Instituto Tecnológico y de Estudios Superiores de Monterrey Campus Querétaro

Object oriented programing 302

Integrative project:

Presents:
José Ricardo Rosales Castañeda | A01709449

Querétaro, Querétaro June, 2022

Index

Introduction

The problem situation I selected for the integrative project consists of implementing a 2 player chess game or one vs one (traditional), where, the board is represented by an 8 x 8 array, where the white squares are represented by the “-” character and the black squares represented by the “ ” character, the users or player will take turns until a winner is drawn by checkmate or boards.

The classes consist of one abstract, that represents a chess piece, and 6 more subclasses, each one representing one chess piece (rook, pawn, queen, king, bishop and knight), each piece will have his type and color (black/white), a board class and a game class, which is where the game will be implemented and executed.

As usually the white pieces start the game, the turns consist of inputting the coordinates, letter from a to h to select columns and numbers from 1 to 8 to select row, of the piece you want to move and then inputting the coordinates of the square where you want your piece to move, example: move d2 to d3, and every turn must be validated certain things:

- Inputting a number within the 8 x 8 array or board.

- Validate if the square where you want to move your piece is free or not and if the move you want to make is legal, example: the move in shape of an “L” of the knight, if not then another square must be inputted.

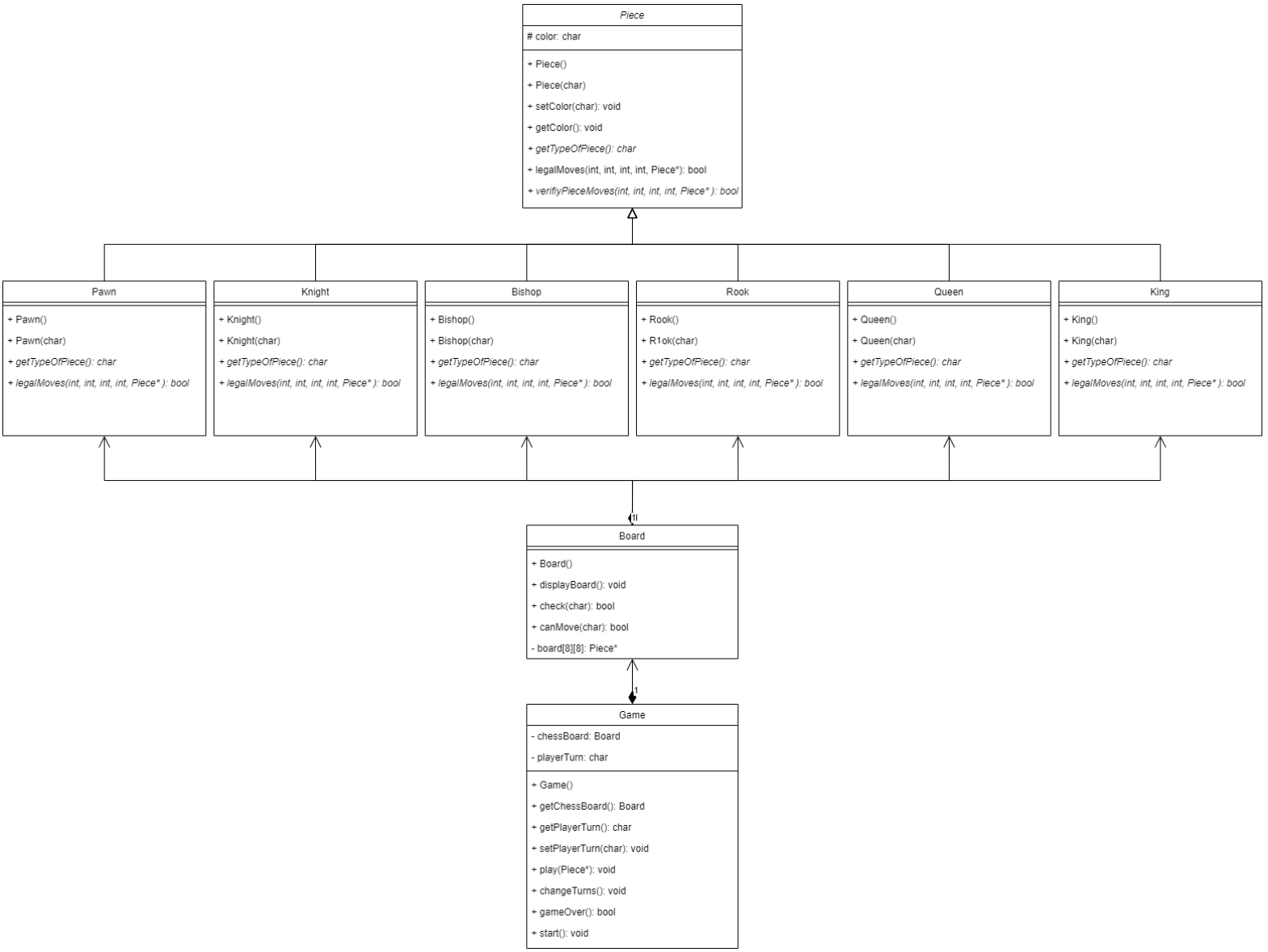
- Validate if a pawn has reached coronation in order for the player to choose between a queen, rook, bishop or knight to add to the board.

