**Introduction**

Chess is one of the oldest boardgames that is still played today. It is a very easy game to pick up, but is notoriously difficult to master. There are black and white pieces that the players control. Although there are several ways a game can end, you can only win by one condition: checkmate. In order to obtain checkmate, you must directly attack the opposing king (called check), whilst making sure the king has no other moves to play (blocking the check, evading the check, eliminating the attacker). Through tactics and strategy, your goal in chess is to obtain checkmate, whether it would be by eliminating their pieces directly or ensnaring them in traps.

I chose to implement the chess board game because it posed a huge hurdle to me. Before this, the most complicated program I’ve coded was tic tac toe. Instead of 9 tiles, there were now 64 on the board. In addition to the sheer size of the board, there was also new pieces to consider. There are 6 unique pieces in chess with distinctly different movements and conditions governing their movement. As an aspiring software engineer, I also wanted to familiarize myself with breaking down a concept of this size into functions, algorithms, and structures.

I do not particularly find the implementation of chess to be an important concept or idea, after all, it has been implemented before. What was most important to me when deciding to implement chess in code was the sheer organization one would need to implement the program. From a software engineering perspective, this was a good choice for me to strengthen my thought process on converting tangible movements to computer logic. Logically, it would seem that programming chess in an object-oriented fashion would be the best approach to programming a chess game.

**Summary and Remarks about Changes**

This project took me about two weeks total to refactor the original functional program to an object-oriented approach. I chose to remove some features from the old program including a 50 move stalemate as well as the optional resignation and offering a draw to the opponent. I have also removed the MVC design pattern. Most of the functions from functional programming version of chess were moved over to become public member function of a class. I opted to create a class for the piece, board, and game. The piece class was responsible for handling the movement checking of the game. The board class was responsible for holding information such as the symbol, whether or not is it currently occupied, what color is the piece that is occupying, and tiles that are “under attack” by other pieces. The game class was mostly a class to hold the board as well as the myriad of functions for verification and checking certain booleans. There were certain ways I could have optimized the code better. I had to overload several functions, including the updateTiles function so I could pass the copy board. Another major change includes splitting the VPMCheckMovement from the previous version in order to establish polymorphic behavior in the pieces.

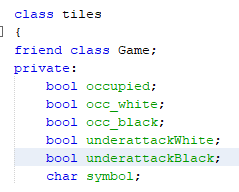
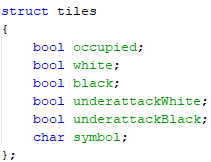
**Basics of Chess**

The rules of chess can be found virtually anywhere on the internet, but for the purpose of readability, I will quickly go over the most basic movements, as well as the more specific rules.

* Each side has 8 pawns, and 8 “special” pieces. The 8 special pieces are made up of: 2 knights, two bishops, two rooks, a queen, and a king.
* The knight can move either two tiles up/down and one tile to the left or right or two tiles to the left/right and one tile up or down
* The bishops can move on any diagonal tile given that it is not blocked by another piece
* The rooks can move on any horizontal or vertical tiles given that it is not blocked by another piece. If a rook moves, castling is unallowed
* The king can only move one tile at a time. The tile must not put him in check. If a king moves, then castling is unallowed. All tiles between the king and rook must be empty as well as not place the king in check. Castling is only allowed to be done once for each color, on either the “queenside” or “kingside”. The king can **NEVER** be captured.
* The queen is the most versatile piece, it can move diagonally, horizontally, vertically, or to any surrounding tile.
* The pawn can only move forward if its intent is not to capture. It can only move forward diagonally one tile if its intent is to capture. The pawn can move forward twice, but only on its first move. And a pawn can capture an opposing pawn on an adjacent tile by going behind it in a move called “enpassant”. This move is only valid if the adjacent pawn just moved forward two tiles. If the pawn reaches the opposite end of the board, it must be promoted to a special piece besides the king.
* The goal of the game is checkmate; however, a game can end in a draw via stalemate, insufficient material, the 50-move rule, or a threefold repetition. I cover these end conditions in the description section.

**Description of Classes**

**Tiles**

 My first approach to tackling the problem of refactoring my code to an object-oriented approach was moving variables to their respective places. The easiest part of the code to refactor was the tiles struct from the earlier version. Below are the previous(left) and updated(right).

This structure is the basis for my board. The board’s tile has 6 data types, most of them being Booleans.

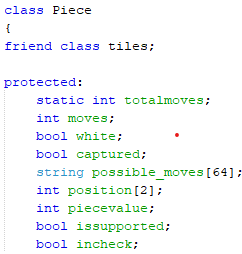
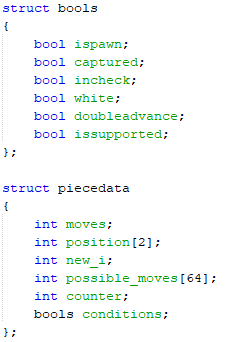
The occupied Boolean governs whether or not the tile is being occupied by a piece, whether white or black. This is necessary to implement because it allows the computer later on to recognize if a piece is being blocked in their movement path. The occupied Boolean also helps us later on when we need to “scan” the board for pieces.

The white and black Booleans determine whether or not the tile is occupied by a white or black piece. This helps us determine capture, whether or not a movement is valid since you can capture opposing pieces is valid, but capturing your own is invalid. These three Booleans are all initialized and checked for simultaneously, therefore if a tile is not occupied, then white and black are false.

The next two Booleans determine whether or not a tile can be occupied by a white or black piece in the next turn. These Booleans help us determine whether or not a king can move to a tile given that in chess, Kings cannot move “into check”.

The last data type is the character symbol and just stores the symbol to be displayed by the displayBoard function, either a piece character, or a space, signifying an empty tile.

**Pieces**

The pieces were not as straightforward as the tiles in terms of refactoring. The pieces had to be modified so that it is universal across all pieces. We have to have this because the Piece class is actually a base class. The derived class pieces also cannot access their own member functions because the pieces are represented as base class pointers in the program. Below are the old (left) and new (right) representation of the pieces and their respective attributes.

* The captured Boolean tells us if the piece is captured. This is important later on because it will be a filter for our logic, allowing us to differentiate working with pieces that are alive and not working with pieces that have been captured.
* The incheck Boolean is only relevant for the kings and tells us if the king is under attack (therefore allowing us to see if the move is valid or not). Doubleadvance is a Boolean that is relevant only for the pawns and the king. It will initially be initialized to true, telling the program that the kings(castling) and pawns can move twice in one turn.
* The final Boolean is the issupported Boolean which tells us if a piece is supported by another piece of the same color, which we use later to determine if a king can capture its attacker.

The integer data types for pieces are essential in describing numeric values of the piece’s data

* The first integer is moves, which keeps tracks of all moves made by the respective piece. This helps us determine castling conditions, as well as the pawn’s first movement.
* The second integer is an array of two integers, which stores the position of the piece in rows and columns.
* The possible\_moves integer array is responsible for determining all the possible moves a piece can make. There are 64 elements in the array representing the 64 tiles on the board. If a move is invalid, the element is declared to -1. This datatype was made for the purpose of checking whether or not a move can be made to block checks. This is also, personally, the hardest concept to implement in the program.

I had to remove the ispawn boolean as well as doubleadvance because not every piece is a pawn or advance twice like a pawn. Although I could have had the boolean set to false for non-pawn pieces, I opted for a better solution. My previous reasoning for having this boolean was mainly for pawn promotion, enpassant, double advance. However, I was able to get around all of that with the following:

* Pawn promotion: instead of checking is a piece is a pawn and if they’re in the eighth rank, the OOP approach just checks if the pieces in the vector sub indices 16-32 are in the eight rank.
* Enpassant: instead of checking for double advance, I simply checked if the tiles behind the enpassant square was occupied by the opposite color, the piece on that tile had the piece value of 0 (pawn), that it had only moved once, and if it was on the same row for a double advance.
* Pawn promotion: surprisingly, this was not that hard to get around. I used the same function and idea of the previous non-OOP chess. The function would check for pawns in the eighth rank and then modify the piece value accordingly. Then, whenever the pawn is called to check for movement, it would first check if the piecevalue data had been modified, thus indicating a pawn promotion. Then, it would call the respective checkMovement in accordance to the new piece (ex. The piecevalue represents a rook (5), so it would then call checkMovement for a rook piece and check if the movement was valid.

**Enumerations**

This enumeration is used for the blackPieces and whitePieces array, allowing us to easily access the array element which contains the symbol for the respective piece. These values are also what we use for the new piecevalue integer.

**Important and Recurring Variables**

|  |  |  |  |
| --- | --- | --- | --- |
| TYPE | NAME | DESCRIPTION | LOCATION |
| tiles | board[8][8] | Contains all the information on tiles | In model functions |
| piecedata | piece[] | Contains all the information for each piece | In model functions |
| string | playermove | Contains the alphanumeric and pure numeric coordinates of the player’s starting and ending move | In model and controller functions |
| bool | whiteturn | Tells the program who’s turn it is | In model functions |
| int | startmoverow | The integer row value of the piece selected to move. A derivation of playermove[] | In model functions |
| int | startmovecol | The integer column value of the piece selected to move. A derivation of playermove[] | In model functions |
| int | endmoverow | The integer row value of the tile the player wants to move the piece to | In model functions |
| int | endmovecol | The integer column value of the tile the player wants to move the piece to | In model functions |
| char | whitePieces[] | Has the character values which dictate the symbol of the white pieces | In view, model, and controller functions |
| char | blackPieces[] | Has the character values which dictate the symbol of the black pieces. | In view, model, and controller functions |
| bool | simulation | Determines whether or not the model is in “simulation mode” | in model functions |

**Important Functions and Explanations**

|  |  |  |  |
| --- | --- | --- | --- |
| TYPE | NAME | LOCATION | EXPLANATION |
| void | initializeBoardPieces | Game.h | Initializes the board symbols and the Pieces. Calls the respective constructors to set the default values and position of the pieces. |
| Void | updateTiles(5 parameters) | Game.h | Overloaded updateTiles function used for simulation |
| Void | updateTiles(4 parameters) | Game.h | Overloaded updateTiles function used to actually update the attributes of the current instance of the game |
| Void | updateTiles(1 parameter) | Game.h | Only called once during the initialization to set the occupied tiles and tiles in check etc. |
| Bool | CheckMovement | Game.h | Calls the checkMovement for the respective piece. |
| Bool | kingNotinCheck | Game.h | Checks if the king will be in check if the move goes through. Performs a simulation to accomplish this. |
| string | movePossible | Game.h | Checks if a piece can move to a tile on the board by checking the movement to every tile on the board. |
| Void | updateGameState | Main.cpp | Checks if the game is over via stalemate, checkmate, or insufficient material. |
| bool | checkMovement | Piece.h | Checks if a piece can move to a certain location. |
| Bool | checkBlock | Piece.h | Subset of the checkMovement function. Simply checks if the tiles are blocked for movement for the rook, bishop, and queen. Every other piece is checked manually (hardcoded) |

**The Conversion from non-OOP chess to OOP chess**

In this section, I will go over some of the major hurdles from refactoring the code. One of the hurdles I encountered was refactoring the simulation. The original updateTiles function from the non-OOP chess had tiles as an argument. Since this version’s updateTiles had direct access to the board (as per being a member of the Game class), it would use that. However, this became a problem when running a simulation because I used a copy of the board as well as the pieces. However, these copies were not in the Game class and were initialized in the function itself. So, I had to create an updateTiles that could accept an array of tiles (the board). This actually made the code a lot easier to work with.

Another problem I had with the refactoring of the code was the pieces. I had previously used an array of structures in order to keep all the data about the pieces, however in this iteration, I wanted to use polymorphism and derived classes for the pieces. At first, I wanted the piece class to be abstract, however, this proved to be a problem when I wouldn’t be able to access the member function. I wanted to keep all the pieces in a single array in order to make rewriting the function parameters easier. The only way to do that was with an array (or vector) of base class pointers. To counteract this while fulfilling the requirements of the project, I created a vector of base class pointers and used the virtual keyword to overwrite the checkMovement function (since each piece’s movement is different).

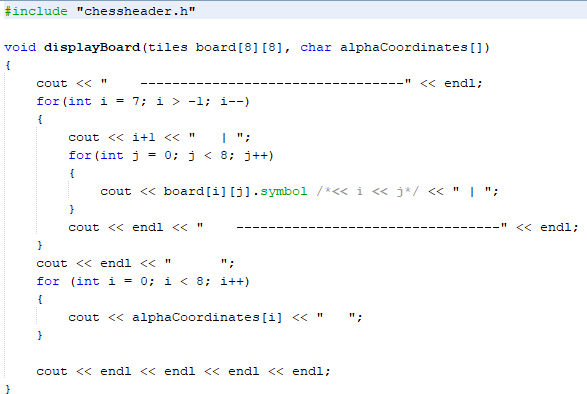
A new feature I added that was not in the previous version was the template. Instead of using an array of strings to store the information, I instead opted for a template class so I can store both the string and the integer representation of the player’s moves.

Those were main changes I had to make to the structure of the program. The rest of the functions are mostly unimportant as they don’t present any unique algorithms most people have seen. Most of the functions that I won’t cover are usually for determining some booleans, such as is the piece white, flipping the turn, checking if a piece can attack another piece and etc. The main backbone of this project is checkMovement functions as well as the updateTiles function. These functions are repeatedly used in other functions for determining the validity of moves as well as updating most of the conditions on the board as well as pieces.

