**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. I also think that this project is perfect for a transition to object-oriented programming because a chess program should easily convert to objects and classes.

**Summary**

The project pasted at the end of this write-up is about 2000 lines in total without comments. I used a total of 40 variables including the data types inside of the structure array. Overall, I’m glad I took on the chess project as my first project. I had no help in this project besides borrowing the concept of checking for all possible moves from a user on stack overflow and some advice on checking for checks from Professor Lehr. I also tried to challenge myself by attempting to follow the MVC design pattern. I realized that the MVC pattern was pretty standard in most games anyways functionally. Towards the end of the project, I had more and more difficulty debugging the project because the simulations would run too many operations for me to keep count. I hope to learn how to use a proper debugger. Another thing I hope to learn more about it how to organize the model of a program. Time after time, I found myself having no clear-cut path to get things done. I know like my project is heavily unoptimized and can clearly be done better. For most of the project, I stayed within the chapters of the book that we reviewed. There was one part of the project where I did use a set to check if the piecedata was unique.

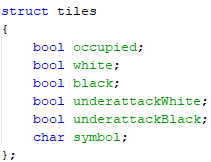
**Basics of Chess**

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**

When first starting out with the chess program, or any program for that matter, the first thing to do is initialize the data types. For the sake of efficiency, I will not cover “convenience function” which have no relation to the logic of chess and are mainly used for outputting instructions and such. This section will begin with an explanation the major variables and structures.

**The Tiles Structure**

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

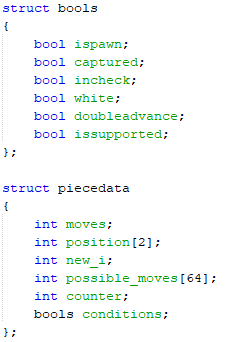
The occupied Boolean governs whether or not the tile is being occupied by a piece, whether white or black. This is necessary to implement because it allows the computer later on to recognize if a piece is being blocked in their movement path. The occupied Boolean also helps us later on when we need to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

**The Piecedata & Bools Structure**



This structure contains all the data for any given piece on the board. There is a nested structure which contains all the Booleans, making it more manageable later on. I’ll first explain the “bools” structure. An important thing to note is that the movement of a piece is dictated by its location in the array of the piecedata structures.

* The ispawn bool allows us to determine if the piece is a pawn or not. Seeing as half of all pieces on the board are pawns, this is important, especially when considering how tricky the movements a pawn can make are.
* 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 third integer is new\_i, and it is initialized to -1 for all pieces. However, when a pawn is promoted, it gets a new\_i value respective to the piece it is promoted to. As mentioned earlier, the movement of a piece is determined by its position in the array of piecedata structures. When a pawn is promoted, the new\_i allows the program to determine that the pawn can move differently without any changes to the array of structures.
* 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.
* The counter integer is my solution to enpassant validity. The counter is initialized to zero. When a pawn makes a double advance, the double advance boolean is set to true, which increments the counter integer. The counter integer will increment every move that the double advance boolean is true. When the counter integer increments a second time, the Doubleadvance boolean is set to false. Enpassants are only allowed when the double advance boolean is true.

**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.

**Function List**

**Convenience Functions**

string convertAtlasCoordinatesToNumeric(string playermove, char alphaCoordinates[]);

void outputName(bool whiteturn, string playername[]);

int pieceSearchArrayPosition(int row, int col, piecedata piece[]);

bool isWhite(tiles board[8][8], int row, int col, char whitePieces[]);

string convertNumericToAtlasCoordinates(int row, int col, char alphaCoordinates[]);

**Basic Initialization – Model & View**

void outputInstructions();

void displayBoard(tiles board[8][8], char alphaCoordinates[]);

void initializeBoardPieces(tiles board[8][8], char blackPieces[], char whitePieces[], piecedata piece[]);

void readInPieceDataNames(char blackPieces[], char whitePieces[], ifstream &input, ifstream &names, string playernames[]);

**Choosing A Piece - Control**

void getPlayerMoveStart(string playermove[4], bool whiteturn);

void verifyPlayerMoveStart(string playermove[4], bool whiteturn, tiles board[8][8], char blackPieces[], char whitePieces[], char alphaCoordinates[], piecedata piece[]);

bool VPMCheckPiece(string playermove, tiles board[8][8], char blackPieces[], char whitePieces[]);

bool VPMCheckAlpha(string playermove, char alphaCoordinates[]);

bool VPMCheckNum(string playermove);

bool VPMCheckColor(string playermove, bool whiteturn, tiles board[8][8], char blackPieces[], char whitePieces[]);

**Sending A Piece – Control & Model**

void getPlayerMoveEnd(string playermove[]);

void verifyPlayerMoveEnd(tiles board[8][8], char blackPieces[], char whitePieces[], string playermove[], piecedata piece[], bool whiteturn, char alphaCoorindates[]);

bool VPMCheckMovement(tiles board[8][8], int startmoverow, int startmovecol, int endmoverow, int endmovecol, piecedata piece[], bool capture);

bool VPMisBlocked(int i, int startmoverow, int startmovecol, int endmoverow, int endmovecol, tiles board[8][8], piecedata piece[]);

bool VPMCheckCapture(bool whiteturn, string playermove[], tiles board[8][8], bool simulation, piecedata piece[], bool realtimewhiteturn);

bool VPMKingMoveSafe(tiles board[8][8], piecedata piece[], int endmoverow, int endmovecol, bool whiteturn);

bool verifyPlayerMove(tiles board[8][8], char blackPieces[8], char whitePieces[8], string playermove[], piecedata piece[], bool whiteturn, char alphaCoordinates[], bool hypo, bool simulation, bool realtimewhiteturn);

bool VPMkingNotinCheck(tiles board[8][8], piecedata piece[], string playermove[], char alphaCoordinates[], char blackPieces[], char whitePieces[], bool whiteturn, bool hypo, bool simulation);

**End Condition Checkers - Model**

bool gameEnd(bool resignation, bool whiteturn, bool offerdraw, piecedata piece[], tiles board[8][8], char whitePieces[], char blackPieces[], char alphaCoordinates[], string playernames[], int moves);

bool isCheckmate(tiles board[8][8], piecedata piece[], char blackPieces[], char whitePieces[], char alphaCoordinates[], bool whiteturn);

bool isStalemate(piecedata piece[], bool whiteturn);

bool is50Moves(int moves);

bool isInsufficient(piecedata piece[]);

**Checkmate Conditions - Model**

bool isCheck(tiles board[8][8], piecedata piece[], int endmoverow, int endmovecol, bool white, bool hypo);

bool cannotMove(piecedata piece[], int K);

//output file stream functions - output

void updateMoveList(string moveList[], int &moves, string playermove[]);

void printMoveList(string\* listptr, int moves, string moveList[], bool whiteturn, char whitePieces[], char blackPieces[]);

string\* createMoveList(int moves);

void outputData(char whitePieces[], char blackPieces[], piecedata piece[]);

**Model**

void updateTiles(tiles board[8][8], char blackPieces[], char whitePieces[], string playermove[], piecedata piece[], char alphaCoordinates[], bool whiteturn, bool simulation);

void pawnUpdate(piecedata piece[], char blackPieces[], char whitePieces[], tiles board[8][8], bool simulation);

void pawnPromotion(piecedata piece[], char blackPieces[], char whitePieces[], tiles board[8][8], bool isWhite, int i);

**Simulation**

int movePossible(int i, int k, tiles board[8][8], piecedata piece[], char blackPieces[], char whitePieces[], char alphaCoordinates[], int c, bool whiteturn);

void generatePossibleMoves(piecedata piece[], tiles board[8][8], char blackPieces[], char whitePieces[], char alphaCoordinates[], bool whiteturn);

void findSupports(piecedata piece[], tiles board[8][8]);

tiles copyTiles(tiles board[8][8], int i, int j);