- Verify if the Rooks and king are in position for the castle

- Validate if the king is in check and cannot make any more moves or by declaring boards, which is when the king can't move but is not in check.

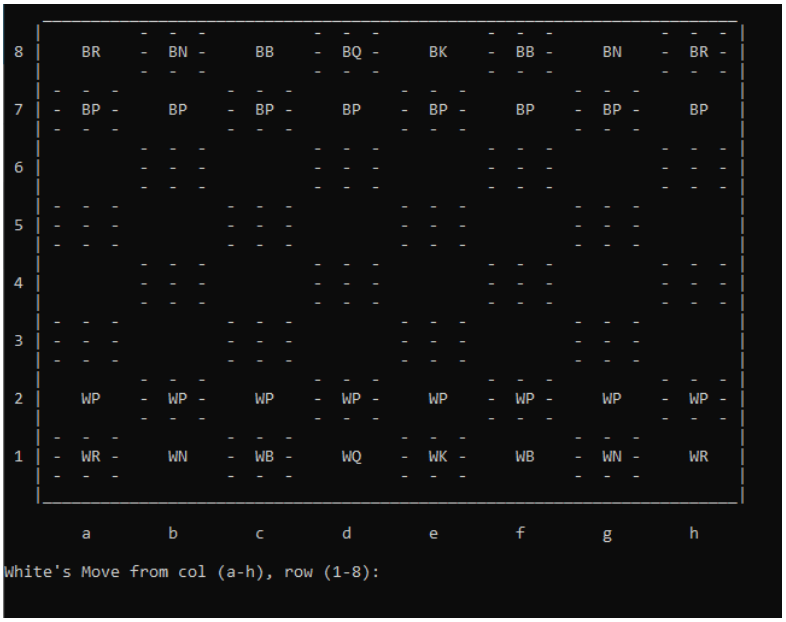
In this case I decided to use the class Game as the main, which is why I implemented the main method in this class because if not I would have to create another file for the main that is only 4 lines long.