**Basic Explanation of my Implementation of Chess and Some Algorithms**

In this section, I will display the code for the most notable functions and explain my thought process on creating them. Now that I have covered the main variables used in the program, the first order of business was to initialize the symbols and create the displayBoard function

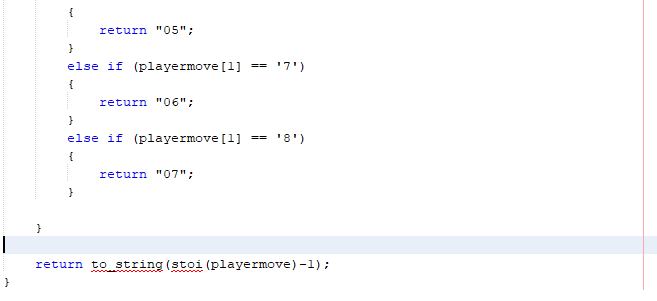
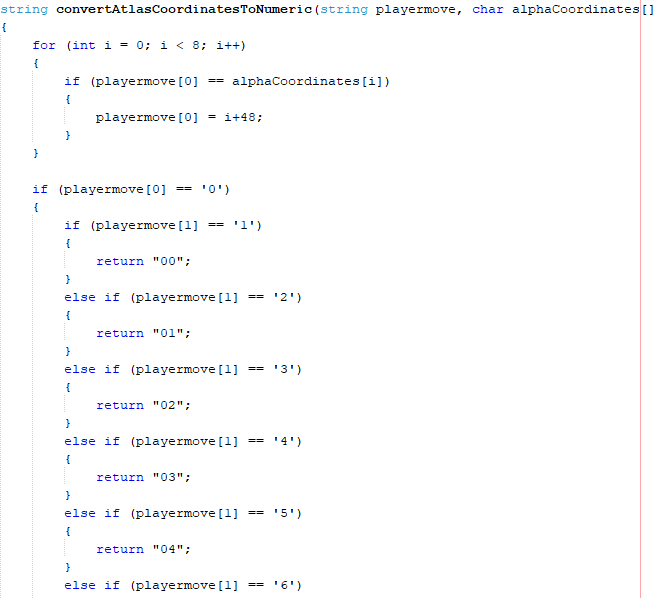
**The Display Board Function**



This function is a simple modified nested for loop that is fairly generic for printing out the contents of a 2d array. The first thing to notice about this function is that instead of incrementing the first for loop, there is a decrement operator. I did this because the chess board has its rows incrementing upwards. It’s necessary to do this for our display and it also doesn’t affect us because the board array is still called by [row][column]. At the end of the function, I output the alpha coordinates for the user to easily read the tiles.

**Obtaining and Verifying User Input**

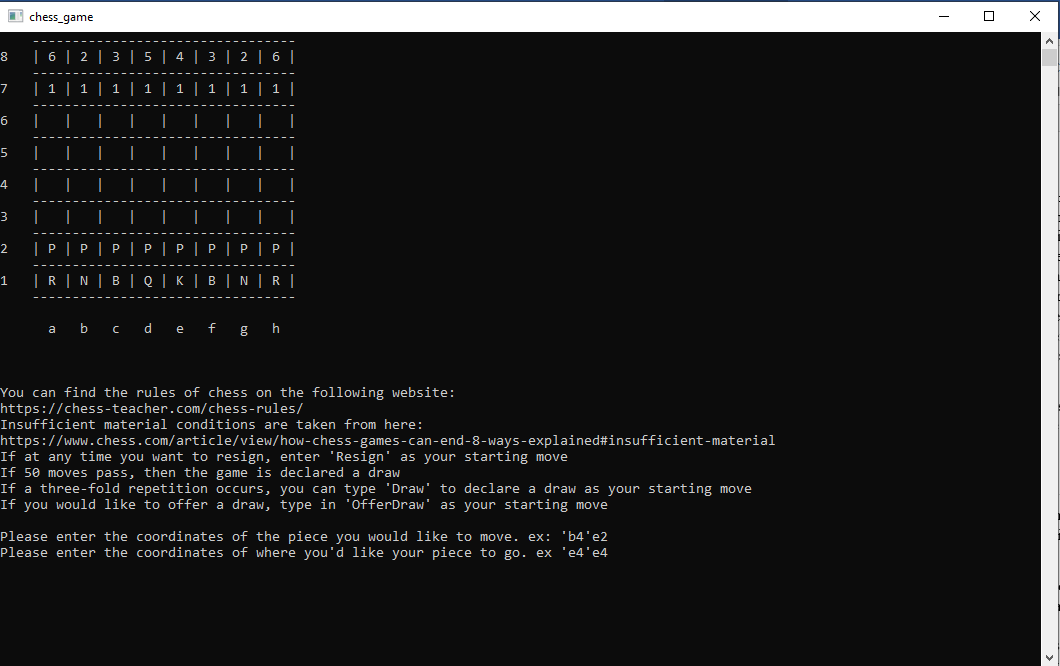
The user input in this program is twofold. Each player must input two coordinates. The first input takes the coordinate of the piece they want to move. The second one will tell the program where the player wants to move the piece they had selected. In the earliest versions of my program, I wanted to verify the user input after I obtained them. Therefore, I would have two functions for getting input and two functions for verifying the input per move. The player moves are stored in a four-element string array. The first and third element contained the start and end moves in alphanumeric notation respectively, while the second and third contained the pure numeric forms of the coordinates in their respective previous elements. The conversions from alphanumeric to pure numeric is demonstrated in the following function:



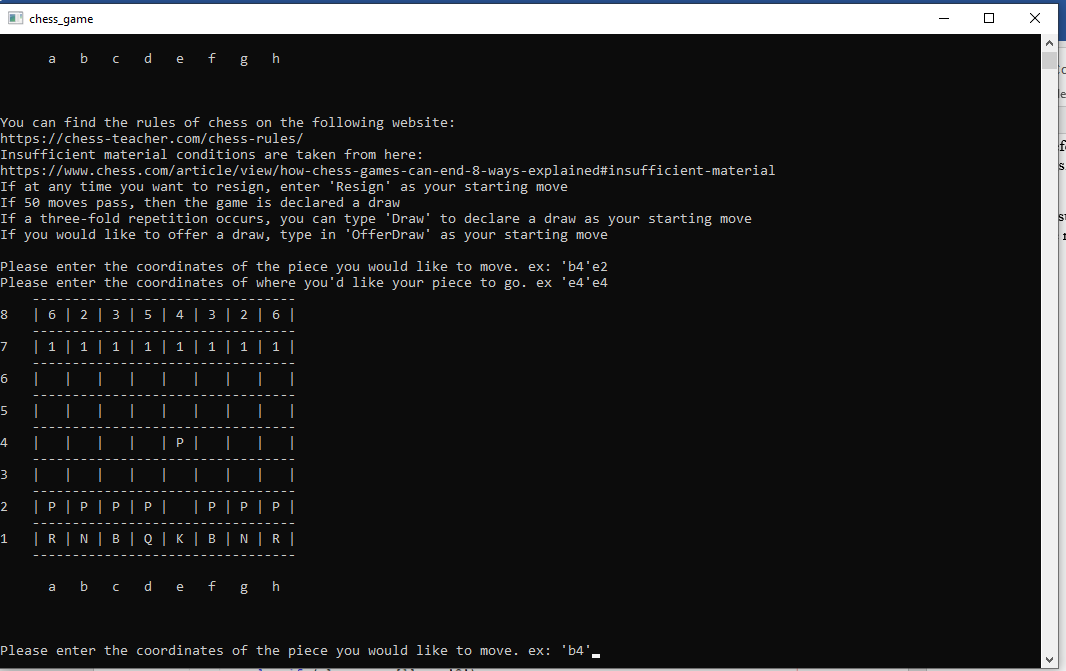
This function will simply go through a predefined c\_ string array of the alphabet coordinates which indicated the columns on the chess board. When it finds the respective letter that matches, it converts the value of the alpha coordinate to a numeric representation that is still contained as a character data type. The long if if-else if branch is my solution to the following problem: Whenever a coordinate that is preceded with ‘a’, the program return statement will return the value of the string to integer representation minus 1. The return statement must convert the string to an integer temporarily in order to subtract 1 (since the row values are off by 1). However, the string to integer conversion of anything preceded by ‘0’ truncates the ‘0’. Therefore, an earlier return is necessary to prevent the function from returning single character strings.

This conversion is performed on both the player start and player end moves, and return the result directly to the corresponding element which should contain the pure numeric notation of the moves.

This is what an input would look like



And here is the result



In my initial designs, the verify player move end function was responsible for checking if the movement of the piece was valid. I later found out that this was a bad idea because the input validation would keep the player inputting a “end move” until it was valid. Since the player could not return to the start move, players could possibly end up in a dead end where although they selected a piece that was theirs, that piece could have no possible moves (ex. King is in check and must be defended immediately). In my updated version, I chose to have my two input verification functions to solely cover if:

* The piece selected was their piece
* The coordinates they chose contained a piece
* Their coordinates were in the correct format

Later, there is a do-while loop that verifies the actual movements, check conditions, and capture conditions of the player move which encompasses the start and end steps, allowing the user to choose a different piece if the move is invalid.

**Verifying Player Movement**

Easily the most difficult yet crucial portion of this code is the verify player movement function. This function is the basis of the code. It is used repeatedly throughout the code, and is the centerpiece of another important section which I will cover later: simulations. The purpose of the function is as follows:

* The movement must be valid (ex. Bishops only moving diagonally)
* The capture must be valid (not capturing your own piece. Checking pawn capture conditions)
* The king must not be in check after

The first order of business was to check if the movement was valid, which was a lot harder to conceptualize than to implement.

I chose to have the function take in the direct integer conversion of the player start and player end moves. They are declared as startmoverow, startmovecol, endmoverow, and endmovecol. I chose this because I would primarily be working with the integer conversion anyways and it also allows for the function to check any player movement, without constant conversion from string to integer. This is important later when we work with simulations that work with 10,000+ operations. All movement is determined by the position of the piece in the base class.

**Pawn Movement**

Pawns are one of the trickiest pieces (only second to the king) to code in the chess game. They can only move forward if they do not intend to capture a piece. However, they can move forward and diagonally one tile if they do intend to capture a piece. Furthermore, they can move twice on their first move and enpassant adjacent tiles that move twice in one turn, but only on the next turn.

The most basic movements can be done by subtracting the end integer values from the beginning. Doing so will tell us the vertical and horizontal movement of the piece. From here, it is just a matter of implementing lots of if statements to see if the piece movement is correct. It also important to note that a black pawn “forward” movement is technically negative, since the board is flipped from the computer’s “perspective”.

To check if the pawn move is blocked, I can simply have an if statement that checks if the tiles in front of it are occupied using the bools struct. Harding coding the blocked conditions are okay for the pawn since it only moves at most two tiles. Later we will use a blocked function for the queen, bishop, and rooks.

Diagonal movements are only allowed if board at the endmoverow and endmovecol are occupied and occupied by the opposite color.

The last important thing to implement is pawn promotion. Since the movement of a piece is dictated by its position in the array, I wanted to find a way to change the value of the position without changing its actual position. I did this with the use of the piecevalue variable covered earlier. The piecevalue variable is initially initialized to the respective piece value in the enumeration, but when a pawn promotion occurs, it is initialized to the piecevalue of the piece it is promoted to. Then if a program detects that a pawn derived class has a piecevalue that is not of a pawn, it reroutes the checkMovement to the respective class it was promoted to.

**Bishop Movement**

The bishop movement is defined as moving on a diagonal square. There is no difference in movement between the white and black bishop as they both can move in four diagonal directions. In real chess, there is a distinction to be made by the black square and white square bishop, however, for the purposes of this chess program, they are unnecessary to implement.

The tile validation is simply done by making sure that the absolute vertical movement is equal to the absolute horizontal movement.

There is no additional validation besides blocked tiles, which we will cover in a later section.

**Knight Movement**

The easiest movement to make, the knight can only move in a maximum of eight different tiles. The knights also have a unique ability of jumping over pieces, meaning we do not need to check if the movement is blocked. Coding this is a matter of checking is the absolute value of its horizontal/vertical movement is one or two, but making sure the other is the opposite.

**King Movement**

The king’s movement is relatively simple. He can only move one tile as a time, unless he is castling. The single tile movement is easily programmed in three if statements. The castling is more difficult, but can be easily hard coded in the if statements. The if statements will simply check if the tiles are “underAttack” by the opposite color, if the king has moved, if the rook has moved, and if the tiles in between the king and rook are empty. Kingside and queenside castling is broken up in two if statements and is determined by whether or not the horizontal movement is positive or negative.

**Rook Movement**

The rook’s movement is simply horizontal and vertical tiles. To code this, we make sure that either the vertical movement or horizontal movement is zero, then you check if the piece is blocked.

**Queen Movement**

The queen’s movement is most diverse in the game, but the easiest to code, especially if you already have rook and bishop movements done. The queen is essentially, the king, rook, and bishop combined. So, coding her is just simply copy pasting the bishop, rook, and king movements. Of course, we don’t copy over the castling checks.

**Checking if a Piece is Blocked**

The function we will be covering in this section is the isBlocked. The blocked function only applies to rooks, bishops, and the queen. The knight cannot be blocked, and the pawn and king are already hardcoded. The blocked function is unique for each piece that it was implemented in. For the rooks and bishops, and consequently queen, we must know which diagonal “path” or horizontal/vertical “path” the piece is taking. This is done by subtracting the end move row and end move col from the start move row and start move col. Then, we check both values to see if they are negative in positive. After that a while loop is implemented, checked the next tile in front of it to see if the tile is occupied. The while loop will then increment the row and col appropriately depending on the “path” the piece is taking. The while loop must stop on the tile right before the destination, because the destination can be captured, and does not “block” the piece.

**Checking Capture Validity**

In the verify player move function, you might notice that that boolean checkpoint that governs the capture validity is set to true, rather than false like the rest of the checkpoints. This is done so that a piece can essentially move to any tile (remember that we don’t check the destination tile in the isBlocked function). However, if the destination tile is occupied, then the CheckCapture function is called. The check capture function will simply just check if the piece is capturing a piece of the opposite color. If it is not, then the checkpoint will be set to false.

**Model: Updating Tiles**

By far the most crucial and important function in this entire program. The updating Tiles function is responsible for editing the pieces as well as the board. The function first starts by checking if there is a piece in the destination tile, which would indicate a capture. If a capture is true, then the piece will be updated to lose its position, as well as be marked as captured. This is important because the pieceSearchArrayPosition depends on piece array being updated for captured pieces. Since there are no Booleans to keep track of whether or not castling or an enpassant is taking place, I do have to reuse code that I’ve used in the CheckMovement function in order to update the pieces accordingly. Also, since the enpassant capture isn’t on the destination tile but the one behind it, we must adjust for that as well. The update tiles function is also responsible for keeping track of the moves, new position, and watching for pawn promotions Then, the board is rescanned and all tiles under check are recorded into the tiles class.