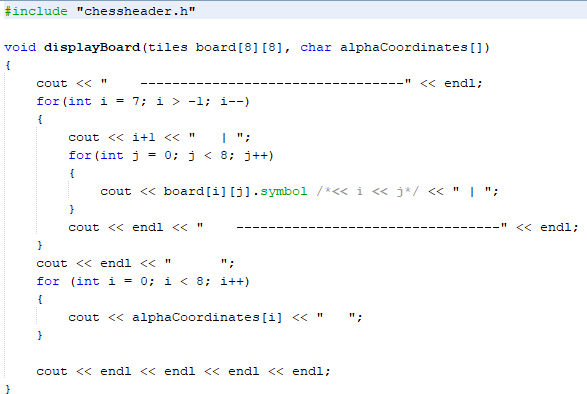
**Important and Recurring Variables**

|  |  |  |  |
| --- | --- | --- | --- |
| 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 |

**My Process of Thought in the Implementation of Chess**

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

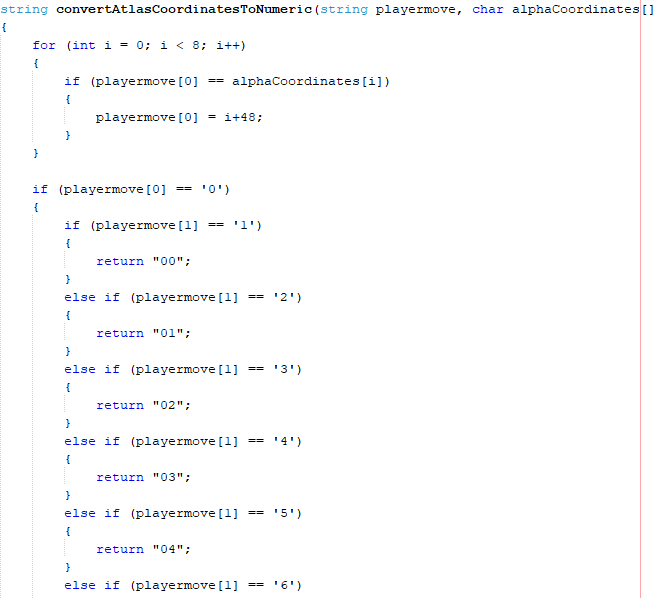
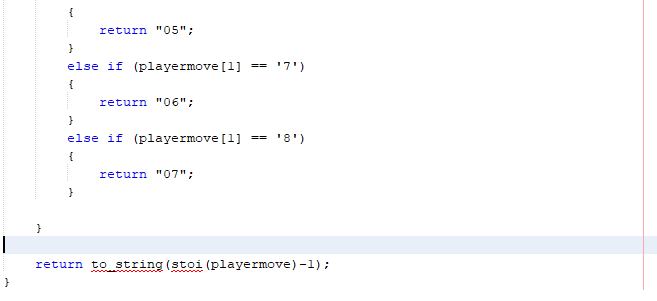
**The Display Board Function**



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**

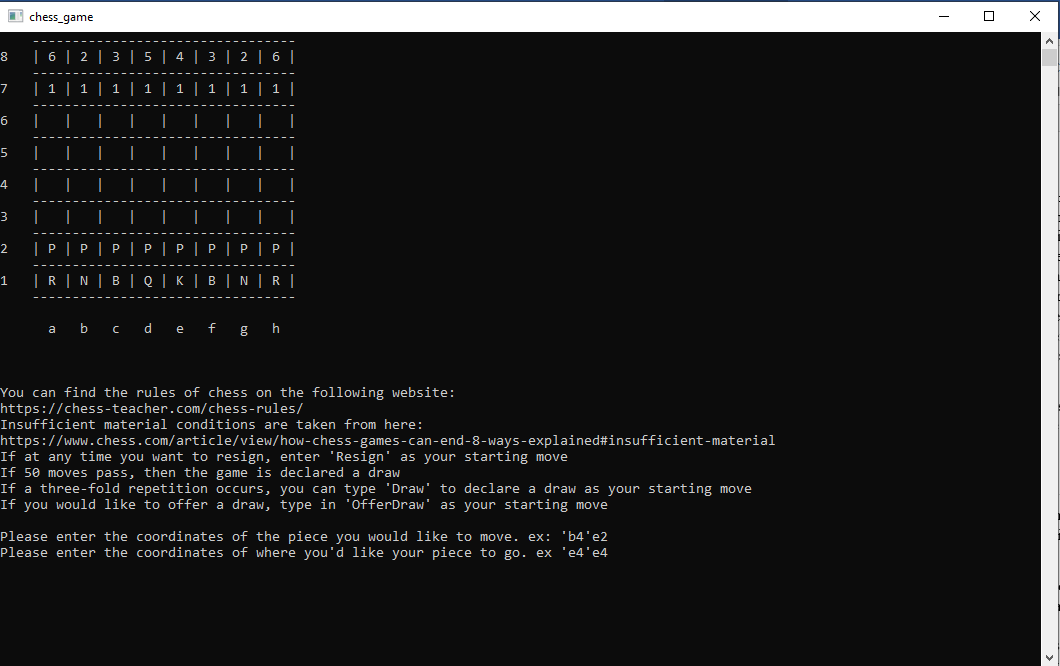
The user input in this program is twofold. Each player must input two coordinates. The first input takes the coordinate of the piece they want to move. The second one will tell the program where the player wants to move the piece they had selected. In the earliest versions of my program, I wanted to 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:



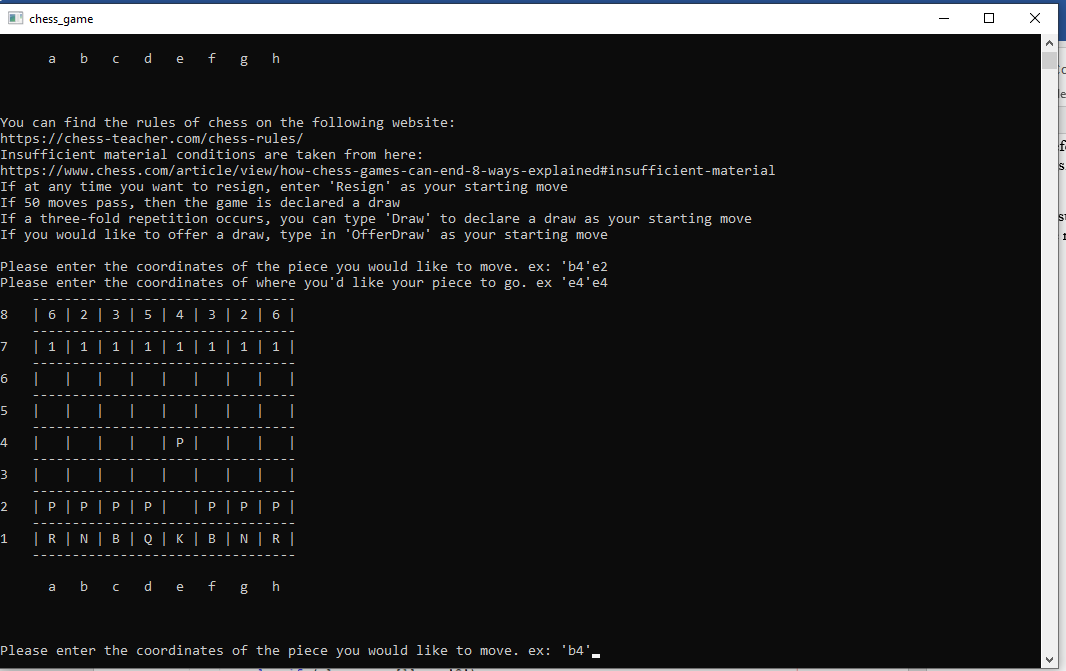
This function will simply go through a predefined c\_ string array of the alphabet coordinates which indicated the columns on the chess board. When it finds the respective letter that matches, it converts the value of the alpha coordinate to a numeric representation that is still contained as a character 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



The next issue that needed to be addressed was a player choosing to resign or offer a draw. I did this simply by having if statements before the move verification function in order to catch these unique valid inputs. The if statements would set a boolean indicating that a resignation was inputted or a draw offer was inputted. Then, a continue statement is executed, allowing the program to skip over the input of the second move and both verification steps.