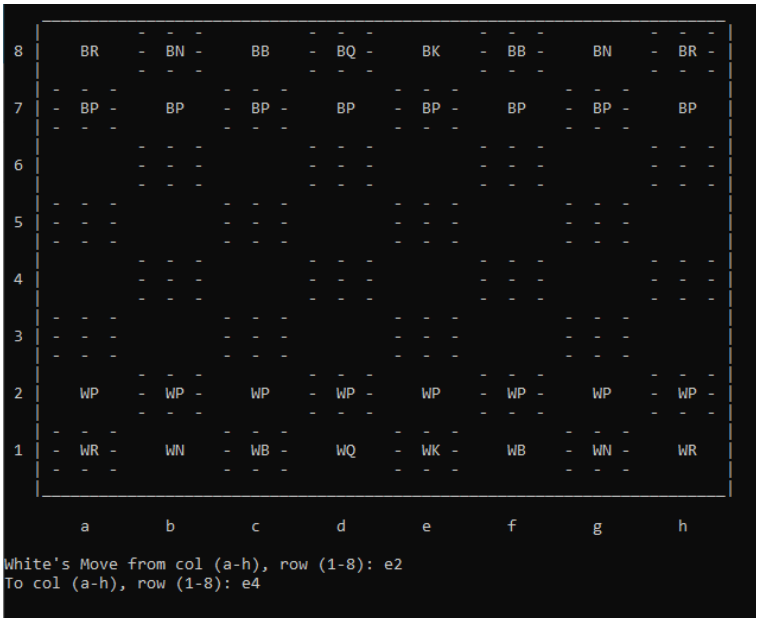
UML class diagram:

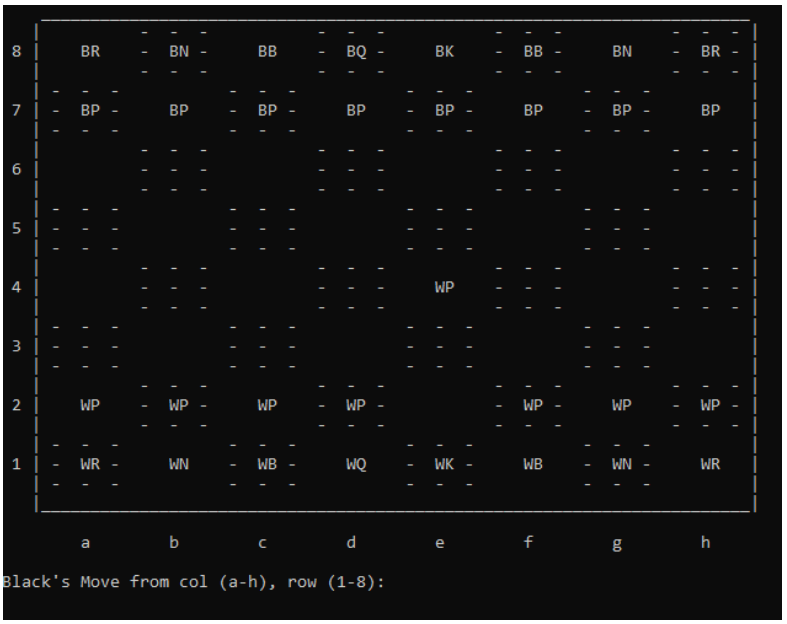


Decided for this design, where all the chess pieces derive from a parent class, which functions as an abstract class, from which they inherit its attributes, methods and perform polymorphism by overriding methods and overloading operators, then all the chess pieces have a composition relation with the Board class, decided to do this in this way because the objects instantiated from the chess pieces classes are created in the Board class, and finally the main class Game, where all the game and classes are implemented including the main method.

Execution example:







Argumentation:

a) Classes: for this project I decided to create 9 classes, one abstract class which is the Piece class, will help to create other classes, 6 classes which derive from Piece class and represent the traditional pieces in chess: Paw, Knight, Bishop, Rook, Queen and King, one class for the displaying the board with its pieces and implement principal logic from pieces movements, and one class Game in which all the game its implemented and executed.

b) Inheritance: Inheritance is implemented by a child parent relation between the abstract class PIece and the 6 derived classes Chess pieces.

c) Access modifiers: In this project the access modifiers were implemented or called when the creating an inheritance relation among classes, in this case all de child classes have a public access to parent class and also when declaring methods and attributes from each class, where promoting encapsulation by defining attributes as private, protected in case of parent classes, and methods as public.

d) Method overriding: method overriding is implemented in this project when getting the type of piece and in the method used to verify if the type of piece can make a certain movement, like the move in form of an L from the knight, this was made like this in order to make the code more clean and not having a lot of methods to verify every move in every piece.