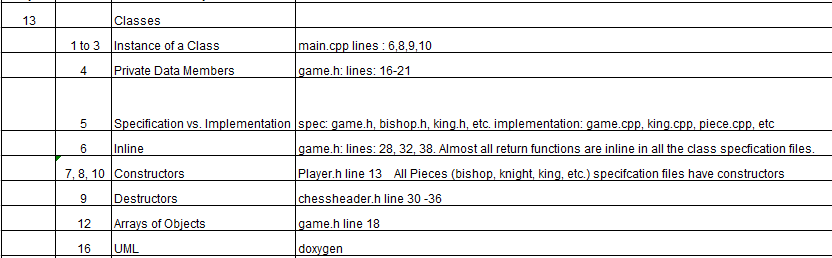
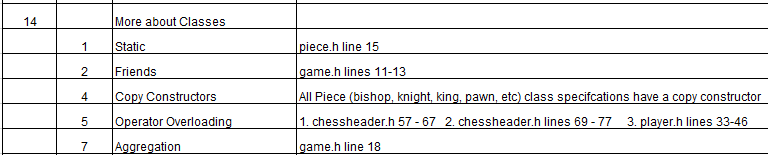
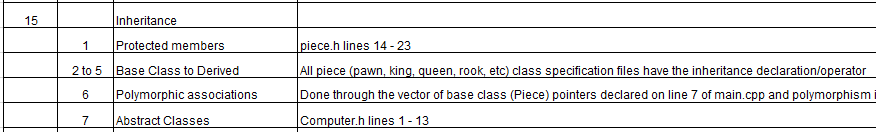
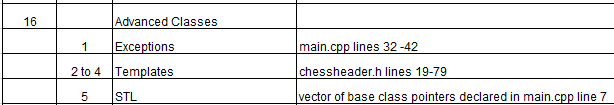
**Simulation Mode**

A problem that occurs when updating models after a move is executed is that you cannot do checks for discovered checks and piece that are supported by their own. An example of pieces supported by their own would be a bishop and a queen on the same diagonal file. Since the piece cannot technically capture their own color, they are not marked as “in check” or under attack. The solution to this is to move whatever piece you want to a valid tile, THEN recheck the board for validity. This is what the simulation mode does and tries to solve. The inclusion of the simulation boolean tells the model some things: that we must create a copy of a board and pieces structures and to keep the move separate from the real board. Our copyTiles function will return the board at a specific tile, and will loop until the whole board is copied. Then, we call our update tiles function but pass only the copied pieces and tiles arrays, so that our simulation does not override the original piece and board. The simulation boolean is also used later in the generatePossibleMoves function. Simulation mode is also utilized when we are generating all the possible moves. We must generate all possible moves for two reasons. The first one is to check for checkmates. It was hard to conceptualize the “blocking” of check. Instead, I allowed the player to figure that out and let the computer check the possible moves to see if there was an error. The second reason is to check for stalemate. It was easier to have an if statement checking all possible moves in order to check for stalemate.

**End Conditions**

The gameEnd function checks for all end game conditions. Checkmate is only true when: the king is in check, there are no possible moves that bring him out of check. These two conditions are checked in the isCheckmate function which will search if all the possible moves a king can make would still put him in check. Another important thing is to make sure that a king cannot capture a piece that is guarded by another piece. As I mentioned earlier, this was not an easy problem to debug. I found out that the isCheck function was not detecting that the piece was under attack by white because of the check Capture and well as the isBlocked function. In order to remedy this, I modified the isBlocked function, and create a whole new function, the findSupports function.

**Concepts Used**

****

**Debugger**

Although it is not very relevant with the explanation of this project, I’d like to document my experience with the gdb debugger. I found that tracking the chess program is ridiculously complex. In the previous version, I would be able to at least go through the individual function files. However, in the object-oriented version, there was now inline function, and separate classes, and certain classes can access certain functions and others could not. It all became too much to debug the program without help. So, I took up to learning how to use a debugger. The only thing I can think about now is why I did not learn earlier. The concept of breakpoints and having the program highlight the exact line you’re on as well as having a list of all the local variables helped tremendously when debugging the program. Here are some funny bugs I found that are worth noting.

* Pawns can promote, but they do not move like the piece it promotes to.
  + The bug was that I was passing piecevalue as the sub index of the array of pieces, which are not the same.
  + Took 15 minutes to debug
* Checkmate happened randomly
  + The generate moves function would send the wrong start and end integer representations because I mixed up the i and j iterators. This would lead the computer to producing a false list of possible moves.
  + Took 5 hours to debug
* King would always be in check when not really in check in rare occasions
  + The function responsible for checking if the king is in check would send the actual vector of pieces, rather than the copy pieces, because I had forgot to prefix “copy”
  + Took 2 hours to debug
* A pawn would enpassant, then proceed to crash the whole program.
  + The part of the code that checked for enpassant had mixed up the i and j iterators and would send trash values to the pieceSearchArrayPosition function. Of course, the function returned -1 and then it would try to access an array of pieces at sub index -1, causing a fault.
  + Took 3 hours to debug.
* Program would not end despite it being checkmate.
  + The updateGameState function did not have reference access to the game class
  + Took 1 hour to debug.

**Class Specifications and Header Files**

#ifndef CHESSHEADER\_H

#define CHESSHEADER\_H

#include <bits/stdc++.h>

using namespace std;

#include "Piece.h"

#include "tiles.h"

#include "Knight.h"

#include "Pawn.h"

#include "Bishop.h"

#include "Queen.h"

#include "Rook.h"

#include "King.h"

#include "Player.h"

#include "Game.h"

void updateGameState(bool whiteturn, Game& game, vector<Piece\*>& piece);

int pieceSearchArrayPosition(int row, int col, vector<Piece\*>& piece);

bool verifyPlayerMove(Game game, Player player, vector <Piece\*>& piece, bool whiteturn, bool hypo, bool simulation, bool realtimewhiteturn);

enum piecevalue {PAWN = 0, KNIGHT = 1, BISHOP = 2, KING = 3, QUEEN = 4, ROOK = 5};

#endif /\* CHESSHEADER\_H \*/

#ifndef GAME\_H

#define GAME\_H

#include "tiles.h"

class Game

{

friend class tiles;

friend class Piece;

friend class Player;

private:

bool gamestate;

bool whiteturn;

tiles board[8][8];

char whitePieces[6] = {'P','N','B','K','Q','R'};

char blackPieces[6] = {'1','2','3','4','5','6'};

char alphaCoordinates[8] = {'a','b','c','d','e','f','g','h'};

public:

Game();

bool getGameState()

{ return this->gamestate; }

void setGameState(bool dummy);

bool getWhiteTurn()

{ return this->whiteturn; }

void flipWhiteTurn();

void displayBoard();

tiles getBoard(int i, int j)

{ return board[i][j]; }

char getWhitePieces(int val)

{ return whitePieces[val]; }

char getBlackPieces(int val)

{ return blackPieces[val]; }

char getAlphaCoordinates(int val)

{ return alphaCoordinates[val]; }

void initializeBoardPieces(vector<Piece\*>& piece);

void updateTiles(tiles board[8][8], vector<Piece\*>& piece, string playermove[], bool whiteturn, bool simulation);

void updateTiles(vector<Piece\*>& piece, Player player, bool whiteturn, bool simulation);

void updateTiles(vector <Piece\*>& piece);

bool isCheck(vector<Piece\*>& piece, int endmoverow, int endmovecol, bool white, bool hypo);

bool isWhite(int row, int col);

void verifyPlayerMoveStart(string playermove);

bool checkPieceColor(string playermove);

void verifyPlayerMoveEnd(string playermove);

bool verifyPlayerMove(Game game, string playermove[], vector <Piece\*>& piece, bool whiteturn, bool hypo, bool simulation, bool realtimewhiteturn);

string convertAtlasCoordinatesToNumeric(string playermove);

bool checkCapture(bool whiteturn, string playermove[], bool simulation, vector<Piece\*>& piece, bool realtimewhiteturn);

bool getBoardOccupied(int i, int j)

{ return board[i][j].occupied; }

bool checkMovement(int pos, int startmoverow, int startmovecol, int endmoverow, int endmovecol, vector<Piece\*>& piece, bool intentcapture)

{ return piece[pos]->checkMovement(board, startmoverow, startmovecol, endmoverow, endmovecol, piece, false); }

bool kingNotincheck(vector<Piece\*>& piece, string playermove[], bool whiteturn, bool hypo, bool simulation);

void findSupports(vector<Piece\*>& piece);

bool KingMoveSafe(vector<Piece\*>& piece, int endmoverow, int endmovecol, bool whiteturn);

bool isCheck(tiles copyboard[8][8], vector<Piece\*>& piece, int endmoverow, int endmovecol, bool white, bool hypo);

void updateGameState();

void generatePossibleMoves(vector <Piece\*>& piece, bool whiteturn);

string movePossible(int a, int b, vector <Piece\*>& piece, int c, bool whiteturn);

string convertNumericToAtlasCoordinates(int row, int col);

bool isCheckmate(vector<Piece\*>& piece, bool whiteturn);

bool isStalemate(vector <Piece\*>& piece, bool whiteturn);

bool isInsufficient(vector<Piece\*>& piece);

bool cannotMove(vector<Piece\*>& piece, int K);

void pawnUpdate(vector<Piece\*>& piece, bool simulation);

void pawnPromotion(vector<Piece\*>& piece, bool isWhite, int i);

};

#endif /\* GAME\_H \*/

#ifndef PIECE\_H

#define PIECE\_H

#include <string>

#include <iostream>

#include "tiles.h"

#include <bits/stdc++.h>

using namespace std;

class Piece

{

friend class tiles;

protected:

static int totalmoves;

int moves;

bool white;

bool captured;

string possible\_moves[64];

int position[2];

int piecevalue;

bool issupported;

bool incheck;

public:

int getPieceValue()

{ return this->piecevalue; }

void setPieceValue(int dummy);

int getMoves()

{ return this->moves; }

void setMoves(int dummy);

bool getWhite()

{ return this->white; }

void setWhite(bool dummy);

bool getCaptured()

{ return this->captured; }

void setCaptured(bool dummy);

string getPossible\_Moves(int val)

{ return possible\_moves[val]; }

void setPossible\_Moves(string dummy, int val);

int getPosition(int val)

{ return this->position[val]; }

void setPosition(int val, int dummy);

bool getIsSupported()

{ return this->issupported; }

void setIsSupported(bool dummy);

bool getInCheck()

{ return this->incheck; }

void setInCheck(bool dummy);

void addMoves(int dummy);

virtual bool checkMovement(tiles board[8][8], int startmoverow, int startmovecol, int endmoverow, int endmovecol, vector<Piece\*>& piece, bool intentcapture)

{ cout << "error at piece.h checkmovement"; }

virtual bool checkBlock(int startmoverow, int startmovecol, int endmoverow, int endmovecol, tiles board[8][8], vector<Piece\*>& piece)

{ cout << "error at piece.h checkblock"; }

};

#endif /\* PIECE\_H \*/

#ifndef PLAYER\_H

#define PLAYER\_H

#include "chessheader.h"

class Player

{

friend class Game;

private:

string playermove[4];

public:

Player();

string getPlayerMove(int val)

{ return this->playermove[val]; }

void setPlayerMove(int val, string dummy);

bool checkAlpha(string playermove);

bool checkNum(string playermove);

void getPlayerMoveStart();

void verifyPlayerMoveStart();

string convertAtlasCoordinatesToNumeric(string playermove);

void getPlayerMoveEnd();

void verifyPlayerMoveEnd();

};

#endif /\* PLAYER\_H \*/

#include "chessheader.h"

class Computer

{

private:

string computermove[4];

public:

virtual void determineMove() = 0;

virtual void setMove() = 0;

virtual void executeMove() = 0;

};

#ifndef PAWN\_H

#define PAWN\_H

#include "Piece.h"

#include "tiles.h"

class Pawn : public Piece

{

private:

bool ispawn;

public:

Pawn();

Pawn(bool white, int pos, int pieceval);

Pawn(Pawn\* obj);

bool getIsPawn()

{ return this->ispawn; }

void setIsPawn (bool dummy)

{ this->ispawn = dummy; }

// bool getDoubleAdvance()

// { return this->doubleadvance; }

// void setDoubleAdvance (bool dummy)

// { this->doubleadvance = dummy; }

//

// bool getPromotionValue()

// { return this->promotionvalue; }

// void setPromotionValue (bool dummy)

// { this->promotionvalue = dummy; }

virtual bool checkMovement(tiles board[8][8], int startmoverow, int startmovecol, int endmoverow, int endmovecol, vector<Piece\*>& piece, bool intentcapture);

};

#endif /\* PAWN\_H \*/

#ifndef ROOK\_H

#define ROOK\_H

#include "Piece.h"

#include "tiles.h"

#include <bits/stdc++.h>

class Rook : public Piece

{

private:

public:

Rook(bool white, int a, int b, int pieceval);

Rook(Rook\* obj);

virtual bool checkMovement(tiles board[8][8], int startmoverow, int startmovecol, int endmoverow, int endmovecol, vector<Piece\*>& piece, bool intentcapture);

bool checkBlock(int startmoverow, int startmovecol, int endmoverow, int endmovecol, tiles board[8][8], vector<Piece\*>& piece);

};

#endif /\* ROOK\_H \*/

#ifndef QUEEN\_H

#define QUEEN\_H

#include "Piece.h"

#include "tiles.h"

#include <bits/stdc++.h>

class Queen : public Piece

{

private:

bool issupported;

public:

Queen(bool white, int a, int b, int pieceval);

Queen(Queen\* obj);

virtual bool checkMovement(tiles board[8][8], int startmoverow, int startmovecol, int endmoverow, int endmovecol, vector<Piece\*>& piece, bool intentcapture);

bool checkBlock(int startmoverow, int startmovecol, int endmoverow, int endmovecol, tiles board[8][8], vector<Piece\*>& piece);

};

#endif /\* QUEEN\_H \*/

#ifndef KNIGHT\_H

#define KNIGHT\_H

#include <bits/stdc++.h>

#include "Piece.h"

#include "tiles.h"

class Knight : public Piece

{

private:

public:

Knight(bool white, int a, int b, int pieceval);

Knight(Knight\* obj);

virtual bool checkMovement(tiles board[8][8], int startmoverow, int startmovecol, int endmoverow, int endmovecol, vector<Piece\*>& piece, bool intentcapture);

};

#endif /\* KNIGHT\_H \*/

#ifndef KING\_H

#define KING\_H

#include "Piece.h"

#include "tiles.h"

#include <bits/stdc++.h>

class King : public Piece

{

private:

public:

King(bool white, int a, int b, int pieceval);

King(King\* obj);

virtual bool checkMovement(tiles board[8][8], int startmoverow, int startmovecol, int endmoverow, int endmovecol, vector<Piece\*>& piece, bool intentcapture);

};

#endif /\* KING\_H \*/

#ifndef BISHOP\_H

#define BISHOP\_H

#include "Piece.h"