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

\*Note that all subfunction of verifying player input/movement is preceded with the acronym “VPM”.

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 piecedata piece[] array.

**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”.

Logically, my solution would have to pair the double advance movement with the enpassant check. My solution involved initializing all pawns to Doubleadvance false. They can only double advance if their move count is exactly zero. Once they do, the double advance movement is set to. As soon as the double advance boolean is marked true, a counter will increment every consequent turn. As soon as the counter hits 1 (post increment in the pawnUpdate function), then Doubleadvance is set to false. Enpassants are only valid if the Doubleadvance boolean is set to true.

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 endrow and endcol 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 new\_i variable covered earlier. The new\_i variable is initially initialized to -1, but when a pawn promotion occurs, it is initialized to the “pseudo position” of the piece it is promoted to. Before the mess of if statements, there is a simple check to see if new\_i is not equal to -1, indicating that a promotion occurred. The “pseudo position” i is assigned the value of new\_i

**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 VPMisBlocked. The blocked function only applies to rooks, bishops, and the queen. The knight cannot be blocked, and the pawn and king are already hardcoded. First, the function is passed the value of i. The value will dictate the movement tiles the function is supposed to check. Then, 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 VPMisBlocked function). However, if the destination tile is occupied, then the VPMCheckCapture 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 VPMCheckMovement 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 maintaining all the pawn conditions (including promotions). Then, the board is rescanned and all tiles under check are recorded into the tiles structure.

**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. The only end condition that I did not implement due to a lack of time was the threefold repetition. 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 VPMisBlocked function. In order to remedy this, I modified the VPMisBlocked function, and create a whole new function, the findSupports function.

**Pseudocode**

Main

{

While (!gameEnd)

{

Getplayermove

verifyplayermove

Updateboard

DisplayBoard

Switch turn

}

}

This is the basic pseudocode that my code is based off.

**Flowcharts**

Since the flowcharts are too large for this file as well as having too many (43 functions), the link will be on my GitHub repository in the project 1 folder.

**Concepts Used**

|  |  |
| --- | --- |
| Concept | Location |
| Pointer Variables | String \*listptr at the end of welcome.cc (main) |
| Arrays/Pointers | In printMoveList |
| Function Parameters | In printMoveList |
| Memory Allocation | In createMoveList |
| Return Parameters | In createMoveList |
| C Strings | whitePieces, blackPieces, alphaCoordinates |
| Strings | The playermove string |
| Arrays of Structs | updateTiles, several other functions |
| Nested Structures | updateTiles, several other functions |
| Function Argument | updateTiles, several other functions |
| Function Return | copyTiles |
| Pointers | outputData |
| Enumeration | In initializeBoardPieces and readInPieceDataNames |
| Formatting | printMoveList |
| Function Parameters | readInPieceDataNames |
| Member Functions | printMoveList uses .put() |
| Multiple Files | readInPieceDataNames |
| Binary Files | outputData |
| Records Structures | outputData |
| Random Access Files | outputData |
| Input / Output Simultaneous | printMoveList |

**Additional Features**

* Prints moves to text document
* Can read in custom piece symbols
* Outputs the winner
* Can read in custom names

**Known Bugs**

* Black h file Rook disappears from board sometimes
* Some stalemates do not work

**Program**

#include <bits/stdc++.h>

using namespace std;

struct bools

{

bool ispawn;

bool captured;

bool incheck;

bool white;

bool doubleadvance;

bool issupported;

};

struct piecedata

{

int moves;

int position[2];

int new\_i;

int possible\_moves[64];

int counter;

bools conditions;

};

struct tiles

{

bool occupied;

bool white;

bool black;

bool underattackWhite;

bool underattackBlack;

char symbol;

};

struct record

{

int totalmoveswhite;

int totalmovesblack;

};

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

//convenience functions

string convertAtlasCoordinatesToNumeric(string playermove, char alphaCoordinates[]);

void outputName(bool whiteturn, string playername[]);

int pieceSearchArrayPosition(int row, int col, piecedata piece[]);

bool isWhite(tiles board[8][8], int row, int col, char whitePieces[]);

string convertNumericToAtlasCoordinates(int row, int col, char alphaCoordinates[]);

//basic initialization

void outputInstructions();

void displayBoard(tiles board[8][8], char alphaCoordinates[]);

void initializeBoardPieces(tiles board[8][8], char blackPieces[], char whitePieces[], piecedata piece[]);

void readInPieceDataNames(char blackPieces[], char whitePieces[], ifstream &input, ifstream &names, string playernames[]);

//choosing a piece

void getPlayerMoveStart(string playermove[4], bool whiteturn);

void verifyPlayerMoveStart(string playermove[4], bool whiteturn, tiles board[8][8], char blackPieces[], char whitePieces[], char alphaCoordinates[], piecedata piece[]);

bool VPMCheckPiece(string playermove, tiles board[8][8], char blackPieces[], char whitePieces[]);

bool VPMCheckAlpha(string playermove, char alphaCoordinates[]);

bool VPMCheckNum(string playermove);

bool VPMCheckColor(string playermove, bool whiteturn, tiles board[8][8], char blackPieces[], char whitePieces[]);

//sending a piece

void getPlayerMoveEnd(string playermove[]);

void verifyPlayerMoveEnd(tiles board[8][8], char blackPieces[], char whitePieces[], string playermove[], piecedata piece[], bool whiteturn, char alphaCoorindates[]);

bool VPMCheckMovement(tiles board[8][8], int startmoverow, int startmovecol, int endmoverow, int endmovecol, piecedata piece[], bool capture);

bool VPMisBlocked(int i, int startmoverow, int startmovecol, int endmoverow, int endmovecol, tiles board[8][8], piecedata piece[]);

bool VPMCheckCapture(bool whiteturn, string playermove[], tiles board[8][8], bool simulation, piecedata piece[], bool realtimewhiteturn);

bool VPMKingMoveSafe(tiles board[8][8], piecedata piece[], int endmoverow, int endmovecol, bool whiteturn);

bool verifyPlayerMove(tiles board[8][8], char blackPieces[8], char whitePieces[8], string playermove[], piecedata piece[], bool whiteturn, char alphaCoordinates[], bool hypo, bool simulation, bool realtimewhiteturn);

bool VPMkingNotinCheck(tiles board[8][8], piecedata piece[], string playermove[], char alphaCoordinates[], char blackPieces[], char whitePieces[], bool whiteturn, bool hypo, bool simulation);

//end condition checkers

bool gameEnd(bool resignation, bool whiteturn, bool offerdraw, piecedata piece[], tiles board[8][8], char whitePieces[], char blackPieces[], char alphaCoordinates[], string playernames[], int moves);

bool isCheckmate(tiles board[8][8], piecedata piece[], char blackPieces[], char whitePieces[], char alphaCoordinates[], bool whiteturn);

bool isStalemate(piecedata piece[], bool whiteturn);

bool is50Moves(int moves);

bool isInsufficient(piecedata piece[]);

//checkmate conditions

bool isCheck(tiles board[8][8], piecedata piece[], int endmoverow, int endmovecol, bool white, bool hypo);

bool cannotMove(piecedata piece[], int K);

//output file stream functions

void updateMoveList(string moveList[], int &moves, string playermove[]);

void printMoveList(string\* listptr, int moves, string moveList[], bool whiteturn, char whitePieces[], char blackPieces[]);

string\* createMoveList(int moves);

void outputData(char whitePieces[], char blackPieces[], piecedata piece[]);

