---

- This class implements the Pawn.
- A black piece can move in increasing row number (downwards in the figure).
- A white piece can move in decreasing row number (upwards in the figure).
- A black piece on row 1 can move 1 or 2 steps along the same column.
- A white piece on row n-2 can move 1 or 2 steps along the same column.
- In all other cases, a pawn can only move 1 step forward along the same column.
- A pawn can consume another piece by moving to a diagonally adjacent square with a higher row number for a black piece, or a lower row number for a white piece.

## RookPiece

---

- This class implements the Rook.
- A Rook can move horizontally or vertically with no limit to distance.

## BishopPiece

---

- This class implements the Bishop.
- A Bishop can move diagonally with no limit to distance.

## KingPiece

---

- This class implements the King.
- A King can move horizontally, vertically and diagonally by one step.
- For the first two milestones, do not worry about the King. For the final submission however, you must ensure that when a player makes a move, it does not leave her King in a position where it is under "check".

## Checking Move Validity

---

You should check for move validity following the rules of chess for the subset of pieces being implemented. Checks should include:

- Ensuring the piece that moves is of the correct color.
- The piece should not move to the same spot.
- The piece should not move out of bounds.
- Like in chess, the piece should not be obstructed by another piece between its initial and final position.
- The final position should not contain another piece of the same color.

## IsPieceUnderThreat() when implementing King for Part 3.

---

IsPieceUnderThreat() only checks if there is a direct line of attack from an opponent piece whether or not the opponent is allowed to move the piece. E.g., if White Pawn is in same row as Black Rook, the function must return Yes even if the Black Rook cannot actually be moved. For example, consider a situation where moving the Black Rook would expose the Black King to a check. In this case IsPieceUnderThreat() for the White Pawn is True. However, IsValidMove() for the Black Rook to the position of the White Pawn would return False.

## Namespace

---

Our grading code compares your implementation with ours. To avoid conflicting names of the two Chessboard classes, we use namespaces. All you need to do is:

Include "using Student::ChessBoard" once in each .cc file that implements it

```
using Student::ChessBoard;

ChessBoard::ChessBoard() {}
ChessBoard::~~ChessBoard() {}
```

## List and vector

---

You will use the List and Vector classes of C++.

For List, please be sure to review the use of iterators.

Further, you may find the remove\_if function, and the for(iterator it: list) code pattern useful.

```
l.remove_if([target_value](node* n){
    return n->value == target_value;
});
for(node* n : l)
```

For Vector, you may find the following functions useful:

```
std::vector<typename> v = std::vector<typename>(size, initial_value);
// returns error for out-of-bounds access, unlike array[x] which does not
v.at(5) = 7;
```

## Config file

---

The test file consists of the initial setup.

```
<number of points awarded to this test>
<number of rows> <number of columns>
<b: black, w: white> <r: rook, p: pawn, b: bishop> <row> <column>
```

Followed by a '~' and then the tests to be executed. Three possible tests may be invoked. We describe the tests below::

```
movePiece <current row> <current column> <target row> <target column>
```

This tests whether your code correctly handles an attempt to move a piece located at and to and . Note that the move may not be legal, and if so, your code should correctly handle it. If the move is legal, the state of the chess board after the move must be correct. Note that if this is a legal move, subsequent tests relate to the new state of the board (thus this step may affect the state of the board, and consequently impact the results of future steps).

#### isValidScan

Calls isValidMove() from every position to every other position on the chess board for all pieces. To pass this test, your code must produce the correct result for every move queried.

#### underThreatScan

Calls isPieceUnderThreat for all chess pieces. This test passes if your code returns the correct result for isPieceUnderThreat for every chess piece for that board state.

Below is a config file format.

```
0
4 4
b k 0 0
w k 2 0
b b 0 3
b p 1 2
~
// call isValid from every spot to every other spot
isValidScan
// call isPieceUnderThreat from every spot to every other spot
underThreatScan
// move white king diagonally
movePiece 2 0 3 1
// move black pawn two steps as it is in row 1 (starting row for black pawn)
movePiece 1 2 3 2
// move white king diagonally
movePiece 3 1 2 2
// move black bishop diagonally
movePiece 0 3 3 0
// invalid move as no piece exists at (0, 3) after the previous step
movePiece 0 3 1 0
```

```
// next round of isValid calls from every spot to every other spot
isValidScan
```

# Build

---

Please read the makefile to understand the list of commands.

```
make custom_tests : Compile using main.cc to run your own function calls
```

# Gradescope output

---

Example execution with error:

If your program has an error, you may see a result like shown below. The first picture depicts the state of the board at the start. The error line indicates that your code did not produce the expected result for line 13 of the config file. The final picture depicts the state of the board after line 13. [In the case there is no error, the final picture would show the state of the board after all the moves in the test file].

Initial state:

```
01234567
-----
0|      |
1|  ♖ ♖  |
2|   ♖ ♖  |
3|  ♖     ♜ |
4| ♜       |
5|       ♜  |
6|   ♙     |
7|     ♙    |
-----
```

A function call did not match expected output at line 13

Final state:

```
01234567
-----
0|      |
1|  ♖ ♖  |
2|   ♖ ♖  |
3|  ♖     ♜ |
4| ♜       |
5|       ♜  |
6|   ♙     |
7|     ♙    |
-----
```

# References

---

Articles that have inspired the design of this project.

[1] Yiran Zhong 2019 J. Phys.: Conf. Ser. 1195 012013.

[2] "P4: Inheritance and the game of chess," CS161 Programming Assignment 4: Inheritance and the game of chess. [Online]. Available:

[https://www.cs.colostate.edu/~cs161/Summer16/more\\_assignments/P4/P4.html](https://www.cs.colostate.edu/~cs161/Summer16/more_assignments/P4/P4.html) [Accessed: 22-Aug-2022].