#include "tiles.h"

#include <bits/stdc++.h>

using namespace std;

class Bishop : public Piece

{

private:

public:

Bishop(bool white, int a, int b, int pieceval);

Bishop(Bishop\* obj);

virtual bool checkMovement(tiles board[8][8], int startmoverow, int startmovecol, int endmoverow, int endmovecol, vector<Piece\*>& piece, bool intentcapture);

bool checkBlock(int startmoverow, int startmovecol, int endmoverow, int endmovecol, tiles board[8][8], vector<Piece\*>& piece);

};

#endif /\* BISHOP\_H \*/

**Class Implementation and Function Files**

#include "chessheader.h"

main()

{

Game game;

vector<Piece\*> piece;

Player player;

bool tryagain = true;

int scount = 0;

int icount = 0;

game.initializeBoardPieces(piece);

game.displayBoard();

game.updateTiles(piece);

while(!game.getGameState())

{

do

{

player.getPlayerMoveStart();

tryagain = true;

while (tryagain)

{

try

{

game.verifyPlayerMoveStart(player.getPlayerMove(1));

tryagain = false;

}

catch(string exception)

{

cout << exception << endl;

player.getPlayerMoveStart();

}

}

player.getPlayerMoveEnd();

}

while (!verifyPlayerMove(game, player, piece, game.getWhiteTurn(), false, false, game.getWhiteTurn()));

game.updateTiles(piece, player, game.getWhiteTurn(), false);

game.displayBoard();

game.flipWhiteTurn();

updateGameState(game.getWhiteTurn(), game, piece);

}

for (int i = 0; i < piece.size(); i++)

{

delete (piece[i]);

}

piece.clear();

}

#include "chessheader.h"

#include "Game.h"

Game::Game()

{

this->whiteturn = true;

this->gamestate = false;

}

void Game::setGameState(bool dummy)

{

gamestate = dummy;

}

void Game::flipWhiteTurn()

{

this->whiteturn = !this->whiteturn;

}

void Game::displayBoard()

{

//displaying the board using a nested for loop

cout << " ---------------------------------" << endl;

for(int i = 7; i > -1; i--)

{

cout << i+1 << " | ";

for(int j = 0; j < 8; j++)

{

cout << board[i][j].getSymbol() /\*<< i << j\*/ << " | ";

}

cout << endl << " ---------------------------------" << endl;

}

cout << endl << " ";

//displaying the coordinates

for (int i = 0; i < 8; i++)

{

cout << this->getAlphaCoordinates(i) << " ";

}

cout << endl << endl << endl << endl;

}

void Game::initializeBoardPieces(vector<Piece\*>& piece)

{

for (int i = 2; i < 6; i++)

{

for (int j = 0; j < 8; j++)

{

//space signifies empty space

board[i][j].setSymbol(' ');

}

}

for (int i = 0; i < 8; i++)

{

board[1][i].setSymbol(this->getWhitePieces(PAWN));

board[6][i].setSymbol(this->getBlackPieces(PAWN));

}

board[7][0].setSymbol(this->getBlackPieces(ROOK));

board[7][7].setSymbol(this->getBlackPieces(ROOK));

board[7][1].setSymbol(this->getBlackPieces(KNIGHT));

board[7][6].setSymbol(this->getBlackPieces(KNIGHT));

board[7][2].setSymbol(this->getBlackPieces(BISHOP));

board[7][5].setSymbol(this->getBlackPieces(BISHOP));

board[7][4].setSymbol(this->getBlackPieces(KING));

board[7][3].setSymbol(this->getBlackPieces(QUEEN));

board[0][7].setSymbol(this->getWhitePieces(ROOK));

board[0][0].setSymbol(this->getWhitePieces(ROOK));

board[0][6].setSymbol(this->getWhitePieces(KNIGHT));

board[0][1].setSymbol(this->getWhitePieces(KNIGHT));

board[0][5].setSymbol(this->getWhitePieces(BISHOP));

board[0][2].setSymbol(this->getWhitePieces(BISHOP));

board[0][4].setSymbol(this->getWhitePieces(KING));

board[0][3].setSymbol(this->getWhitePieces(QUEEN));

piece.push\_back(new Rook(false, 7, 0, ROOK));

piece.push\_back(new Knight(false, 7, 1, KNIGHT));

piece.push\_back(new Bishop(false, 7, 2, BISHOP));

piece.push\_back(new Queen(false, 7, 3, QUEEN));

piece.push\_back(new King(false, 7, 4, KING));

piece.push\_back(new Bishop(false, 7, 5, BISHOP));

piece.push\_back(new Knight(false, 7, 6, KNIGHT));

piece.push\_back(new Rook(false, 7, 7, ROOK));

piece.push\_back(new Rook(true, 0, 0, ROOK));

piece.push\_back(new Knight(true, 0, 1, KNIGHT));

piece.push\_back(new Bishop(true, 0, 2, BISHOP));

piece.push\_back(new Queen(true, 0, 3, QUEEN));

piece.push\_back(new King(true, 0, 4, KING));

piece.push\_back(new Bishop(true, 0, 5, BISHOP));

piece.push\_back(new Knight(true, 0, 6, KNIGHT));

piece.push\_back(new Rook(true, 0, 7, ROOK));

//white pawns

for (int i = 0; i < 8; i++)

{

piece.push\_back(new Pawn(true, i, PAWN));

}

//black pawns

for (int i = 0; i < 8; i++)

{

piece.push\_back(new Pawn(false, i, PAWN));

}

}

void Game::updateTiles(tiles copyboard[8][8], vector<Piece\*>& copypiece, string playermove[], bool whiteturn, bool simulation)

{

if (!(playermove[0] == "-1" || playermove[0] == "Resign" || playermove[0] == "OfferDraw"))

{

//integer values conversion

int startmoverow = playermove[1][1] - '0';

int startmovecol = playermove[1][0] - '0';

int endmoverow = playermove[3][1] - '0';

int endmovecol = playermove[3][0] - '0';

//getting the position of the pieces. if pos1 is -1 then there is no attempt to capture

int pos1 = pieceSearchArrayPosition(endmoverow, endmovecol, copypiece);

int pos2 = pieceSearchArrayPosition(startmoverow, startmovecol, copypiece);

//generic captures

if (pos1 != -1)

{

copypiece[pos1]->setPosition(0, -1);

copypiece[pos1]->setPosition(1, -1);

copypiece[pos1]->setCaptured(true);

}

//enpassants

if (pos1 == -1 && copypiece[pos2]->getPieceValue() == 0 && endmoverow - startmoverow == 1 && endmovecol - startmovecol == 1 && copyboard[endmoverow-1][endmovecol].getOccupied())

{

int pos3 = pieceSearchArrayPosition(startmoverow,startmovecol+1, copypiece);

copypiece[pos3]->setPosition(0, -1);

copypiece[pos3]->setPosition(1, -1);

copypiece[pos3]->setCaptured(true);

copyboard[startmoverow][startmovecol+1].symbol = ' ';

}

else if (pos1 == -1 && copypiece[pos2]->getPieceValue() == 0 && endmoverow - startmoverow == 1 && endmovecol - startmovecol == -1 && copyboard[endmoverow-1][endmovecol].getOccupied())

{

int pos3 = pieceSearchArrayPosition(startmoverow,startmovecol-1, copypiece);

copypiece[pos3]->setPosition(0, -1);

copypiece[pos3]->setPosition(1, -1);

copypiece[pos3]->setCaptured(true);

copyboard[startmoverow][startmovecol-1].symbol = ' ';

}

else if (pos1 == -1 && copypiece[pos2]->getPieceValue() == 0 && endmoverow - startmoverow == -1 && endmovecol - startmovecol == 1 && copyboard[endmoverow+1][endmovecol].getOccupied())

{

int pos3 = pieceSearchArrayPosition(startmoverow,startmovecol+1, copypiece);

copypiece[pos3]->setPosition(0, -1);

copypiece[pos3]->setPosition(1, -1);

copypiece[pos3]->setCaptured(true);

copyboard[startmoverow][startmovecol+1].symbol = ' ';

}

else if (pos1 == -1 && copypiece[pos2]->getPieceValue() == 0 && endmoverow - startmoverow == -1 && endmovecol - startmovecol == -1 && copyboard[endmoverow+1][endmovecol].getOccupied())

{

int pos3 = pieceSearchArrayPosition(startmoverow,startmovecol-1, copypiece);

copypiece[pos3]->setPosition(0, -1);

copypiece[pos3]->setPosition(1, -1);

copypiece[pos3]->setCaptured(true);

copyboard[startmoverow][startmovecol-1].symbol = ' ';

}

//castling

if (pos2 == 12 && endmovecol == 6 && copypiece[pos2]->getMoves() == 0)

{

copypiece[15]->setPosition(1, 5);

copypiece[15]->addMoves(1);

copyboard[0][5].symbol = copyboard[0][7].symbol;

copyboard[0][7].symbol = ' ';

}

else if (pos2 == 12 && endmovecol == 2 && copypiece[pos2]->getMoves() == 0)

{

copypiece[8]->setPosition(1, 3);

copypiece[8]->addMoves(1);

copyboard[0][3].symbol = copyboard[0][0].symbol;

copyboard[0][0].symbol = ' ';

}

else if (pos2 == 4 && endmovecol == 2 && copypiece[pos2]->getMoves() == 0)

{

copypiece[0]->setPosition(1, 3);

copypiece[0]->addMoves(1);

copyboard[7][3].symbol = copyboard[7][0].symbol;

copyboard[7][0].symbol = ' ';

}

else if (pos2 == 4 && endmovecol == 6 && copypiece[pos2]->getMoves() == 0)

{

copypiece[7]->setPosition(1, 5);

copypiece[7]->addMoves(1);

copyboard[7][5].symbol = copyboard[7][7].symbol;

copyboard[7][7].symbol = ' ';

}

//updating data

copypiece[pos2]->setPosition(0, endmoverow);

copypiece[pos2]->setPosition(1, endmovecol);

copypiece[pos2]->addMoves(1);

//updating board

copyboard[endmoverow][endmovecol].symbol = copyboard[startmoverow][startmovecol].symbol;

copyboard[startmoverow][startmovecol].symbol = ' ';

//updating pawns for double advance and promotion

// pawnUpdate(piece, blackPieces, whitePieces, board, simulation);

}

for (int i = 0; i < 8; i++)

{

for (int j = 0; j < 8; j++)

{

if (copyboard[i][j].getSymbol() != ' ')

{

copyboard[i][j].setOccupied(true);

copyboard[i][j].setOccupiedWhite(this->isWhite(i, j));

copyboard[i][j].setOccupiedBlack(!this->isWhite(i, j));

}

else if (copyboard[i][j].getSymbol() == ' ')

{

copyboard[i][j].setOccupied(false);

copyboard[i][j].setOccupiedWhite(false);

copyboard[i][j].setOccupiedBlack(false);

}

}

}

for (int i = 0; i < 8; i++)

{

for (int j = 0; j < 8; j++)

{

copyboard[i][j].setUnderAttackWhite(this->isCheck(copyboard, copypiece, i, j, true , false));

// if (board[i][j].getUnderAttackWhite())

// {

// cout << i << " " << j << " is under attack by white\n";

// }

copyboard[i][j].setUnderAttackBlack(this->isCheck(copyboard, copypiece, i, j, false, false));

// if (board[i][j].getUnderAttackBlack())

// {

// cout << i << " " << j << " is under attack by black\n";

// }

}

}

this->findSupports(copypiece);

}

bool Game::isCheck(vector<Piece\*>& piece, int endmoverow, int endmovecol, bool white, bool hypo)

{

bool incheck = false;

//checking if any white piece can come to the destination tile for check

if (white && !hypo)

{

for (int i = 8; i < 24 && !incheck; i++)

{

if (piece[i]->getCaptured() == false)

{

// cout << "piece " << i;

incheck = piece[i]->checkMovement(board, piece[i]->getPosition(0), piece[i]->getPosition(1), endmoverow, endmovecol, piece, true);

// if (incheck)

// {

// cout << endmoverow << " " << endmovecol << "is under attack by " << i << endl;

// }

}

}

}

//same thing but for all black pieces

else if (!white && !hypo)

{

for (int i = 0; i < 8 && !incheck; i++)

{

// cout << "piece " << i;

if (piece[i]->getCaptured() == false)

{

incheck = piece[i]->checkMovement(board, piece[i]->getPosition(0), piece[i]->getPosition(1), endmoverow, endmovecol, piece, true);

}

}

for (int i = 24; i < 32 && !incheck; i++)

{

// cout << "piece " << i;

if (piece[i]->getCaptured() == false && !incheck)

{

incheck = piece[i]->checkMovement(board, piece[i]->getPosition(0), piece[i]->getPosition(1), endmoverow, endmovecol, piece, true);

}

}

}

return incheck;

}

bool Game::isWhite(int row, int col)

{

bool white = false;

for (int i = 0; i < 6; i++)

{

if (board[row][col].getSymbol() == whitePieces[i])

{

white = true;

}

}

//returning flag

return white;

}

void Game::verifyPlayerMoveStart(string playermove)

{

if(!this->checkPieceColor(playermove))

{

throw string("Invalid selection");

}

}

bool Game::checkPieceColor(string playermove)

{

bool valid = false;

int movecol = playermove[0] - '0';

int moverow = playermove[1] - '0';

if (this->getWhiteTurn())

{

for (int i = 0; i < 6; i++)

{

if (board[moverow][movecol].getSymbol() == whitePieces[i])

{

valid = true;

}

}

}

else if (!this->getWhiteTurn())

{

for (int i = 0; i < 6; i++)

{

if (board[moverow][movecol].getSymbol() == blackPieces[i])

{

valid = true;

}

}

}

return valid;

}

string Game::convertAtlasCoordinatesToNumeric(string playermove)

{

playermove[0] -= 49;

//checks for single integer since conversion to int truncates the zero

if (playermove[0] == '0')

{

if (playermove[1] == '1')

{

return "00";

}

else if (playermove[1] == '2')

{

return "01";

}

else if (playermove[1] == '3')

{

return "02";

}

else if (playermove[1] == '4')

{

return "03";

}

else if (playermove[1] == '5')

{

return "04";

}

else if (playermove[1] == '6')

{

return "05";

}

else if (playermove[1] == '7')

{

return "06";

}

else if (playermove[1] == '8')

{

return "07";

}

}

return to\_string(stoi(playermove)-1);

}

bool Game::checkCapture(bool whiteturn, string playermove[], bool simulation, vector<Piece\*>& piece, bool realtimewhiteturn)