//model???

void updateTiles(tiles board[8][8], char blackPieces[], char whitePieces[], string playermove[], piecedata piece[], char alphaCoordinates[], bool whiteturn, bool simulation);

void pawnUpdate(piecedata piece[], char blackPieces[], char whitePieces[], tiles board[8][8], bool simulation);

void pawnPromotion(piecedata piece[], char blackPieces[], char whitePieces[], tiles board[8][8], bool isWhite, int i);

//simulation

int movePossible(int i, int k, tiles board[8][8], piecedata piece[], char blackPieces[], char whitePieces[], char alphaCoordinates[], int c, bool whiteturn);

void generatePossibleMoves(piecedata piece[], tiles board[8][8], char blackPieces[], char whitePieces[], char alphaCoordinates[], bool whiteturn);

void findSupports(piecedata piece[], tiles board[8][8]);

tiles copyTiles(tiles board[8][8], int i, int j);

int main()

{

ifstream pieces;

ifstream names;

ofstream output;

char whitePieces[6];

char blackPieces[6];

piecedata piece[32];

tiles board[8][8];

string playernames[2];

string moveList[250];

bool offerdraw = false;

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

int moves = 0;

bool whiteturn = true;

bool resignation = false;

char drawresponse;

//0 - alphanum start

//1 - numnum start

//2 - alphanum end

//3 - numnum end

string playermove[4] = {"-1","-1","-1","-1"};

//initialization

readInPieceDataNames(blackPieces, whitePieces, pieces, names, playernames);

initializeBoardPieces(board, blackPieces, whitePieces, piece);

updateTiles(board, blackPieces, whitePieces, playermove, piece, alphaCoordinates, whiteturn, false);

//view?

displayBoard(board, alphaCoordinates);

outputInstructions();

//controller?

while(!gameEnd(resignation, whiteturn, offerdraw, piece, board, whitePieces, blackPieces, alphaCoordinates, playernames, moves))

{

do

{

getPlayerMoveStart(playermove, whiteturn);

if (playermove[0] == "Resign")

{

resignation = true;

continue;

}

else if (playermove[0] == "OfferDraw")

{

outputName(whiteturn, playernames);

cout << "has offered a draw. Do you accept?(y/n) ";

cin >> drawresponse;

if (drawresponse == 'Y' || drawresponse == 'y')

{

offerdraw = true;

continue;

}

}

verifyPlayerMoveStart(playermove, whiteturn, board, blackPieces, whitePieces, alphaCoordinates, piece);

getPlayerMoveEnd(playermove);

verifyPlayerMoveEnd(board, blackPieces, whitePieces, playermove, piece, whiteturn, alphaCoordinates);

}

while (!resignation && !offerdraw && !verifyPlayerMove(board, blackPieces, whitePieces, playermove, piece, whiteturn, alphaCoordinates, false, false, whiteturn));

updateMoveList(moveList, moves, playermove);

updateTiles(board, blackPieces, whitePieces, playermove, piece, alphaCoordinates, whiteturn, false);

displayBoard(board, alphaCoordinates);

whiteturn = !whiteturn;

}

string\* listptr = createMoveList(moves);

printMoveList(listptr, moves, moveList, whiteturn, whitePieces, blackPieces);

outputData(whitePieces, blackPieces, piece);

delete[] listptr;

return 0;

}

bool VPMCheckAlpha(string playermove, char alphaCoordinates[])

{

bool alphafound = false;

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

{

if (playermove[0] == alphaCoordinates[i])

{

alphafound = true;

}

}

return alphafound;

}

bool VPMkingNotinCheck(tiles board[8][8], piecedata piece[], string playermove[], char alphaCoordinates[], char blackPieces[], char whitePieces[], bool whiteturn, bool hypo, bool simulation)

{

tiles copyboard[8][8];

piecedata copypiece[32];

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

{

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

{

copyboard[i][j] = copyTiles(board, i, j);

}

}

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

{

copypiece[i].conditions.ispawn = piece[i].conditions.ispawn;

copypiece[i].conditions.doubleadvance = piece[i].conditions.doubleadvance;

copypiece[i].conditions.white = piece[i].conditions.white;

copypiece[i].conditions.incheck = piece[i].conditions.incheck;

copypiece[i].conditions.captured = piece[i].conditions.captured;

copypiece[i].position[0] = piece[i].position[0];

copypiece[i].position[1] = piece[i].position[1];

copypiece[i].moves = piece[i].moves;

copypiece[i].new\_i = piece[i].new\_i;

}

updateTiles(copyboard, blackPieces, whitePieces, playermove, copypiece, alphaCoordinates, whiteturn, true);

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 pos1 = pieceSearchArrayPosition(startmoverow, startmovecol, piece);

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

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

{

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

{

if (piece[pos2].conditions.issupported)

{

return false;

}

}

}

if (whiteturn)

{

return !copyboard[copypiece[12].position[0]][copypiece[12].position[1]].underattackBlack;

}

else if (!whiteturn)

{

return !copyboard[piece[4].position[0]][copypiece[4].position[1]].underattackWhite;

}

}

bool VPMisBlocked(int i, int startmoverow, int startmovecol, int endmoverow, int endmovecol, tiles board[8][8], piecedata piece[])

{

bool blocked = false;

//bishops

if (i == 2 || i == 5 || i == 10 || i == 13)

{

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

{

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

{

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

{

blocked = true;

}

startmoverow++;

startmovecol++;

}

}

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

{

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

{

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

{

blocked = true;

}

startmoverow--;

startmovecol++;

}

}

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

{

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

{

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

{

blocked = true;

}

startmoverow++;

startmovecol--;

}

}

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

{

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

{

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

{

blocked = true;

}

startmoverow--;

startmovecol--;

}

}

}

//rooks

else if (i == 0 || i == 7 || i == 8 || i == 15)

{

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

{

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

{

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

{

blocked = true;

}

startmovecol++;

}

}

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

{

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

{

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

{

blocked = true;

}

startmovecol--;

}

}

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

{

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

{

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

{

blocked = true;

}

startmoverow++;

}

}

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

{

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

{

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

{

blocked = true;

}

startmoverow--;

}

}

}

return blocked;

}

bool VPMKingMoveSafe(tiles board[8][8], piecedata piece[], int endmoverow, int endmovecol, bool whiteturn)

{

bool safemove = true;

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

{

safemove = false;

}

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

{

safemove = false;

}

return safemove;

}

bool VPMCheckNum(string playermove)

{

bool numfound = false;

//cout << "i am comparing" << playermove[1] << " to " << static\_cast<int>(playermove[1]) << endl;

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

{

numfound = true;

}

return numfound;

}

bool VPMCheckPiece(string playermove, tiles board[8][8], char blackPieces[], char whitePieces[])

{

bool piecefound = false;

//cout << "movenum is " << movenum << endl;

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

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

for (int i = 0; i < 6 && !piecefound; i++)

{

//cout << blackPieces[i] << " " << whitePieces[i] << endl;

//cout << moverow << movecol << i << endl;

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

{

piecefound = true;

//cout << "i found a black piece of " << i << " at " << moverow << " " << movecol << endl;

}

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

{

piecefound = true;

//cout << "i found a white piece of " << i << " at " << moverow << " " << movecol << endl;

}

}

return piecefound;

}

bool VPMCheckMovement(tiles board[8][8], int startmoverow, int startmovecol, int endmoverow, int endmovecol, piecedata piece[], bool intentcapture)

{

bool valid = false;

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

if (piece[i].new\_i != -1)

{

i = piece[i].new\_i;

}

//white pawn

if (i >= 16 && i <= 23)

{

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

{

if (piece[i].moves == 0 && !board[endmoverow][endmovecol].occupied && !board[endmoverow-1][endmovecol].occupied)