e) Polymorphism: polymorphism in this project is made when overriding a method in order to make one method make multiple things in different objects, and in operator overloading, that in this case was used to print in the board at the same time the color and type of piece of a piece in the board, in order to identify pieces and instead of using two couts to print first de color and then the type of piece now it only uses one cout that receives a pointer of type Piece and returns color and type.

f) Abstract class: In this case the abstract class is the parent class Piece that represents a normal piece in chess with default color black and no type of piece, this in order to have kind of a mold to create the other classes of the chess pieces.

g) Operator overloading: within this project the operator u decided to overload was the “<<” operator, in order to print at the same time the type of piece and color of the piece when displaying the board and pieces, i decided to overload this operator because with this instead of using two lines of code, one for the color and one for the type, I only use one and also because U at first tried to overload the “==” operator but I couldn't get it to work.

Special cases:

In this case I didn’t identify any special cases which would make the program malfunction.

The validation made for the movement of the piece are this ones:

Piece: Check if the move made by the player is legal, define a destination using a Piece object pointer that holds destination row and column, then used an if to check if the destination square is occupied this by making occupied squares equal to one, and checking if there's an enemy piece, if there is an enemy piece you get to eat it if there's else if the square is occupied, it return the method to check if according to the type of piece the move is legal, for example the movement in form of an L of the knight.

Pawn: Check if the pawn is able to move, in this case for making it simple, don't allow a 2 square move at the beginning, then using an if to check if the desired destination is empty, empty = 0, if it's empty make the move, by checking if the move is ade in the same col, this because pawn can only move in the same col, except when eating, and defining the desired dest as the origin +- 1, depending if it is a white or black piece(going up or going down).

Knight: Check if the move made or desired by the player is in a col upper or lower(+-1) from the original col if it is then the destination row is defined by moving from de origin row 2 rows(+-2) and if it's the opposite(2 col and 1 row) made the move, validating that it can only make a move in shape of an L like in classic chess.

Bishop: Check if the way of the bishop is free(diagonal) by, depending if the piece is white or black(up or down) calculates a row and col offset to check each square a t a time and will also help to track the path diagonally from the origin all the way to the desired destination, allowing only diagonals of the color of the origin square (white or black in this case "-" or " ") and allowing to eat(replace) opposite pieces.

Rook: First check if the destination proposed by the player is in the same col than in the origin, using a col offset and a row offset, this to move square bys square all the way to the desired destination, this offset depending if it is a black or white piece will be +-1(up or down), in this case using a for cycle it will check if that col is free to move and if it is made the move in a straight line across columns, but if the columns change and the row are the same, then it will check if that row is free and made the move but across rows.

Queen: In this case I realized that the piece of the Queen in chess is kind of a mix between the bishop (diagonal movement) and the rook (straight line movement) so I decided to implement the same code from the bishop and the rook.

King; In this case the King piece is very similar to the Queen piece, in terms of the path he makes when he moves, the only difference is that the king piece can only moves 1 square at a time, so here we use a variable to know if the desired destination is one square away from the origin, this by making a subtraction between origin col and row, and destination col and row, validating if this is equal bigger or smaller than 1 and 1 also allow castle if possible.

Personal conclusión:

I decided for the final project to create a chess game because I enjoy playing chess and programming a lot, so mixing these two things in one project really made me enjoy it and made me understand and implement better the new concepts of OOP we saw in this class, like inheritance and polymorphism.

References:

E. (2019b, marzo 25). *Chess Notation - The Language of the Game*. Chess.Com. Recuperado 11 de junio de 2022, de https://www.chess.com/article/view/chess-notation

GeeksforGeeks. (2022, 21 mayo). *Operator Overloading in C++*. Recuperado 11 de junio de 2022, de https://www.geeksforgeeks.org/operator-overloading-c/

GeeksforGeeks. (2021, 28 junio). *Pointers in C/C++ with Examples*. Recuperado 11 de junio de 2022, de https://www.geeksforgeeks.org/pointers-c-examples/