{

//declare flag

bool capturable = true;

// //initialize the integer representation of moves as well as the piece array position

// int startmoverow = playermove[1][1] - '0';

// int startmovecol = playermove[1][0] - '0';

//

// int i = pieceSearchArrayPosition(startmoverow, startmovecol, piece);

int endmoverow = playermove[3][1] - '0';

int endmovecol = playermove[3][0] - '0';

//check if the destination tile is occupied by same color piece

if (whiteturn && board[endmoverow][endmovecol].occ\_white)

{

capturable = false;

}

else if (!whiteturn && board[endmoverow][endmovecol].occ\_black)

{

capturable = false;

}

//return flag

return capturable;

}

bool Game::kingNotincheck(vector<Piece\*>& piece, string playermove[], bool whiteturn, bool hypo, bool simulation)

{

//declaring copy array of structs for simulation

tiles copyboard[8][8];

vector<Piece\*> copypiece;

//copying over the data from original

for (int i = 0; i < 8; i++)

{

for (int j = 0; j < 8; j++)

{

copyboard[i][j] = board[i][j];

}

}

copypiece.push\_back(new Rook(static\_cast<Rook\*>(piece[0])));

copypiece.push\_back(new Knight(static\_cast<Knight\*>(piece[1])));

copypiece.push\_back(new Bishop(static\_cast<Bishop\*>(piece[2])));

copypiece.push\_back(new Queen(static\_cast<Queen\*>(piece[3])));

copypiece.push\_back(new King(static\_cast<King\*>(piece[4])));

copypiece.push\_back(new Bishop(static\_cast<Bishop\*>(piece[5])));

copypiece.push\_back(new Knight(static\_cast<Knight\*>(piece[6])));

copypiece.push\_back(new Rook(static\_cast<Rook\*>(piece[7])));

copypiece.push\_back(new Rook(static\_cast<Rook\*>(piece[8])));

copypiece.push\_back(new Knight(static\_cast<Knight\*>(piece[9])));

copypiece.push\_back(new Bishop(static\_cast<Bishop\*>(piece[10])));

copypiece.push\_back(new Queen(static\_cast<Queen\*>(piece[11])));

copypiece.push\_back(new King(static\_cast<King\*>(piece[12])));

copypiece.push\_back(new Bishop(static\_cast<Bishop\*>(piece[13])));

copypiece.push\_back(new Knight(static\_cast<Knight\*>(piece[14])));

copypiece.push\_back(new Rook(static\_cast<Rook\*>(piece[15])));

//direct copying piece data array

for (int i = 16; i < 24; i++)

{

copypiece.push\_back(new Pawn(static\_cast<Pawn\*>(piece[i])));

}

for (int i = 24; i < 32; i++)

{

copypiece.push\_back(new Pawn(static\_cast<Pawn\*>(piece[i])));

}

//simulation with copied data

updateTiles(copyboard, copypiece, playermove, whiteturn, true);

//declaring integer values of movements

int startmoverow = playermove[1][1] - '0';

int startmovecol = playermove[1][0] - '0';

int endmoverow = playermove[3][1] - '0';

int endmovecol = playermove[3][0] - '0';

//getting position values

int pos1 = pieceSearchArrayPosition(startmoverow, startmovecol, piece);

int pos2 = pieceSearchArrayPosition(endmoverow, endmovecol, piece);

//checking if the king is trying to attack a supported piece

if (pos1 == 4 || pos1 == 12)

{

if (board[endmoverow][endmovecol].getOccupied())

{

if (copypiece[pos2]->getIsSupported())

{

return false;

}

}

}

if (whiteturn)

{

return !copyboard[copypiece[12]->getPosition(0)][copypiece[12]->getPosition(1)].underattackBlack;

}

else if (!whiteturn)

{

return !copyboard[copypiece[4]->getPosition(0)][copypiece[4]->getPosition(1)].underattackWhite;

}

}

void Game::findSupports(vector<Piece\*>& piece)

{

bool support;

for (int i = 8; i < 24; i++)

{

support = false;

for (int j = 8; j < 24; j++)

{

if (!piece[j]->getCaptured() && !piece[i]->getCaptured())

{

if (piece[j]->checkMovement(board, piece[j]->getPosition(0), piece[j]->getPosition(1), piece[i]->getPosition(0), piece[i]->getPosition(1), piece, true))

{

support = true;

piece[i]->setIsSupported(true);

}

}

}

if (support == false)

{

piece[i]->setIsSupported(false);

}

}

//going through first part black pieces

for (int i = 0; i < 8; i++)

{

support = false;

for (int j = 0; j < 8; j++)

{

if (!piece[j]->getCaptured() && !piece[i]->getCaptured())

{

if (piece[j]->checkMovement(board, piece[j]->getPosition(0), piece[j]->getPosition(1), piece[i]->getPosition(0), piece[i]->getPosition(1), piece, true))

{

support = true;

piece[i]->setIsSupported(true);

}

}

}

for (int j = 24; j < 32; j++)

{

if (!piece[j]->getCaptured() && !piece[i]->getCaptured())

{

if (piece[j]->checkMovement(board, piece[j]->getPosition(0), piece[j]->getPosition(1), piece[i]->getPosition(0), piece[i]->getPosition(1), piece, true))

{

support = true;

piece[i]->setIsSupported(true);

}

}

}

if (support == false)

{

piece[i]->setIsSupported(false);

}

}

//second part of black pieces

for (int i = 24; i < 32; i++)

{

support = false;

for (int j = 0; j < 8; j++)

{

if (!piece[j]->getCaptured() && !piece[i]->getCaptured())

{

if (piece[j]->checkMovement(board, piece[j]->getPosition(0), piece[j]->getPosition(1), piece[i]->getPosition(0), piece[i]->getPosition(1), piece, true))

{

support = true;

piece[i]->setIsSupported(true);

}

}

}

for (int j = 24; j < 32; j++)

{

if (!piece[j]->getCaptured() && !piece[i]->getCaptured())

{

if (piece[j]->checkMovement(board, piece[j]->getPosition(0), piece[j]->getPosition(1), piece[i]->getPosition(0), piece[i]->getPosition(1), piece, true))

{

support = true;

piece[i]->setIsSupported(true);

}

}

}

if (support == false)

{

piece[i]->setIsSupported(false);

}

}

}

void Game::updateTiles(vector <Piece\*>& piece)

{

for (int i = 0; i < 8; i++)

{

for (int j = 0; j < 8; j++)

{

if (board[i][j].getSymbol() != ' ')

{

board[i][j].setOccupied(true);

board[i][j].setOccupiedWhite(isWhite(i, j));

board[i][j].setOccupiedBlack(!isWhite(i, j));

// if (board[i][j].getOccupiedBlack())

// {

// cout << i << j << "is occupied by black\n";

// }

}

else if (board[i][j].getSymbol() == ' ')

{

board[i][j].setOccupied(false);

board[i][j].setOccupiedWhite(false);

board[i][j].setOccupiedBlack(false);

}

}

}

for (int i = 0; i < 8; i++)

{

for (int j = 0; j < 8; j++)

{

board[i][j].setUnderAttackWhite(this->isCheck(piece, i, j, true , false));

board[i][j].setUnderAttackBlack(this->isCheck(piece, i, j, false, false));

}

}

this->findSupports(piece);

}

bool Game::KingMoveSafe(vector<Piece\*>& piece, int endmoverow, int endmovecol, bool whiteturn)

{

//declare flag

bool safemove = true;

//check if the destination tile is under check based on color

if (whiteturn && board[endmoverow][endmovecol].underattackBlack)

{

safemove = false;

}

else if (!whiteturn && board[endmoverow][endmovecol].underattackWhite)

{

safemove = false;

}

//return flag

return safemove;

}

void Game::updateTiles(vector<Piece\*>& piece, Player player, bool whiteturn, bool simulation)

{

string playermove[4];

playermove[0] = player.getPlayerMove(0);

playermove[1] = player.getPlayerMove(1);

playermove[2] = player.getPlayerMove(2);

playermove[3] = player.getPlayerMove(3);

if (!(playermove[0] == "-1" || playermove[0] == "Resign" || playermove[0] == "OfferDraw"))

{

//integer values conversion

int startmoverow = playermove[1][1] - '0';

int startmovecol = playermove[1][0] - '0';

int endmoverow = playermove[3][1] - '0';

int endmovecol = playermove[3][0] - '0';

//getting the position of the pieces. if pos1 is -1 then there is no attempt to capture

int pos1 = pieceSearchArrayPosition(endmoverow, endmovecol, piece);

int pos2 = pieceSearchArrayPosition(startmoverow, startmovecol, piece);

//generic captures

if (pos1 != -1)

{

piece[pos1]->setPosition(0, -1);

piece[pos1]->setPosition(1, -1);

piece[pos1]->setCaptured(true);

}

//enpassants

if (pos1 == -1 && piece[pos2]->getPieceValue() == 0 && endmoverow - startmoverow == 1 && endmovecol - startmovecol == 1 && board[endmoverow-1][endmovecol].getOccupied())

{

int pos3 = pieceSearchArrayPosition(startmoverow,startmovecol+1, piece);

piece[pos3]->setPosition(0, -1);

piece[pos3]->setPosition(1, -1);

piece[pos3]->setCaptured(true);

board[startmoverow][startmovecol+1].symbol = ' ';

}

else if (pos1 == -1 && piece[pos2]->getPieceValue() == 0 && endmoverow - startmoverow == 1 && endmovecol - startmovecol == -1 && board[endmoverow-1][endmovecol].getOccupied())

{

int pos3 = pieceSearchArrayPosition(startmoverow,startmovecol-1, piece);

piece[pos3]->setPosition(0, -1);

piece[pos3]->setPosition(1, -1);

piece[pos3]->setCaptured(true);

board[startmoverow][startmovecol-1].symbol = ' ';

}

else if (pos1 == -1 && piece[pos2]->getPieceValue() == 0 && endmoverow - startmoverow == -1 && endmovecol - startmovecol == 1 && board[endmoverow+1][endmovecol].getOccupied())

{

int pos3 = pieceSearchArrayPosition(startmoverow,startmovecol+1, piece);

piece[pos3]->setPosition(0, -1);

piece[pos3]->setPosition(1, -1);

piece[pos3]->setCaptured(true);

board[startmoverow][startmovecol+1].symbol = ' ';

}

else if (pos1 == -1 && piece[pos2]->getPieceValue() == 0 && endmoverow - startmoverow == -1 && endmovecol - startmovecol == -1 && board[endmoverow+1][endmovecol].getOccupied())

{

int pos3 = pieceSearchArrayPosition(startmoverow,startmovecol-1, piece);

piece[pos3]->setPosition(0, -1);

piece[pos3]->setPosition(1, -1);

piece[pos3]->setCaptured(true);

board[startmoverow][startmovecol-1].symbol = ' ';

}

//castling

if (pos2 == 12 && endmovecol == 6 && piece[pos2]->getMoves() == 0)

{

piece[15]->setPosition(1, 5);

piece[15]->addMoves(1);

board[0][5].symbol = board[0][7].symbol;

board[0][7].symbol = ' ';

}

else if (pos2 == 12 && endmovecol == 2 && piece[pos2]->getMoves() == 0)

{

piece[8]->setPosition(1, 3);

piece[8]->addMoves(1);

board[0][3].symbol = board[0][0].symbol;

board[0][0].symbol = ' ';

}

else if (pos2 == 4 && endmovecol == 2 && piece[pos2]->getMoves() == 0)

{

piece[0]->setPosition(1, 3);

piece[0]->addMoves(1);

board[7][3].symbol = board[7][0].symbol;

board[7][0].symbol = ' ';

}

else if (pos2 == 4 && endmovecol == 6 && piece[pos2]->getMoves() == 0)

{

piece[7]->setPosition(1, 5);

piece[7]->addMoves(1);

board[7][5].symbol = board[7][7].symbol;

board[7][7].symbol = ' ';

}

//updating data

piece[pos2]->setPosition(0, endmoverow);

piece[pos2]->setPosition(1, endmovecol);

piece[pos2]->addMoves(1);

//updating board

board[endmoverow][endmovecol].symbol = board[startmoverow][startmovecol].symbol;

board[startmoverow][startmovecol].symbol = ' ';

//updating pawns for double advance and promotion

pawnUpdate(piece, simulation);

}

for (int i = 0; i < 8; i++)

{

for (int j = 0; j < 8; j++)

{

if (board[i][j].getSymbol() != ' ')

{

board[i][j].setOccupied(true);

board[i][j].setOccupiedWhite(this->isWhite(i, j));

board[i][j].setOccupiedBlack(!this->isWhite(i, j));

}

else if (board[i][j].getSymbol() == ' ')

{

board[i][j].setOccupied(false);

board[i][j].setOccupiedWhite(false);

board[i][j].setOccupiedBlack(false);

}

}

}

for (int i = 0; i < 8; i++)

{

for (int j = 0; j < 8; j++)

{

board[i][j].setUnderAttackWhite(this->isCheck(piece, i, j, true , false));

// if (board[i][j].getUnderAttackWhite())

// {

// cout << i << " " << j << " is under attack by white\n";

// }

board[i][j].setUnderAttackBlack(this->isCheck(piece, i, j, false, false));

// if (board[i][j].getUnderAttackBlack())

// {

// cout << i << " " << j << " is under attack by black\n";

// }

}

}

this->findSupports(piece);

// for (int i = 0; i < piece.size(); i++)

// {

// piece[i]->getIsSupported()?cout<<i<<"is supported":cout<<i<<" is not supported";

// cout << endl;

// }

}

bool Game::isCheck(tiles copyboard[8][8], vector<Piece\*>& copypiece, int endmoverow, int endmovecol, bool white, bool hypo)

{

bool incheck = false;

//checking if any white piece can come to the destination tile for check

if (white && !hypo)

{

for (int i = 8; i < 24 && !incheck; i++)

{

if (copypiece[i]->getCaptured() == false)

{

// cout << "piece " << i;

incheck = copypiece[i]->checkMovement(copyboard, copypiece[i]->getPosition(0), copypiece[i]->getPosition(1), endmoverow, endmovecol, copypiece, true);

// if (incheck)

// {

// cout << endmoverow << " " << endmovecol << "is under attack by " << i << endl;

// }

}

}

}

//same thing but for all black pieces

else if (!white && !hypo)

{

for (int i = 0; i < 8 && !incheck; i++)