{

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

valid = true;

piece[i].conditions.doubleadvance = true;

}

}

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

{

valid = true;

}

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

{

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

{

valid = true;

}

else if (board[endmoverow-1][endmovecol].black && piece[pieceSearchArrayPosition(endmoverow-1, endmovecol, piece)].conditions.doubleadvance)

{

valid = true;

}

}

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

{

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

{

valid = true;

}

else if (board[endmoverow-1][endmovecol].black && piece[pieceSearchArrayPosition(endmoverow-1, endmovecol, piece)].conditions.doubleadvance)

{

valid = true;

}

}

}

//black pawn

else if (i >=24)

{

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

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

{

if (piece[i].moves == 0 && !board[endmoverow][endmovecol].occupied && !board[endmoverow+1][endmovecol].occupied)

{

valid = true;

piece[i].conditions.doubleadvance = true;

}

}

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

{

valid = true;

}

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

{

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

{

valid = true;

}

else if (board[endmoverow+1][endmovecol].white && piece[pieceSearchArrayPosition(endmoverow+1, endmovecol, piece)].conditions.doubleadvance)

{

valid = true;

}

}

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

{

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

{

valid = true;

}

else if (board[endmoverow+1][endmovecol].white && piece[pieceSearchArrayPosition(endmoverow+1, endmovecol, piece)].conditions.doubleadvance)

{

valid = true;

}

}

}

//all knights

else if (i == 1 || i == 6 || i == 9 || i == 14)

{

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

{

valid = true;

}

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

{

valid = true;

}

}

//all bishops

else if (i == 2 || i == 5 || i == 10 || i == 13)

{

if (abs(endmoverow - startmoverow) == abs(endmovecol - startmovecol) && !VPMisBlocked(i, startmoverow, startmovecol, endmoverow, endmovecol, board, piece))

{

valid = true;

}

}

//all rooks

else if (i == 0 || i == 7 || i == 8 || i == 15)

{

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

{

if (!VPMisBlocked(i, startmoverow, startmovecol, endmoverow, endmovecol, board, piece))

{

valid = true;

}

}

}

//black king

else if (i == 4)

{

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 (piece[i].moves == 0 && !board[startmoverow][startmovecol].underattackWhite && !board[startmoverow][startmovecol+1].underattackWhite && !board[startmoverow][startmovecol+2].underattackWhite)

{

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

{

if (piece[pieceSearchArrayPosition(7,7,piece)].moves == 0)

{

valid = true;

}

}

}

}

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

{

if (piece[i].moves == 0 && !board[startmoverow][startmovecol].underattackWhite && !board[startmoverow][startmovecol-1].underattackWhite && !board[startmoverow][startmovecol-2].underattackWhite)

{

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

{

if (piece[pieceSearchArrayPosition(7,0,piece)].moves == 0)

{

valid = true;

}

}

}

}

}

//white king

else if (i == 12)

{

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 (piece[i].moves == 0 && !board[startmoverow][startmovecol].underattackBlack && !board[startmoverow][startmovecol+1].underattackBlack && !board[startmoverow][startmovecol+2].underattackBlack)

{

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

{

if (piece[pieceSearchArrayPosition(0,7,piece)].moves == 0)

{

valid = true;

}

}

}

}

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

{

if (piece[i].moves == 0 && !board[startmoverow][startmovecol].underattackBlack && !board[startmoverow][startmovecol-1].underattackBlack && !board[startmoverow][startmovecol-2].underattackBlack)

{

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

{

if (piece[pieceSearchArrayPosition(0,0,piece)].moves == 0)

{

valid = true;

}

}

}

}

}

//both queens

else if (i == 3 || i == 11)

{

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 (!VPMisBlocked(0, startmoverow, startmovecol, endmoverow, endmovecol, board, piece))

{

valid = true;

}

}

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

{

if (!VPMisBlocked(2, startmoverow, startmovecol, endmoverow, endmovecol, board, piece))

{

valid = true;

}

}

}

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

{

valid = false;

}

return valid;

}

bool VPMCheckColor(string playermove, bool whiteturn, tiles board[8][8], char blackPieces[], char whitePieces[])

{

bool colorfound = false;

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

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

//cout << movecol << moverow;

if (whiteturn)

{

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

{

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

{

colorfound = true;

}

}

}

else if (!whiteturn)

{

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

{

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

{

colorfound = true;

}

}

}

return colorfound;

}

bool VPMCheckCapture(bool whiteturn, string playermove[], tiles board[8][8], bool simulation, piecedata piece[], bool realtimewhiteturn)

{

bool capturable = true;

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';

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

{

capturable = false;

}

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

{

capturable = false;

}

return capturable;

}

bool cannotMove(piecedata piece[], int K)

{

bool cannotMove = true;

if (K == 4)

{

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

{

if (piece[K].possible\_moves[m] != -1)

{

cannotMove = false;

}

}

}

else if (K == 12)

{

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

{

if (piece[K].possible\_moves[m] != -1)

{

cannotMove = false;

}

}

}

if (K == 4)

{

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

{

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

{

if (piece[i].possible\_moves[m] != -1)

{

cannotMove = false;

}

}

}

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

{

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

{

if (piece[i].possible\_moves[m] != -1)

{

cannotMove = false;

}

}

}

}

else if (K == 12)

{

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

{

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

{

if (piece[i].possible\_moves[m] != -1)

{

cannotMove = false;

}

}

}

}

return cannotMove;

}

string convertAtlasCoordinatesToNumeric(string playermove, char alphaCoordinates[])

{

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

{

if (playermove[0] == alphaCoordinates[i])

{

playermove[0] = i+48;

}

}

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 gameEnd(bool resignation, bool whiteturn, bool offerdraw, piecedata piece[], tiles board[8][8], char whitePieces[], char blackPieces[], char alphaCoordinates[], string playernames[], int moves)

{

bool gameEnd = false;

bool endConditions[6] = {0,0,0,0,0};

endConditions[0] = isCheckmate(board, piece, blackPieces, whitePieces, alphaCoordinates, whiteturn);

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

endConditions[2] = is50Moves(moves);

endConditions[3] = isInsufficient(piece);

endConditions[4] = resignation;

endConditions[5] = offerdraw;

if (endConditions[0])

{

gameEnd = true;

cout << "Checkmate!" << endl;

outputName(!whiteturn, playernames);

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

}

else if (endConditions[1])

{

gameEnd = true;

cout << "Stalemate!" << endl;

}

else if (endConditions[2])

{

gameEnd = true;

cout << "50 move draw declared";

}

else if (endConditions[3])

{

gameEnd = true;

cout << "Insufficient material";

}

else if (endConditions[4])

{

gameEnd = true;

outputName(!whiteturn, playernames);

cout << "has resigned the game." << endl;

}

else if (endConditions[5])

{

if (offerdraw)

{

outputName(whiteturn, playernames);

cout << "has accepted the draw" << endl;

gameEnd = true;

}

else

{

gameEnd = false;

}

}

return gameEnd;

}

void findSupports(piecedata piece[], tiles board[8][8])

{

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

{

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

{

if (!piece[j].conditions.captured && !piece[i].conditions.captured)

{

if (VPMCheckMovement(board, piece[i].position[0], piece[i].position[1], piece[j].position[0], piece[j].position[1], piece, true))

{

piece[j].conditions.issupported = true;

}

}

}

}

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

{

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

{

if (!piece[j].conditions.captured && !piece[i].conditions.captured)

{

if (VPMCheckMovement(board, piece[i].position[0], piece[i].position[1], piece[j].position[0], piece[j].position[1], piece, true))

{

piece[j].conditions.issupported = true;

}

}

}

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

{

if (!piece[j].conditions.captured && !piece[i].conditions.captured)

{

if (VPMCheckMovement(board, piece[i].position[0], piece[i].position[1], piece[j].position[0], piece[j].position[1], piece, true))