{

// cout << "piece " << i;

if (copypiece[i]->getCaptured() == false)

{

incheck = copypiece[i]->checkMovement(copyboard, copypiece[i]->getPosition(0), copypiece[i]->getPosition(1), endmoverow, endmovecol, copypiece, true);

}

}

for (int i = 24; i < 32 && !incheck; i++)

{

// cout << "piece " << i;

if (copypiece[i]->getCaptured() == false && !incheck)

{

incheck = copypiece[i]->checkMovement(copyboard, copypiece[i]->getPosition(0), copypiece[i]->getPosition(1), endmoverow, endmovecol, copypiece, true);

}

}

}

return incheck;

}

void Game::generatePossibleMoves(vector <Piece\*>& piece, bool whiteturn)

{

//set counter to zero

int m = 0;

//for loop to go thorough every move

//else if statement to catch captured pieces

for (int i = 0; i < 32; i++)

{

m = 0;

if (!piece[i]->getCaptured())

{

for (int j = 0; j < 8; j++)

{

for (int k = 0; k < 8; k++)

{

piece[i]->setPossible\_Moves(movePossible(j, k, piece, i, whiteturn), m++);

}

}

}

else if (piece[i]->getCaptured())

{

for (int j = 0; j < 64; j++)

{

piece[i]->setPossible\_Moves("-1", j);

}

}

}

}

string Game::movePossible(int a, int b, vector <Piece\*>& piece, int c, bool whiteturn)

{

//assigning integer values of the movements

string playermove[4];

playermove[0] = convertNumericToAtlasCoordinates(piece[c]->getPosition(0), piece[c]->getPosition(1));

playermove[1] = convertAtlasCoordinatesToNumeric(playermove[0]);

playermove[2] = convertNumericToAtlasCoordinates(a, b);

playermove[3] = convertAtlasCoordinatesToNumeric(playermove[2]);

//boolean if the move is possible

bool result = verifyPlayerMove(\*this, playermove, piece, piece[c]->getWhite(), true, true, whiteturn);

//returning the result to the array element

if (!result)

{

return "-1";

}

if (result)

{

return playermove[3];

}

}

string Game::convertNumericToAtlasCoordinates(int row, int col)

{

//declare string

string str;

//assigning string values

str.push\_back(getAlphaCoordinates(col));

str.push\_back(row + 1 + '0');

//returning string values

return str;

}

bool Game::verifyPlayerMove(Game game, string playermove[], vector <Piece\*>& piece, bool whiteturn, bool hypo, bool simulation, bool realtimewhiteturn)

{

//checkpoints

bool checkpoints[4] = {1,0,0,1};

//if not simulation then the player 3 move must be assigned since its not preset

if (!hypo)

{

playermove[3] = game.convertAtlasCoordinatesToNumeric(playermove[2]);

}

//integer conversions

int startmoverow = playermove[1][1] - '0';

int startmovecol = playermove[1][0] - '0';

int endmoverow = playermove[3][1] - '0';

int endmovecol = playermove[3][0] - '0';

int pos = pieceSearchArrayPosition(startmoverow, startmovecol, piece);

//check capture only if destination tile is occupied

if(game.getBoardOccupied(endmoverow, endmovecol))

{

checkpoints[0] = game.checkCapture(whiteturn, playermove, simulation, piece, realtimewhiteturn);

}

//checking movement

checkpoints[1] = game.checkMovement(pos, startmoverow, startmovecol, endmoverow, endmovecol, piece, false);

//making sure the king is not in check with a simulatio

checkpoints[2] = game.kingNotincheck(piece, playermove, whiteturn, hypo, true);

//making sure that the king doesnt move into a checked square if the king is selected

if (pieceSearchArrayPosition(startmoverow, startmovecol, piece) == 4 && !whiteturn)

{

checkpoints[3] = game.KingMoveSafe(piece, endmoverow, endmovecol, whiteturn);

}

else if (pieceSearchArrayPosition(startmoverow, startmovecol, piece) == 12 && whiteturn)

{

checkpoints[3] = game.KingMoveSafe(piece, endmoverow, endmovecol, whiteturn);

}

//validation

if (!(checkpoints[0] && checkpoints[1] && checkpoints[2] && checkpoints[3]))

{

//prevent spamming the console during simulations

if (!simulation)

{

//outputting whats wrong

if (!checkpoints[0])

{

cout << "you cannot capture your own piece" << endl;

}

if (!checkpoints[1])

{

cout << "your movement is invalid" << endl;

}

if (!checkpoints[2])

{

cout << "Your king cannot move there or is still in check!" << endl;

}

if (!checkpoints[3])

{

cout << "Your king is under attack!" << endl;

}

cout << endl << endl;

return false;

}

}

else if(checkpoints[0] && checkpoints[1] && checkpoints[2])

{

return true;

}

}

bool Game::isCheckmate(vector<Piece\*>& piece, bool whiteturn)

{

//flag and checkpoints

bool checkmate = false;

bool checkmateConditions[3] = {0,0,0};

//assignment

int i = -1;

//in check flag

if (board[piece[12]->getPosition(0)][piece[12]->getPosition(1)].underattackBlack)

{

checkmateConditions[0] = true;

i=12;

}

else if (board[piece[4]->getPosition(0)][piece[4]->getPosition(1)].underattackWhite)

{

checkmateConditions[0] = true;

i=4;

}

//return if first conditions fail as to prevent unneccessary simulation

if (i == -1)

{

return checkmate;

}

//simulation

//checking if there are no moves for black to get out of check

checkmateConditions[1] = cannotMove(piece, i);

if (checkmateConditions[1])

{

checkmate = true;

}

//return flag

return checkmate;

}

bool Game::isStalemate(vector <Piece\*>& piece, bool whiteturn)

{

//set flags

bool stalemate = false;

bool flag = false;

//checking through all possible moves to see if any are valid

if (!whiteturn)

{

for (int i = 0; i < 8; i++)

{

for (int m = 0; m < 64 && !flag; m++)

{

if (piece[i]->getPossible\_Moves(m) != "-1")

{

flag = true;

}

}

}

for (int i = 24; i < 32; i++)

{

for (int m = 0; m < 64 && !flag; m++)

{

if (piece[i]->getPossible\_Moves(m) != "-1")

{

flag = true;

}

}

}

if (!flag && !piece[4]->getInCheck())

{

stalemate = true;

}

}

//same thing but for white

else if (whiteturn)

{

for (int i = 8; i < 24; i++)

{

for (int m = 0; m < 64 && !flag; m++)

{

if (piece[i]->getPossible\_Moves(m) != "-1")

{

flag = true;

}

}

}

if (!flag && !piece[12]->getInCheck())

{

stalemate = true;

}

}

//return flag

return stalemate;

}

bool Game::isInsufficient(vector<Piece\*>& piece)

{

//set counters for minor pieces

int whiteknights = 0;

int whitebishops = 0;

int blackknights = 0;

int blackbishops = 0;

int whiteminor;

int blackminor;

//checking if any pawn is alive, if so, not insufficient

for (int i = 16; i < 32; i++)

{

if (!piece[i]->getCaptured())

{

if (piece[i]->getPieceValue() == 0)

{

return false;

}

else if (piece[i]->getPieceValue() != -1)

{

if (piece[i]->getPieceValue() == 1 || piece[i]->getPieceValue() == 2)

{

if (piece[i]->getWhite())

{

whiteminor++;

}

else if (!piece[i]->getWhite())

{

blackminor++;

}

}

else if (piece[i]->getPieceValue() == 4 || piece[i]->getPieceValue() == 0)

{

return false;

}

}

}

}

//if any rook exists, then false;

if (!piece[0]->getCaptured() || !piece[7]->getCaptured() || !piece[8]->getCaptured() || !piece[15]->getCaptured())

{

return false;

}

//if any queen exists, then false;

else if (!piece[3]->getCaptured() || !piece[11]->getCaptured())

{

return false;

}

//counting all alive minor pieces

for (int i = 1; i < 15; i++)

{

if (!piece[i]->getCaptured())

{

if (i == 1 || i == 6)

{

blackknights++;

}

else if ( i == 2 || i == 5)

{

blackbishops++;

}

else if (i == 9 || i == 14)

{

whiteknights++;

}

else if (i == 10 || i == 13)

{

whitebishops++;

}

}

}

//totaling pieces

whiteminor = whiteknights + whitebishops;

blackminor = blackknights + blackbishops;

//if the function gets to here, then no pawns, rooks, or queens exist

//no minor pieces

if (whiteminor + blackminor == 0)

{

return true;

}

//if white has one knight and no other pieces exist

if ((whiteknights == 1 && blackknights == 0) && (blackbishops == 0 && whitebishops == 0))

{

return true;

}

//if black has one knight and no other pieces exist

if ((whiteknights == 0 && blackknights == 1) && (blackbishops == 0 && whitebishops == 0))

{

return true;

}

//if white has 2 knights and no other pieces exist

if ((whiteknights == 2 && blackknights == 0) && (blackbishops == 0 && whitebishops == 0))

{

return true;

}

//if black has 2 knights and no other pieces exist

if ((whiteknights == 0 && blackknights == 2) && (blackbishops == 0 && whitebishops == 0))

{

return true;

}

//one minor piece each side, checkmate cannot be forced, but it possible

if (blackminor == 1 && whiteminor == 1)

{

return true;

}

}

bool Game::cannotMove(vector<Piece\*>& piece, int K)

{

//declaring flag

bool cannotMove = true;

//checking black king possible moves

if (K == 4)

{

for (int m = 0; m < 64; m++)

{

if (piece[K]->getPossible\_Moves(m) != "-1")

{

cannotMove = false;

}

}

}

//checking white king possible moves

else if (K == 12)

{

for (int m = 0; m < 64; m++)

{

if (piece[K]->getPossible\_Moves(m) != "-1")

{

cannotMove = false;

}

}

}

//checking all black pieces

if (K == 4)

{

for (int i = 0; i < 8; i++)

{

for (int m = 0; m < 64; m++)

{

if (piece[i]->getPossible\_Moves(m) != "-1")

{

cannotMove = false;

}

}

}

for (int i = 24; i < 32; i++)

{

for (int m = 0; m < 64; m++)

{

if (piece[i]->getPossible\_Moves(m) != "-1")

{

cannotMove = false;

}

}

}

}

//checking all white pieces.

else if (K == 12)

{

for (int i = 8; i < 24; i++)

{

for (int m = 0; m < 64; m++)

{

if (piece[i]->getPossible\_Moves(m) != "-1")

{

cannotMove = false;

}

}

}

}

return cannotMove;

}

void Game::pawnUpdate(vector<Piece\*>& piece, bool simulation)

{

for (int i = 16; i < 32; i++)

{

if(piece[i]->getWhite() && piece[i]->getPieceValue() == PAWN)

{

if (piece[i]->getPosition(0) == 7)

{

pawnPromotion(piece, true, i);

}

}

else if (!piece[i]->getWhite() && piece[i]->getPieceValue() == PAWN)

{

if (piece[i]->getPosition(0) == 0)

{

pawnPromotion(piece, false, i);

}

}

}

}

void Game::pawnPromotion(vector<Piece\*>& piece, bool isWhite, int i)

{

int pvalue;

//choosing promotion piece

cout << "1. Knight " << endl << "2. Bishop " << endl;

cout << "4. Queen " << endl << "5. Rook" << endl;

cout << "Enter the promotion value: ";

cin >> pvalue;

while (pvalue != 1 && pvalue != 2 && pvalue != 4 && pvalue != 5)

{

cout << "Invalid value: try again ";

cin >> pvalue;

}

//assignment of new\_i for movement and new symbol

if(isWhite)

{

if (pvalue == 1)

{

piece[i]->setPieceValue(1);

board[piece[i]->getPosition(0)][piece[i]->getPosition(1)].symbol = whitePieces[KNIGHT];

}

else if (pvalue == 2)

{

piece[i]->setPieceValue(2);

board[piece[i]->getPosition(0)][piece[i]->getPosition(1)].symbol = whitePieces[BISHOP];

}

else if (pvalue == 4)

{

piece[i]->setPieceValue(4);

board[piece[i]->getPosition(0)][piece[i]->getPosition(1)].symbol = whitePieces[QUEEN];

}

else if (pvalue == 5)

{

piece[i]->setPieceValue(5);

board[piece[i]->getPosition(0)][piece[i]->getPosition(1)].symbol = whitePieces[ROOK];

}

}

else if(!isWhite)

{

if (pvalue == 1)

{

piece[i]->setPieceValue(1);

board[piece[i]->getPosition(0)][piece[i]->getPosition(1)].symbol = blackPieces[KNIGHT];

}

else if (pvalue == 2)

{

piece[i]->setPieceValue(2);

board[piece[i]->getPosition(0)][piece[i]->getPosition(1)].symbol = blackPieces[BISHOP];

}

else if (pvalue == 4)

{

piece[i]->setPieceValue(4);

board[piece[i]->getPosition(0)][piece[i]->getPosition(1)].symbol = blackPieces[QUEEN];

}

else if (pvalue == 5)

{

piece[i]->setPieceValue(5);

board[piece[i]->getPosition(0)][piece[i]->getPosition(1)].symbol = blackPieces[ROOK];

}

}

}

#include "chessheader.h"

bool verifyPlayerMove(Game game, Player player, vector <Piece\*>& piece, bool whiteturn, bool hypo, bool simulation, bool realtimewhiteturn)

{

//checkpoints

bool checkpoints[4] = {1,0,0,1};

string playermove[4];

playermove[0] = player.getPlayerMove(0);

playermove[1] = player.getPlayerMove(1);

playermove[2] = player.getPlayerMove(2);

playermove[3] = player.getPlayerMove(3);

//if not simulation then the player 3 move must be assigned since its not preset

if (!hypo)

{

playermove[3] = game.convertAtlasCoordinatesToNumeric(playermove[2]);

}

//integer conversions

int startmoverow = playermove[1][1] - '0';

int startmovecol = playermove[1][0] - '0';

int endmoverow = playermove[3][1] - '0';

int endmovecol = playermove[3][0] - '0';

int pos = pieceSearchArrayPosition(startmoverow, startmovecol, piece);

//check capture only if destination tile is occupied

if(game.getBoardOccupied(endmoverow, endmovecol))

{

checkpoints[0] = game.checkCapture(whiteturn, playermove, simulation, piece, realtimewhiteturn);

}

//checking movement

checkpoints[1] = game.checkMovement(pos, startmoverow, startmovecol, endmoverow, endmovecol, piece, false);

//making sure the king is not in check with a simulatio

checkpoints[2] = game.kingNotincheck(piece, playermove, whiteturn, hypo, true);

//making sure that the king doesnt move into a checked square if the king is selected

if (pieceSearchArrayPosition(startmoverow, startmovecol, piece) == 4 && !whiteturn)

{

checkpoints[3] = game.KingMoveSafe(piece, endmoverow, endmovecol, whiteturn);

}

else if (pieceSearchArrayPosition(startmoverow, startmovecol, piece) == 12 && whiteturn)

{

checkpoints[3] = game.KingMoveSafe(piece, endmoverow, endmovecol, whiteturn);

}

//validation

if (!(checkpoints[0] && checkpoints[1] && checkpoints[2] && checkpoints[3]))

{

//prevent spamming the console during simulations

if (!simulation)

{

//outputting whats wrong

if (!checkpoints[0])

{

cout << "you cannot capture your own piece" << endl;

}

if (!checkpoints[1])

{

cout << "your movement is invalid" << endl;

}

if (!checkpoints[2])

{

cout << "Your king cannot move there or is still in check!" << endl;

}

if (!checkpoints[3])

{

cout << "Your king is under attack!" << endl;

}

cout << endl << endl;

return false;

}

}

else if(checkpoints[0] && checkpoints[1] && checkpoints[2])

{

return true;

}

}

#include "chessheader.h"

void tiles::setSymbol(char sym)

{

this->symbol = sym;

}

tiles::tiles(char sym)

{

this->symbol = sym;

}

tiles::tiles()

{

this->occupied = false;

this->occ\_white = false;

this->occ\_black = false;

this->underattackWhite = false;

this->underattackBlack = false;

this->symbol = false;

}

void tiles::setOccupiedWhite(bool dummy)

{

this->occ\_white = dummy;

}

void tiles::setOccupiedBlack(bool dummy)

{

this->occ\_black = dummy;

}

void tiles::setOccupied(bool dummy)

{

this->occupied = dummy;

}

void tiles::setUnderAttackBlack(bool dummy)

{

this->underattackBlack = dummy;

}

void tiles::setUnderAttackWhite(bool dummy)

{

this->underattackWhite = dummy;

}

tiles::tiles(tiles &obj)

{

occupied = obj.occupied;

occ\_white = obj.occ\_white;

occ\_black = obj.occ\_black;

underattackWhite = obj.underattackWhite;

underattackBlack = obj.underattackBlack;

symbol = obj.symbol;

}

#include "chessheader.h"

void Piece::setMoves(int dummy)

{

this->moves = dummy;

}

void Piece::setWhite(bool dummy)

{

this->white = dummy;

}

void Piece::setCaptured(bool dummy)

{

this->captured = dummy;

}

void Piece::setPossible\_Moves(string dummy, int val)

{

this->possible\_moves[val] = dummy;

}

void Piece::setPosition(int val, int dummy)

{

this->position[val] = dummy;

}

void Piece::setIsSupported(bool dummy)

{

this->issupported = dummy;

}

void Piece::setInCheck(bool dummy)

{

this->incheck = dummy;

}

void Piece::addMoves(int dummy)

{

this->moves += dummy;

}

void Piece::setPieceValue(int dummy)

{

this->piecevalue = dummy;

}

#include "chessheader.h"

Player::Player()

{

playermove[0] = "-1";

playermove[1] = "-1";

playermove[2] = "-1";

playermove[3] = "-1";

}

void Player::setPlayerMove(int val, string dummy)

{

playermove[val] = dummy;

if (val == 0)

{

this->verifyPlayerMoveStart();

}

else if (val == 2)

{

this->verifyPlayerMoveEnd();

}

}

bool Player::checkAlpha(string playermove)

{

//declare flag

bool alphafound = false;

if (playermove[0] > 96 && playermove[0] < 105)

{

alphafound = true;

}

//return flag

return alphafound;

}

bool Player::checkNum(string playermove)

{

//declare flag

bool numfound = false;

// search if the number is within 0-9 range

if(playermove[1] > 48 && playermove[1] < 57)

{

numfound = true;

}

//return flag

return numfound;

}

void Player::getPlayerMoveStart()

{

string dummy;

cout << "Enter the start coordinates. ex: 'e2'";

cin >> dummy;

this->setPlayerMove(0, dummy);

}

void Player::verifyPlayerMoveStart()

{

static bool checkpoints[2] = {0,0};

//checkpoints

checkpoints[0] = this->checkAlpha(playermove[0]);

checkpoints[1] = this->checkNum(playermove[0]);

//checking if the coordinate annotation is correct

while(!(checkpoints[0] && checkpoints[1]))

{

if (!checkpoints[0])

{

cout << "Your alphanumerical character is incorrect";

}

else if (!checkpoints[1])

{

cout << "Your numerical value is incorrect";

}

cout << endl;

this->getPlayerMoveStart();

this->verifyPlayerMoveStart();

}

playermove[1] = this->convertAtlasCoordinatesToNumeric(playermove[0]);

}

string Player::convertAtlasCoordinatesToNumeric(string playermove)

{

playermove[0] -= 49;

//checks for single integer since conversion to int truncates the zero

if (playermove[0] == '0')

{

if (playermove[1] == '1')

{

return "00";

}

else if (playermove[1] == '2')

{

return "01";

}

else if (playermove[1] == '3')

{

return "02";

}

else if (playermove[1] == '4')

{

return "03";

}

else if (playermove[1] == '5')

{

return "04";

}

else if (playermove[1] == '6')

{

return "05";

}

else if (playermove[1] == '7')

{

return "06";

}

else if (playermove[1] == '8')

{

return "07";

}

}

return to\_string(stoi(playermove)-1);

}

void Player::getPlayerMoveEnd()

{

string dummy;

cout << "Enter the end coordinates. ex: 'e4'";

cin >> dummy;

this->setPlayerMove(2, dummy);

}

void Player::verifyPlayerMoveEnd()

{

static bool checkpoints[2] = {0,0};

//checkpoints

checkpoints[0] = this->checkAlpha(playermove[2]);

checkpoints[1] = this->checkNum(playermove[2]);

//checking if the coordinate annotation is correct

while(!(checkpoints[0] && checkpoints[1]))

{

if (!checkpoints[0])

{

cout << "Your alphanumerical character is incorrect";

}

else if (!checkpoints[1])

{

cout << "Your numerical value is incorrect";

}

cout << endl;

this->getPlayerMoveEnd();

this->verifyPlayerMoveEnd();

}

playermove[3] = this->convertAtlasCoordinatesToNumeric(playermove[2]);

}

#include "Bishop.h"

Bishop::Bishop(bool white, int a, int b, int pieceval)

{

this->position[0] = a;

this->position[1] = b;

this->captured = false;

this->white = white;

this->moves = 0;

this->issupported = false;

this->piecevalue = pieceval;

this->incheck = false;

}

bool Bishop::checkMovement(tiles board[8][8], int startmoverow, int startmovecol, int endmoverow, int endmovecol, vector<Piece\*>& piece, bool intentcapture)

{

bool valid = false;

// cout << " " << startmoverow << " " << startmovecol << " " << endmoverow << " " << endmovecol;

if (abs(endmoverow - startmoverow) == abs(endmovecol - startmovecol) && !this->checkBlock(startmoverow, startmovecol, endmoverow, endmovecol, board, piece))

{

valid = true;

}

if (endmoverow - startmoverow == 0 && endmovecol - startmovecol == 0)

{

valid = false;

}

// valid?cout<<" valid":cout<<"";

// cout << endl;

return valid;

}

bool Bishop::checkBlock(int startmoverow, int startmovecol, int endmoverow, int endmovecol, tiles board[8][8], vector<Piece\*>& piece)

{

bool blocked = false;

if (endmoverow - startmoverow > 0 && endmovecol - startmovecol > 0)

{

while (startmoverow+1 != endmoverow && !blocked)

{

if (board[startmoverow+1][startmovecol+1].getOccupied())

{

blocked = true;

}

startmoverow++;

startmovecol++;

}

}

else if (endmoverow - startmoverow < 0 && endmovecol - startmovecol > 0)

{

while (startmoverow-1 != endmoverow && !blocked)

{

if (board[startmoverow-1][startmovecol+1].getOccupied())

{

blocked = true;

}

startmoverow--;

startmovecol++;

}

}

else if (endmoverow - startmoverow > 0 && endmovecol - startmovecol < 0)

{

while (startmoverow+1 != endmoverow && !blocked)

{

if (board[startmoverow+1][startmovecol-1].getOccupied())

{

blocked = true;

}

startmoverow++;

startmovecol--;

}

}

else if (endmoverow - startmoverow < 0 && endmovecol - startmovecol < 0)

{

while (startmoverow-1 != endmoverow && !blocked)

{

if (board[startmoverow-1][startmovecol-1].getOccupied())

{

blocked = true;

}

startmoverow--;

startmovecol--;

}

}

return blocked;

}

Bishop::Bishop(Bishop\* obj)

{

white = obj->white;

moves = obj->moves;

captured = obj->captured;

for (int i = 0; i < 64; i++)

{

possible\_moves[i] = obj->possible\_moves[i];

}

position[0] = obj->position[0];

position[1] = obj->position[1];

piecevalue = obj->piecevalue;

issupported = obj->issupported;

incheck = obj->incheck;

}

#include "chessheader.h"

void updateGameState(bool whiteturn, Game& game, vector<Piece\*>& piece)

{

//set flag and checkpoints

bool gameEnd = false;

bool endConditions[3] = {0,0,0};

game.generatePossibleMoves(piece, whiteturn);

endConditions[0] = game.isCheckmate(piece, whiteturn);

endConditions[1] = game.isStalemate(piece, whiteturn);

endConditions[2] = game.isInsufficient(piece);

//check if any is true and output respective message

if (endConditions[0])

{

gameEnd = true;

cout << "Checkmate!" << endl;

!whiteturn?cout<<"White":cout<<"Black";

cout << " wins the game! << endl" << endl;

}

else if (endConditions[1])

{

gameEnd = true;

cout << "Stalemate! << endl" << endl;

}

else if (endConditions[2])

{

gameEnd = true;

cout << "Insufficient material" << endl;

}

//return flag

game.setGameState(gameEnd);

}

#include "chessheader.h"

int pieceSearchArrayPosition(int row, int col, vector<Piece\*>& piece)

{

//set flag

int position = -1;

for (int i = 0; i < 32; i++)

{

if(piece[i]->getPosition(0) == row && piece[i]->getPosition(1) == col)

{

position = i;

}

}

//returning position if found

return position;

}

#include <bits/stdc++.h>

#include "Rook.h"

#include "tiles.h"

using namespace std;

Rook::Rook(bool white, int a, int b, int pieceval)

{

this->position[0] = a;

this->position[1] = b;

this->captured = false;

this->white = white;

this->moves = 0;

this->issupported = false;

this->piecevalue = pieceval;

this->incheck = false;

}

bool Rook::checkMovement(tiles board[8][8], int startmoverow, int startmovecol, int endmoverow, int endmovecol, vector<Piece\*>& piece, bool intentcapture)

{

bool valid = false;

// cout << " " << startmoverow << " " << startmovecol << " " << endmoverow << " " << endmovecol;

if (endmoverow - startmoverow == 0 || endmovecol - startmovecol == 0)

{

if (!this->checkBlock(startmoverow, startmovecol, endmoverow, endmovecol, board, piece))

{

valid = true;

}

}

if (endmoverow - startmoverow == 0 && endmovecol - startmovecol == 0)

{

valid = false;

}

// valid?cout<<" valid":cout<<"";

// cout << endl;

return valid;

}

bool Rook::checkBlock(int startmoverow, int startmovecol, int endmoverow, int endmovecol, tiles board[8][8], vector<Piece\*>& piece)

{

bool blocked = false;

if (endmoverow - startmoverow == 0 && endmovecol - startmovecol > 0)

{

while (startmovecol+1 != endmovecol && !blocked)

{

if (board[startmoverow][startmovecol+1].getOccupied())

{

blocked = true;

}

startmovecol++;

}

}

else if (endmoverow - startmoverow == 0 && endmovecol - startmovecol < 0)

{

while (startmovecol-1 != endmovecol && !blocked)

{

if (board[startmoverow][startmovecol-1].getOccupied())

{

blocked = true;

}

startmovecol--;

}

}

else if (endmoverow - startmoverow > 0 && endmovecol - startmovecol == 0)

{

while (startmoverow+1 != endmoverow && !blocked)

{

if (board[startmoverow+1][startmovecol].getOccupied())

{

blocked = true;

}

startmoverow++;

}

}

else if (endmoverow - startmoverow < 0 && endmovecol - startmovecol == 0)

{

while (startmoverow-1 != endmoverow && !blocked)

{

if (board[startmoverow-1][startmovecol].getOccupied())

{

blocked = true;

}

startmoverow--;

}

}

return blocked;

}

Rook::Rook(Rook\* obj)

{

white = obj->white;

moves = obj->moves;

captured = obj->captured;

for (int i = 0; i < 64; i++)

{

possible\_moves[i] = obj->possible\_moves[i];

}

position[0] = obj->position[0];

position[1] = obj->position[1];

piecevalue = obj->piecevalue;

issupported = obj->issupported;

incheck = obj->incheck;

}

#include "Queen.h"

#include <bits/stdc++.h>

#include "tiles.h"

using namespace std;

Queen::Queen(bool white, int a, int b, int pieceval)

{

this->position[0] = a;

this->position[1] = b;

this->captured = false;

this->white = white;

this->moves = 0;

this->issupported = false;

this->piecevalue = pieceval;

this->incheck = false;

}

bool Queen::checkMovement(tiles board[8][8], int startmoverow, int startmovecol, int endmoverow, int endmovecol, vector<Piece\*>& piece, bool intentcapture)

{

bool valid = false;

// cout << " " << startmoverow << " " << startmovecol << " " << endmoverow << " " << endmovecol;

if (abs(endmoverow - startmoverow) == 1 && abs(endmovecol - startmovecol) == 0)

{

valid = true;

}

else if (abs(endmoverow - startmoverow) == 0 && abs(endmovecol - startmovecol) == 1)

{

valid = true;

}

else if (abs(endmoverow - startmoverow) == 1 && abs(endmovecol - startmovecol) == 1)

{

valid = true;

}

else if (endmoverow - startmoverow == 0 || endmovecol - startmovecol == 0)

{

if (!this->checkBlock(startmoverow, startmovecol, endmoverow, endmovecol, board, piece))