{

piece[j].conditions.issupported = true;

}

}

}

}

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

{

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

{

if (!piece[j].conditions.captured && !piece[i].conditions.captured)

{

if (VPMCheckMovement(board, piece[i].position[0], piece[i].position[1], piece[j].position[0], piece[j].position[1], piece, true))

{

piece[j].conditions.issupported = true;

}

}

}

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

{

if (!piece[j].conditions.captured && !piece[i].conditions.captured)

{

if (VPMCheckMovement(board, piece[i].position[0], piece[i].position[1], piece[j].position[0], piece[j].position[1], piece, true))

{

piece[j].conditions.issupported = true;

}

}

}

}

}

void displayBoard(tiles board[8][8], char alphaCoordinates[])

{

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

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

{

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

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

{

cout << board[i][j].symbol /\*<< i << j\*/ << " | ";

}

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

}

cout << endl << " ";

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

{

cout << alphaCoordinates[i] << " ";

}

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

}

string\* createMoveList(int moves)

{

string\* ptr = new string[moves\*2];

return ptr;

}

tiles copyTiles(tiles board[8][8], int i, int j)

{

tiles copyboard;

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

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

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

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

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

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

return copyboard;

}

string convertNumericToAtlasCoordinates(int row, int col, char alphaCoordinates[])

{

string str;

str.push\_back(alphaCoordinates[col]);

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

return str;

}

bool isCheck(tiles board[8][8], piecedata piece[], int endmoverow, int endmovecol, bool white, bool hypo)

{

bool incheck = false;

if (white && !hypo)

{

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

{

if (piece[i].conditions.captured == false)

{

incheck = VPMCheckMovement(board, piece[i].position[0], piece[i].position[1], endmoverow, endmovecol, piece, true);

}

}

}

else if (!white && !hypo)

{

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

{

if (piece[i].conditions.captured == false)

{

incheck = VPMCheckMovement(board, piece[i].position[0], piece[i].position[1], endmoverow, endmovecol, piece, true);

}

}

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

{

if (piece[i].conditions.captured == false && !incheck)

{

incheck = VPMCheckMovement(board, piece[i].position[0], piece[i].position[1], endmoverow, endmovecol, piece, true);

}

}

}

return incheck;

}

bool is50Moves(int moves)

{

bool draw = false;

if (moves/2 > 50)

{

draw = true;

}

return draw;

}

void initializeBoardPieces(tiles board[8][8], char blackPieces[], char whitePieces[], piecedata piece[])

{

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

{

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

{

//space signifies empty space

board[i][j].symbol = ' ';

}

}

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

{

board[1][i].symbol = whitePieces[0];;

board[6][i].symbol = blackPieces[0];;

}

//special pieces

board[7][0].symbol = blackPieces[ROOK];

board[7][7].symbol = blackPieces[ROOK];

board[7][1].symbol = blackPieces[KNIGHT];

board[7][6].symbol = blackPieces[KNIGHT];

board[7][2].symbol = blackPieces[BISHOP];

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

board[7][4].symbol = blackPieces[KING];

board[7][3].symbol = blackPieces[4];

board[0][7].symbol = whitePieces[ROOK];

board[0][0].symbol = whitePieces[ROOK];

board[0][6].symbol = whitePieces[KNIGHT];

board[0][1].symbol = whitePieces[KNIGHT];

board[0][5].symbol = whitePieces[BISHOP];

board[0][2].symbol = whitePieces[BISHOP];

board[0][4].symbol = whitePieces[KING];

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

//white pawns

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

{

piece[i].conditions.ispawn = true;

piece[i].conditions.captured = false;

piece[i].conditions.incheck = false;

piece[i].conditions.white = true;

piece[i].position[0] = 1;

piece[i].position[1] = i-16;

piece[i].conditions.doubleadvance = false;

piece[i].moves = 0;

piece[i].new\_i = -1;

piece[i].conditions.issupported = false;

piece[i].counter = 0;

}

//cout << piece[20].conditions.position[0] << piece[20].conditions.position[1] << endl;

//black pawns

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

{

piece[i].conditions.ispawn = true;

piece[i].conditions.captured = false;

piece[i].conditions.incheck = false;

piece[i].conditions.white = false;

piece[i].position[0] = 6;

piece[i].position[1] = i-24;

piece[i].conditions.doubleadvance = false;

piece[i].moves = 0;

piece[i].new\_i = -1;

piece[i].conditions.issupported = false;

piece[i].counter = 0;

}

//black specials

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

{

piece[i].conditions.ispawn = false;

piece[i].conditions.captured = false;

piece[i].conditions.incheck = false;

piece[i].conditions.white = false;

piece[i].position[0] = 7;

piece[i].position[1] = i;

piece[i].conditions.doubleadvance = false;

piece[i].moves = 0;

piece[i].new\_i = -1;

piece[i].conditions.issupported = false;

piece[i].counter = 0;

}

//white specials

for (int i = 8; i < 16; i++)

{

piece[i].conditions.ispawn = false;

piece[i].conditions.captured = false;

piece[i].conditions.incheck = false;

piece[i].conditions.white = true;

piece[i].position[0] = 0;

piece[i].position[1] = i-8;

piece[i].conditions.doubleadvance = false;

piece[i].moves = 0;

piece[i].new\_i = -1;

piece[i].conditions.issupported = false;

piece[i].counter = 0;

}

}

void getPlayerMoveStart(string playermove[], bool whiteturn)

{

cout << "Please enter the coordinates of the piece you would like to move. ex: 'b4'";

cin >> playermove[0];

}

void getPlayerMoveEnd(string playermove[])

{

cout << "Please enter the coordinates of where you'd like your piece to go. ex 'e4'";

cin >> playermove[2];

}

void generatePossibleMoves(piecedata piece[], tiles board[8][8], char blackPieces[], char whitePieces[], char alphaCoordinates[], bool whiteturn)

{

int m = 0;

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

{

m = 0;

if (!piece[i].conditions.captured)

{

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

{

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

{

piece[i].possible\_moves[m++] = movePossible(j, k, board, piece, blackPieces, whitePieces, alphaCoordinates, i, whiteturn);

}

}

}

else if (piece[i].conditions.captured)

{

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

{

piece[i].possible\_moves[j] = -1;

}

}

}

}

void outputData(char whitePieces[], char blackPieces[], piecedata piece[])

{

fstream binary;

record\* movedata;

movedata->totalmoveswhite = 0;

movedata->totalmovesblack = 0;

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

{

movedata->totalmoveswhite += piece[i].moves;

}

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

{

movedata->totalmovesblack += piece[i].moves;

}

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

{

movedata->totalmovesblack += piece[i].moves;

}

binary.open("totalmoves.txt", ios::binary | ios::out);

binary.seekp(20L, ios::beg);

binary.write(reinterpret\_cast<char \*>(movedata), sizeof(movedata));

binary.close();

}

int movePossible(int a, int b, tiles board[8][8], piecedata piece[], char blackPieces[], char whitePieces[], char alphaCoordinates[], int c, bool whiteturn)

{

string playermove[4];

playermove[0] = convertNumericToAtlasCoordinates(piece[c].position[0], piece[c].position[1], alphaCoordinates);

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

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

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

bool result = verifyPlayerMove(board, blackPieces, whitePieces, playermove, piece, piece[c].conditions.white, alphaCoordinates, true, true, whiteturn);

if (!result)

{

return -1;

}

if (result)

{

return stoi(playermove[3]);

}

}

bool isWhite(tiles board[8][8], int row, int col, char whitePieces[])

{

bool white = false;

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

{

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

{

white = true;

}

}

return white;

}

bool isStalemate(piecedata piece[], bool whiteturn)

{

bool stalemate = false;

bool flag = false;

if (!whiteturn)

{

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

{

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

{

if (piece[i].possible\_moves[m] != -1)