{

valid = true;

}

}

else if (abs(endmoverow - startmoverow) == abs(endmovecol - startmovecol))

{

if (!this->checkBlock(startmoverow, startmovecol, endmoverow, endmovecol, board, piece))

{

valid = true;

}

}

if (endmoverow - startmoverow == 0 && endmovecol - startmovecol == 0)

{

valid = false;

}

// valid?cout<<" valid":cout<<"";

// cout << endl;

return valid;

}

bool Queen::checkBlock(int startmoverow, int startmovecol, int endmoverow, int endmovecol, tiles board[8][8], vector<Piece\*>& piece)

{

bool blocked = false;

if (endmoverow - startmoverow > 0 && endmovecol - startmovecol > 0)

{

while (startmoverow+1 != endmoverow && !blocked)

{

if (board[startmoverow+1][startmovecol+1].getOccupied())

{

blocked = true;

}

startmoverow++;

startmovecol++;

}

}

else if (endmoverow - startmoverow < 0 && endmovecol - startmovecol > 0)

{

while (startmoverow-1 != endmoverow && !blocked)

{

if (board[startmoverow-1][startmovecol+1].getOccupied())

{

blocked = true;

}

startmoverow--;

startmovecol++;

}

}

else if (endmoverow - startmoverow > 0 && endmovecol - startmovecol < 0)

{

while (startmoverow+1 != endmoverow && !blocked)

{

if (board[startmoverow+1][startmovecol-1].getOccupied())

{

blocked = true;

}

startmoverow++;

startmovecol--;

}

}

else if (endmoverow - startmoverow < 0 && endmovecol - startmovecol < 0)

{

while (startmoverow-1 != endmoverow && !blocked)

{

if (board[startmoverow-1][startmovecol-1].getOccupied())

{

blocked = true;

}

startmoverow--;

startmovecol--;

}

}

else if (endmoverow - startmoverow == 0 && endmovecol - startmovecol > 0)

{

while (startmovecol+1 != endmovecol && !blocked)

{

if (board[startmoverow][startmovecol+1].getOccupied())

{

blocked = true;

}

startmovecol++;

}

}

else if (endmoverow - startmoverow == 0 && endmovecol - startmovecol < 0)

{

while (startmovecol-1 != endmovecol && !blocked)

{

if (board[startmoverow][startmovecol-1].getOccupied())

{

blocked = true;

}

startmovecol--;

}

}

else if (endmoverow - startmoverow > 0 && endmovecol - startmovecol == 0)

{

while (startmoverow+1 != endmoverow && !blocked)

{

if (board[startmoverow+1][startmovecol].getOccupied())

{

blocked = true;

}

startmoverow++;

}

}

else if (endmoverow - startmoverow < 0 && endmovecol - startmovecol == 0)

{

while (startmoverow-1 != endmoverow && !blocked)

{

if (board[startmoverow-1][startmovecol].getOccupied())

{

blocked = true;

}

startmoverow--;

}

}

return blocked;

}

Queen::Queen(Queen\* obj)

{

white = obj->white;

moves = obj->moves;

captured = obj->captured;

for (int i = 0; i < 64; i++)

{

possible\_moves[i] = obj->possible\_moves[i];

}

position[0] = obj->position[0];

position[1] = obj->position[1];

piecevalue = obj->piecevalue;

issupported = obj->issupported;

incheck = obj->incheck;

}

#include "chessheader.h"

#include <bits/stdc++.h>

#include "Pawn.h"

#include "tiles.h"

using namespace std;

int pieceSearchArrayPosition(int row, int col, vector<Piece\*>& piece);

Pawn::Pawn(bool white, int pos, int pieceval)

{

this->ispawn = true;

this->captured = false;

this->white = white;

this->moves = 0;

this->issupported = false;

this->piecevalue = pieceval;

this->incheck = false;

if (white)

{

this->position[0] = 1;

this->position[1] = pos;

}

else if (!white)

{

this->position[0] = 6;

this->position[1] = pos;

}

}

bool Pawn::checkMovement(tiles board[8][8], int startmoverow, int startmovecol, int endmoverow, int endmovecol, vector<Piece\*>& piece, bool intentcapture)

{

bool valid = false;

int pawnpiecevalue = this->getPieceValue();

if (pawnpiecevalue != 0)

{

if (pawnpiecevalue == QUEEN)

{

valid = piece[3]->checkMovement(board, startmoverow, startmovecol, endmoverow, endmovecol, piece, intentcapture);

}

else if (pawnpiecevalue == KNIGHT)

{

valid = piece[1]->checkMovement(board, startmoverow, startmovecol, endmoverow, endmovecol, piece, intentcapture);

}

else if (pawnpiecevalue == BISHOP)

{

valid = piece[2]->checkMovement(board, startmoverow, startmovecol, endmoverow, endmovecol, piece, intentcapture);

}

else if (pawnpiecevalue == ROOK)

{

valid = piece[0]->checkMovement(board, startmoverow, startmovecol, endmoverow, endmovecol, piece, intentcapture);

}

return valid;

}

if (this->white)

{

if (endmoverow - startmoverow == 2 && endmovecol - startmovecol == 0 && !intentcapture)

{

if (this->getMoves() == 0 && !board[endmoverow][endmovecol].getOccupied() && !board[endmoverow-1][endmovecol].getOccupied())

{

//cout << "double move" <<endl;

valid = true;

}

}

//single advance

else if (endmoverow - startmoverow == 1 && endmovecol - startmovecol == 0 && !board[endmoverow][endmovecol].getOccupied() && !intentcapture)

{

valid = true;

}

//capture

else if (endmoverow - startmoverow == 1 && endmovecol - startmovecol == 1)

{

if (board[endmoverow][endmovecol].getOccupiedBlack() == true || intentcapture)

{

valid = true;

}

//enpassant

else if (board[endmoverow-1][endmovecol].getOccupiedBlack() && piece[pieceSearchArrayPosition(endmoverow-1, endmovecol, piece)]->getMoves() == 1 && piece[pieceSearchArrayPosition(endmoverow-1, endmovecol, piece)]->getPosition(0) == 4)

{

valid = true;

}

}

//diagonal capture

else if (endmoverow - startmoverow == 1 && endmovecol - startmovecol == -1)

{

if (board[endmoverow][endmovecol].getOccupiedBlack() == true || intentcapture)

{

valid = true;

}

//enpassant

else if (board[endmoverow-1][endmovecol].getOccupiedBlack() && piece[pieceSearchArrayPosition(endmoverow-1, endmovecol, piece)]->getMoves() == 1 && piece[pieceSearchArrayPosition(endmoverow-1, endmovecol, piece)]->getPosition(0) == 4)

{

valid = true;

}

}

}

else if (this->white == false)

{

if (endmoverow - startmoverow == -2 && endmovecol - startmovecol == 0 && !intentcapture)

{

if (this->getMoves() == 0 && !board[endmoverow][endmovecol].getOccupied() && !board[endmoverow+1][endmovecol].getOccupied())

{

valid = true;

}

}

else if (endmoverow - startmoverow == -1 && endmovecol - startmovecol == 0 && !board[endmoverow][endmovecol].getOccupied() && !intentcapture)

{

valid = true;

}

else if (endmoverow - startmoverow == -1 && endmovecol - startmovecol == -1)

{

if (board[endmoverow][endmovecol].getOccupiedWhite() == true || intentcapture)

{

valid = true;

}

else if (board[endmoverow+1][endmovecol].getOccupiedWhite() && piece[pieceSearchArrayPosition(endmoverow+1, endmovecol, piece)]->getMoves() == 1 && piece[pieceSearchArrayPosition(endmoverow-1, endmovecol, piece)]->getPosition(0) == 3)

{

valid = true;

}

}

else if (endmoverow - startmoverow == -1 && endmovecol - startmovecol == 1)

{

if (board[endmoverow][endmovecol].getOccupiedWhite() == true || intentcapture)

{

valid = true;

}

else if (board[endmoverow+1][endmovecol].getOccupiedWhite() && piece[pieceSearchArrayPosition(endmoverow+1, endmovecol, piece)]->getMoves() == 1 && piece[pieceSearchArrayPosition(endmoverow-1, endmovecol, piece)]->getPosition(0) == 3)

{

valid = true;

}

}

}

if (endmoverow - startmoverow == 0 && endmovecol - startmovecol == 0)

{

valid = false;

}

return valid;

}

Pawn::Pawn(Pawn\* obj)

{

white = obj->white;

moves = obj->moves;

captured = obj->captured;

for (int i = 0; i < 64; i++)

{

possible\_moves[i] = obj->possible\_moves[i];

}

position[0] = obj->position[0];

position[1] = obj->position[1];

piecevalue = obj->piecevalue;

issupported = obj->issupported;

incheck = obj->incheck;

}

#include "Knight.h"

#include "tiles.h"

Knight::Knight(bool white, int a, int b, int pieceval)

{

this->captured = false;

this->white = white;

this->moves = 0;

this->issupported = false;

this->position[0] = a;

this->position[1] = b;

this->piecevalue = pieceval;

this->incheck = false;

}

bool Knight::checkMovement(tiles board[8][8], int startmoverow, int startmovecol, int endmoverow, int endmovecol, vector<Piece\*>& piece, bool intentcapture)

{

bool valid = false;

// cout << " " << startmoverow << " " << startmovecol << " " << endmoverow << " " << endmovecol;

if (abs(endmoverow - startmoverow) == 2 && abs(endmovecol - startmovecol) == 1)

{

valid = true;

}

else if (abs(endmoverow -startmoverow) == 1 && abs(endmovecol - startmovecol) == 2)

{

valid = true;

}

if (endmoverow - startmoverow == 0 && endmovecol - startmovecol == 0)

{

valid = false;

}

// valid?cout<<" valid":cout<<"";

// cout << endl;

return valid;

}

Knight::Knight(Knight\* obj)

{

white = obj->white;

moves = obj->moves;

captured = obj->captured;

for (int i = 0; i < 64; i++)

{

possible\_moves[i] = obj->possible\_moves[i];

}

position[0] = obj->position[0];

position[1] = obj->position[1];

piecevalue = obj->piecevalue;

issupported = obj->issupported;

incheck = obj->incheck;

}

#include "King.h"

#include "tiles.h"

#include <bits/stdc++.h>

King::King(bool white, int a, int b, int pieceval)

{

this->position[0] = a;

this->position[1] = b;

this->captured = false;

this->white = white;

this->moves = 0;

this->incheck = false;

this->piecevalue = pieceval;

this->incheck = false;

}

bool King::checkMovement(tiles board[8][8], int startmoverow, int startmovecol, int endmoverow, int endmovecol, vector<Piece\*>& piece, bool intentcapture)

{

bool valid = false;

// cout << " " << startmoverow << " " << startmovecol << " " << endmoverow << " " << endmovecol;

if (this->white)

{

if (abs(endmoverow - startmoverow) == 1 && abs(endmovecol - startmovecol) == 0)

{

valid = true;

}

else if (abs(endmoverow - startmoverow) == 0 && abs(endmovecol - startmovecol) == 1)

{

valid = true;

}

else if (abs(endmoverow - startmoverow) == 1 && abs(endmovecol - startmovecol) == 1)

{

valid = true;

}

else if (endmoverow - startmoverow == 0 && endmovecol - startmovecol == 2)

{

if (this->getMoves() == 0 && !board[startmoverow][startmovecol].getUnderAttackBlack() && !board[startmoverow][startmovecol+1].getUnderAttackBlack() && !board[startmoverow][startmovecol+2].getUnderAttackBlack())

{

if (!board[startmoverow][startmovecol+1].getOccupied() && !board[startmoverow][startmovecol+2].getOccupied())

{

if (piece[15]->getMoves() == 0)

{

valid = true;

}

}

}

}

else if (endmoverow - startmoverow == 0 && endmovecol - startmovecol == -2)

{

if (this->getMoves() == 0 && !board[startmoverow][startmovecol].getUnderAttackBlack() && !board[startmoverow][startmovecol-1].getUnderAttackBlack() && !board[startmoverow][startmovecol-2].getUnderAttackBlack())

{

if (!board[startmoverow][startmovecol-1].getOccupied() && !board[startmoverow][startmovecol-2].getOccupied() && !board[startmoverow][startmovecol-3].getOccupied())

{

if (piece[8]->getMoves() == 0)

{

valid = true;

}

}

}

}

}

else if (this->white == false)

{

if (abs(endmoverow - startmoverow) == 1 && abs(endmovecol - startmovecol) == 0)

{

valid = true;

}

else if (abs(endmoverow - startmoverow) == 0 && abs(endmovecol - startmovecol) == 1)

{

valid = true;

}

else if (abs(endmoverow - startmoverow) == 1 && abs(endmovecol - startmovecol) == 1)

{

valid = true;

}

//castling

else if (endmoverow - startmoverow == 0 && endmovecol - startmovecol == 2)

{

if (this->getMoves() == 0 && !board[startmoverow][startmovecol].getUnderAttackWhite() && !board[startmoverow][startmovecol+1].getUnderAttackWhite() && !board[startmoverow][startmovecol+2].getUnderAttackWhite())

{

if (!board[startmoverow][startmovecol+1].getOccupied() && !board[startmoverow][startmovecol+2].getOccupied())

{

if (piece[7]->getMoves() == 0)

{

valid = true;

}

}

}

}

else if (endmoverow - startmoverow == 0 && endmovecol - startmovecol == -2)

{

if (this->getMoves() == 0 && !board[startmoverow][startmovecol].getUnderAttackWhite() && !board[startmoverow][startmovecol-1].getUnderAttackWhite() && !board[startmoverow][startmovecol-2].getUnderAttackWhite())

{

if (!board[startmoverow][startmovecol-1].getOccupied() && !board[startmoverow][startmovecol-2].getOccupied() && !board[startmoverow][startmovecol-3].getOccupied())

{

if (piece[0]->getMoves() == 0)

{

valid = true;

}

}

}

}

}

if (endmoverow - startmoverow == 0 && endmovecol - startmovecol == 0)

{

valid = false;

}

// valid?cout<<" valid":cout<<"";

// cout << endl;

return valid;

}

King::King(King\* obj)

{

white = obj->white;

moves = obj->moves;

captured = obj->captured;

for (int i = 0; i < 64; i++)

{

possible\_moves[i] = obj->possible\_moves[i];

}

position[0] = obj->position[0];

position[1] = obj->position[1];

piecevalue = obj->piecevalue;

issupported = obj->issupported;

incheck = obj->incheck;

}