{

flag = true;

}

}

}

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

{

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

{

if (piece[i].possible\_moves[m] != -1)

{

flag = true;

}

}

}

if (!flag && !piece[4].conditions.incheck)

{

stalemate = true;

}

}

else if (whiteturn)

{

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

{

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

{

if (piece[i].possible\_moves[m] != -1)

{

flag = true;

}

}

}

if (!flag && !piece[12].conditions.incheck)

{

stalemate = true;

}

}

return stalemate;

}

bool isInsufficient(piecedata piece[])

{

int whiteknights = 0;

int whitebishops = 0;

int blackknights = 0;

int blackbishops = 0;

int whiteminor;

int blackminor;

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

{

if (!piece[i].conditions.captured)

{

if (piece[i].conditions.ispawn)

{

return false;

}

else if (piece[i].new\_i != -1)

{

if (piece[i].new\_i == 1 || piece[i].new\_i == 2)

{

if (piece[i].conditions.white)

{

whiteminor++;

}

else if (!piece[i].conditions.white)

{

blackminor++;

}

}

else if (piece[i].new\_i == 4 || piece[i].new\_i == 0)

{

return false;

}

}

}

}

//if any rook exists, then false;

if (!piece[0].conditions.captured || !piece[7].conditions.captured || !piece[8].conditions.captured || !piece[15].conditions.captured)

{

return false;

}

//if any queen exists, then false;

else if (!piece[3].conditions.captured || !piece[11].conditions.captured)

{

return false;

}

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

{

if (!piece[i].conditions.captured)

{

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++;

}

}

}

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 isCheckmate(tiles board[8][8], piecedata piece[], char blackPieces[], char whitePieces[], char alphaCoordinates[], bool whiteturn)

{

bool checkmate = false;

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

int i = -1;

if (board[piece[12].position[0]][piece[12].position[1]].underattackBlack)

{

checkmateConditions[0] = true;

i=12;

}

else if (board[piece[4].position[0]][piece[4].position[1]].underattackWhite)

{

checkmateConditions[0] = true;

i=4;

}

if (i == -1)

{

return checkmate;

}

generatePossibleMoves(piece, board, blackPieces, whitePieces, alphaCoordinates, whiteturn);

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

if (checkmateConditions[1])

{

checkmate = true;

}

return checkmate;

}

void outputInstructions()

{

cout << "You can find the rules of chess on the following website:" << endl;

cout << "https://chess-teacher.com/chess-rules/" << endl;

cout << "Insufficient material conditions are taken from here:" << endl;

cout << "https://www.chess.com/article/view/how-chess-games-can-end-8-ways-explained#insufficient-material" << endl;

cout << "If at any time you want to resign, enter 'Resign' as your starting move" << endl;

cout << "If 50 moves pass, then the game is declared a draw" << endl;

cout << "If a three-fold repetition occurs, you can type 'Draw' to declare a draw as your starting move" << endl;

cout << "If you would like to offer a draw, type in 'OfferDraw' as your starting move" << endl;

cout << endl;

}

void printMoveList(string\* listptr, int moves, string moveList[], bool whiteturn, char whitePieces[], char blackPieces[])

{

fstream file;

fstream simul;

int maxprint;

string namewhite;

string nameblack;

file.open("output.txt", ios::out);

file << "Total Moves: " << moves << endl;

file << setw(10) << "White" << setw(10) << "Black" << endl;

for (int i = 0; i < moves\*2; i++)

{

\*(listptr + i) = moveList[i];

}

for (int i = 0; i < moves\*2; i++)

{

file << setw(7) << \*(listptr + i++) << " " << \*(listptr + i);

if ((i+1)%4==0)

{

file << endl;

}

}

file << endl;

file << "White Pieces" << endl;

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

{

file.put(whitePieces[i]) << " ";

}

file << endl;

file << "Black Pieces" << endl;

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

{

file.put(blackPieces[i]) << " ";

}

file.close();

simul.open("names.txt", ios::in | ios::out | ios::app);

simul >> namewhite;

simul >> nameblack;

simul << endl;

if (whiteturn)

{

simul << nameblack << " made the last move!";

}

else if (!whiteturn)

{

simul << namewhite << " made the last move!";

}

simul.close();

}

int pieceSearchArrayPosition(int row, int col, piecedata piece[])

{

int position = -1;

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

{

if(piece[i].position[0] == row && piece[i].position[1] == col)

{

//cout << "i is " << i << endl;

position = i;

}

}

// if (position == -1)

// {

// cout << "The piece search function could not find your piece at " << row << " " << col << endl;

// }

return position;

}

void pawnUpdate(piecedata piece[], char blackPieces[], char whitePieces[], tiles board[8][8], bool simulation)

{

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

{

if (piece[i].conditions.doubleadvance)

{

if (piece[i].counter++ == 1)

{

piece[i].conditions.doubleadvance = false;

}

}

}

if (!simulation)

{

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

{

if(piece[i].conditions.white && piece[i].conditions.ispawn)

{

if (piece[i].position[0] == 7)

{

pawnPromotion(piece, blackPieces, whitePieces, board, true, i);

}

}

else if (!piece[i].conditions.white && piece[i].conditions.ispawn)

{

if (piece[i].position[0] == 0)

{

pawnPromotion(piece, blackPieces, whitePieces, board, false, i);

}

}

}

}

}

void pawnPromotion(piecedata piece[], char blackPieces[], char whitePieces[], tiles board[8][8], bool isWhite, int i)

{

int pvalue;

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;

}

if(isWhite)

{

if (pvalue == 1)

{

piece[i].conditions.ispawn = false;

board[piece[i].position[0]][piece[i].position[1]].symbol = whitePieces[KNIGHT];

piece[i].new\_i = 9;

}

else if (pvalue == 2)

{

piece[i].conditions.ispawn = false;

board[piece[i].position[0]][piece[i].position[1]].symbol = whitePieces[BISHOP];

piece[i].new\_i = 10;

}

else if (pvalue == 4)

{

piece[i].conditions.ispawn = false;

board[piece[i].position[0]][piece[i].position[1]].symbol = whitePieces[QUEEN];

piece[i].new\_i = 11;

}

else if (pvalue == 5)

{

piece[i].conditions.ispawn = false;

board[piece[i].position[0]][piece[i].position[1]].symbol = whitePieces[ROOK];

piece[i].new\_i = 8;

}

}

else if(!isWhite)

{

if (pvalue == 1)

{

piece[i].conditions.ispawn = false;

board[piece[i].position[0]][piece[i].position[1]].symbol = blackPieces[KNIGHT];

piece[i].new\_i = 1;

}

else if (pvalue == 2)

{

piece[i].conditions.ispawn = false;

board[piece[i].position[0]][piece[i].position[1]].symbol = blackPieces[BISHOP];

piece[i].new\_i = 2;

}

else if (pvalue == 4)

{

piece[i].conditions.ispawn = false;

board[piece[i].position[0]][piece[i].position[1]].symbol = blackPieces[QUEEN];

piece[i].new\_i = 3;

}

else if (pvalue == 5)

{

piece[i].conditions.ispawn = false;

board[piece[i].position[0]][piece[i].position[1]].symbol = blackPieces[ROOK];

piece[i].new\_i = 0;

}

}

}

void outputName(bool whiteturn, string playernames[])

{

if (whiteturn)

{

cout << playernames[0] << " ";

}

else if (!whiteturn)

{

cout << playernames[1] << " ";

}

}

void verifyPlayerMoveStart(string playermove[4], bool whiteturn, tiles board[8][8], char blackPieces[], char whitePieces[], char alphaCoordinates[], piecedata piece[])

{

static bool checkpoints[3] = {0,0,0};

checkpoints[0] = VPMCheckAlpha(playermove[0], alphaCoordinates);

checkpoints[1] = VPMCheckNum(playermove[0]);

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;

getPlayerMoveStart(playermove, whiteturn);

verifyPlayerMoveStart(playermove, whiteturn, board, blackPieces, whitePieces, alphaCoordinates, piece);

}

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

checkpoints[2] = VPMCheckPiece(playermove[1], board, blackPieces, whitePieces);

checkpoints[3] = VPMCheckColor(playermove[1], whiteturn, board, blackPieces, whitePieces);

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

{

if (!checkpoints[0])

{

cout << "Your alphanumerical character is incorrect";

}

else if (!checkpoints[1])

{

cout << "Your numerical value is incorrect";

}

else if (!checkpoints[2])

{

cout << "There is no piece in the tile you selected";

}

else if (!checkpoints[3])

{

cout << "The piece you selected was not yours!";

}

// else if (!checkpoints[4])

// {

// cout << "You must select the king or a piece that can block the check";

// }

cout << endl;

getPlayerMoveStart(playermove, whiteturn);

verifyPlayerMoveStart(playermove, whiteturn, board, blackPieces, whitePieces, alphaCoordinates, piece);

}

}

void verifyPlayerMoveEnd(tiles board[8][8], char blackPieces[], char whitePieces[], string playermove[], piecedata piece[], bool whiteturn, char alphaCoordinates[])

{

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

checkpoints[0] = VPMCheckAlpha(playermove[2], alphaCoordinates);

checkpoints[1] = VPMCheckNum(playermove[2]);

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;

getPlayerMoveEnd(playermove);

verifyPlayerMoveEnd(board, blackPieces, whitePieces, playermove, piece, whiteturn, alphaCoordinates);

}

}

bool verifyPlayerMove(tiles board[8][8], char blackPieces[8], char whitePieces[8], string playermove[], piecedata piece[], bool whiteturn, char alphaCoordinates[], bool hypo, bool simulation, bool realtimewhiteturn)

{

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

if (!hypo)

{

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

}

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);

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

{

checkpoints[0] = VPMCheckCapture(whiteturn, playermove, board, simulation, piece, realtimewhiteturn);

}

checkpoints[1] = VPMCheckMovement(board, startmoverow, startmovecol, endmoverow, endmovecol, piece, false);

checkpoints[2] = VPMkingNotinCheck(board, piece, playermove, alphaCoordinates, blackPieces, whitePieces, whiteturn, hypo, true);

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

{

checkpoints[3] = VPMKingMoveSafe(board, piece, endmoverow, endmovecol, whiteturn);

}

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

{

checkpoints[3] = VPMKingMoveSafe(board, piece, endmoverow, endmovecol, whiteturn);

}

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

{

//cout << playermove[0] << " " << playermove[2] << endl;

if (!simulation)

{

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 is under attack!" << 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;

}

}

void updateTiles(tiles board[8][8], char blackPieces[], char whitePieces[], string playermove[], piecedata piece[], char alphaCoordinates[], bool whiteturn, bool simulation)

{

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

{

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 pos1 = pieceSearchArrayPosition(endmoverow, endmovecol, piece);

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

//generic captures

if (pos1 != -1)

{

piece[pos1].position[0] = -1;

piece[pos1].position[1] = -1;

piece[pos1].conditions.captured = true;

}

//enpassants

if (pos1 == -1 && piece[pos2].conditions.ispawn && endmoverow - startmoverow == 1 && endmovecol - startmovecol == 1)

{

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

piece[pos3].position[0] = -1;

piece[pos3].position[1] = -1;

piece[pos3].conditions.captured = true;

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

}

else if (pos1 == -1 && piece[pos2].conditions.ispawn && endmoverow - startmoverow == 1 && endmovecol - startmovecol == -1)

{

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

piece[pos3].position[0] = -1;

piece[pos3].position[1] = -1;

piece[pos3].conditions.captured = true;

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

}

else if (pos1 == -1 && piece[pos2].conditions.ispawn && endmoverow - startmoverow == -1 && endmovecol - startmovecol == 1)

{

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

piece[pos3].position[0] = -1;

piece[pos3].position[1] = -1;

piece[pos3].conditions.captured = true;

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

}

else if (pos1 == -1 && piece[pos2].conditions.ispawn && endmoverow - startmoverow == -1 && endmovecol - startmovecol == -1)

{

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

piece[pos3].position[0] = -1;

piece[pos3].position[1] = -1;

piece[pos3].conditions.captured = true;

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

}

piece[pos2].position[0] = endmoverow;

piece[pos2].position[1] = endmovecol;

piece[pos2].moves++;

//castling

if (pos2 == 12 && endmovecol == 6)

{

piece[15].position[1] = 5;

piece[15].moves++;

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

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

}

else if (pos2 == 12 && endmovecol == 2)

{

piece[8].position[1] = 3;

piece[8].moves++;

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

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

}

else if (pos2 == 4 && endmovecol == 2)

{

piece[0].position[1] = 3;

piece[0].moves++;

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

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

}

else if (pos2 == 4 && endmovecol == 6)

{

piece[7].position[1] = 5;

piece[7].moves++;

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

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

}

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

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

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

}

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

{

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

{

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

{

board[i][j].occupied = true;

board[i][j].white = isWhite(board, i, j, whitePieces);

board[i][j].black = !isWhite(board, i, j, whitePieces);

}

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

{

board[i][j].occupied = false;

board[i][j].white = false;

board[i][j].black = false;

}

}

}

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

{

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

{

board[i][j].underattackWhite = isCheck(board, piece, i, j, true , false);

board[i][j].underattackBlack = isCheck(board, piece, i, j, false, false);

}

}

findSupports(piece, board);

}

void updateMoveList(string moveList[], int &moves, string playermove[])

{

static int i = 0;

moveList[i++] = playermove[0];

moveList[i++] = playermove[2];

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

{

moveList[i-1] = " ";

}

moves++;

}

void readInPieceDataNames(char blackPieces[], char whitePieces[], ifstream &pieces, ifstream &names, string playernames[])

{

pieces.open("pieceData.txt");

set<char> s;

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

{

pieces >> blackPieces[i];

s.insert(blackPieces[i]);

}

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

{

pieces >> whitePieces[i];

s.insert(whitePieces[i]);

}

pieces.close();

names.open("names.txt");

names >> playernames[0];

names >> playernames[1];

names.close();

if (s.size() != 12)

{

cout << "You did not have 12 unique piece characters" << endl;

cout << "Therefore, the default pieces are going to be:" << endl;

whitePieces[PAWN] = 'P';

whitePieces[KNIGHT] = 'N';

whitePieces[BISHOP] = 'B';

whitePieces[KING] = 'K';

whitePieces[QUEEN] = 'Q';

whitePieces[ROOK] = 'R';

blackPieces[PAWN] = '1';

blackPieces[KNIGHT] = '2';

blackPieces[BISHOP] = '3';

blackPieces[KING] = '4';

blackPieces[QUEEN] = '5';

blackPieces[ROOK] = '6';

cout << "WHITE PAWN: " << whitePieces[PAWN] << endl;

cout << "WHITE KNIGHT: " << whitePieces[KNIGHT] << endl;

cout << "WHITE BISHOP: " << whitePieces[BISHOP] << endl;

cout << "WHITE KING: " << whitePieces[KING] << endl;

cout << "WHITE QUEEN: " << whitePieces[QUEEN] << endl;

cout << "WHITE ROOK: " << whitePieces[ROOK] << endl;

cout << "BLACK PAWN: " << blackPieces[PAWN] << endl;

cout << "BLACK KNIGHT: " << blackPieces[KNIGHT] << endl;

cout << "BLACK BISHOP: " << blackPieces[BISHOP] << endl;

cout << "BLACK KING: " << blackPieces[KING] << endl;

cout << "BLACK QUEEN: " << blackPieces[QUEEN] << endl;

cout << "BLACK ROOK: " << blackPieces[ROOK] << endl;

}

}