Project 2

<Chess Game>

CIS-17C 43484

Name: Koksal, Attila

Date: 6/5/2022

Introduction

This project revolves around the popular competitive game, Chess. I am doing this game because it was one of the many games I enjoyed playing since when I was a kid. In addition, with this digital version of the game, you are no longer required to purchase a physical copy.

Summary

The program is 1052 lines of code. This project meets the criteria of the second project because of how it utilizes various concepts described in this class such as recursion sorts, maps, iterators, functions, classes, inheritance, operator overloading, polymorphism, templates, structures, pointers, arrays, nesting, enumeration, etc. 82 variables were utilized in this project. 178 constructs were utilized in this project. It was challenging for me, especially when it comes to checking and verification of check and check mate during gameplay. It took 45 hours, including research, review, debugging, and coding, in total to complete this project.

Description

With the goal of creating a playable chess game, there are various sections and challenges to be completed. These steps include initialize board to chess game's initial position, setting rules for how each piece can move (for example, a pawn can move two squares for its first move if it's not blocked and a knight move in an L shape), allowing special moves such as castling, restricting moves that can cause a player to be in check (for example, moving a king to a dangerous square), restricting all irrelevant moves when under check, and identifying and processing check mate when a move is made. On the other hand, the user can input M 1 to M 6 to view the menu options, which include displaying game creator information, displaying the player information, displaying the current winning player's lead status information, displaying the current number of turns played, displaying the board itself, and displaying the castle instructions. After each player moves and the board displays, the most significant part would be to check for any victory that can be caused by this most current player move. This includes checking if opponent is under check by any of current player's pieces. If so, check if opponent have any solution of escaping by evaluating all possible moves that can be made by the opponent. If no move can be made to escape from the check, a check mate is detected, and the game is over.

Sample Input/Output

Game start by asking for users' information:

```
Welcome to the amazing Chess game! You're in for a treat!
The chess pieces all capitalized are the WHITE pieces and the lowercase chess pieces are the BLACK pieces.
Enter a player display name for the WHITE piece: John

Enter a player display name for the BLACK piece: Michael
```

The board is displayed, user is asked to input which piece to move and where to move to:

```
-------------------------------------------------------
8 |  rok |  kgh |  bsh |  qun |  kng |  bsh |  kgh |  rok
-------------------------------------------------------
7 |  pwn |  pwn |  pwn |  pwn |  pwn |  pwn |  pwn |  pwn
-------------------------------------------------------
6 | ____ | ____ | ____ | ____ | ____ | ____ | ____ | ____
-------------------------------------------------------
5 | ____ | ____ | ____ | ____ | ____ | ____ | ____ | ____
-------------------------------------------------------
4 | ____ | ____ | ____ | ____ | ____ | ____ | ____ | ____
-------------------------------------------------------
3 | ____ | ____ | ____ | ____ | ____ | ____ | ____ | ____
-------------------------------------------------------
2 |  PWN |  PWN |  PWN |  PWN |  PWN |  PWN |  PWN |  PWN
-------------------------------------------------------
1 |  ROK |  KGH |  BSH |  QUN |  KNG |  BSH |  KGH |  ROK

       A      B      C      D      E      F      G      H


Player WHITE, enter which piece to move (enter row and column with space in between i.e (E 2))[Or enter 'M 0' for menu display]e 2
Player WHITE, enter where to move to (enter row and column with space in between i.e (E 3))e 4
```

During the game, the user can enter “M 0” to view all available menu options:

```
Player WHITE, enter which piece to move (enter row and column with space in between i.e (E 2))[Or enter 'M 0' for menu display]M 0
Welcome to this game. For information, please select one of the following options below:
'M 1': Display Game Creator Information
'M 2': Display Players
'M 3': Display Current Winning Player Lead Status
'M 4': Display Current Number of Turns
'M 5': Display Board
'M 6': Display Castle Instruction
```

When a check occurs, the message “CHECK!” is displayed:

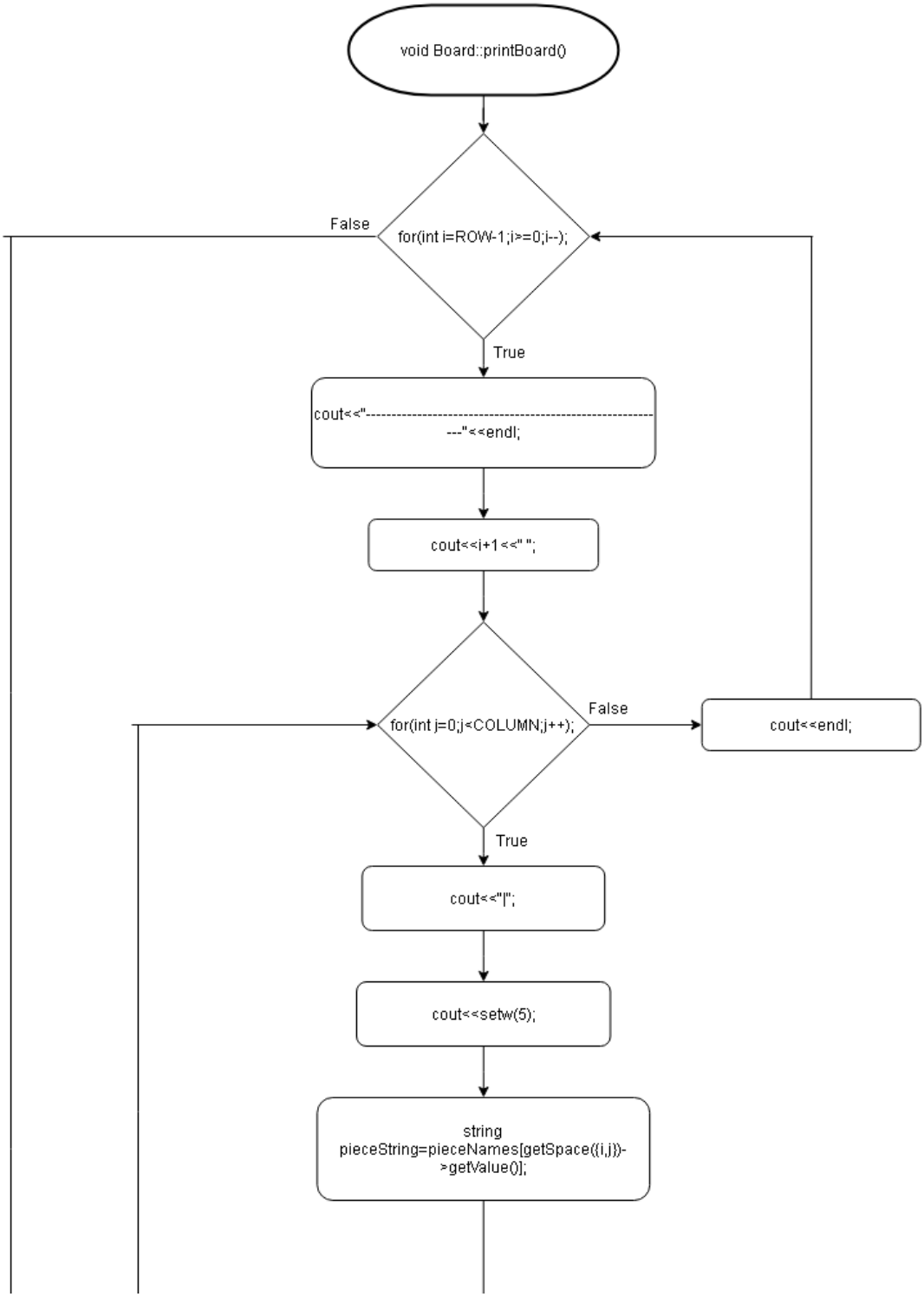
When a check mate occurs, a winner is announced, and the game is ended:

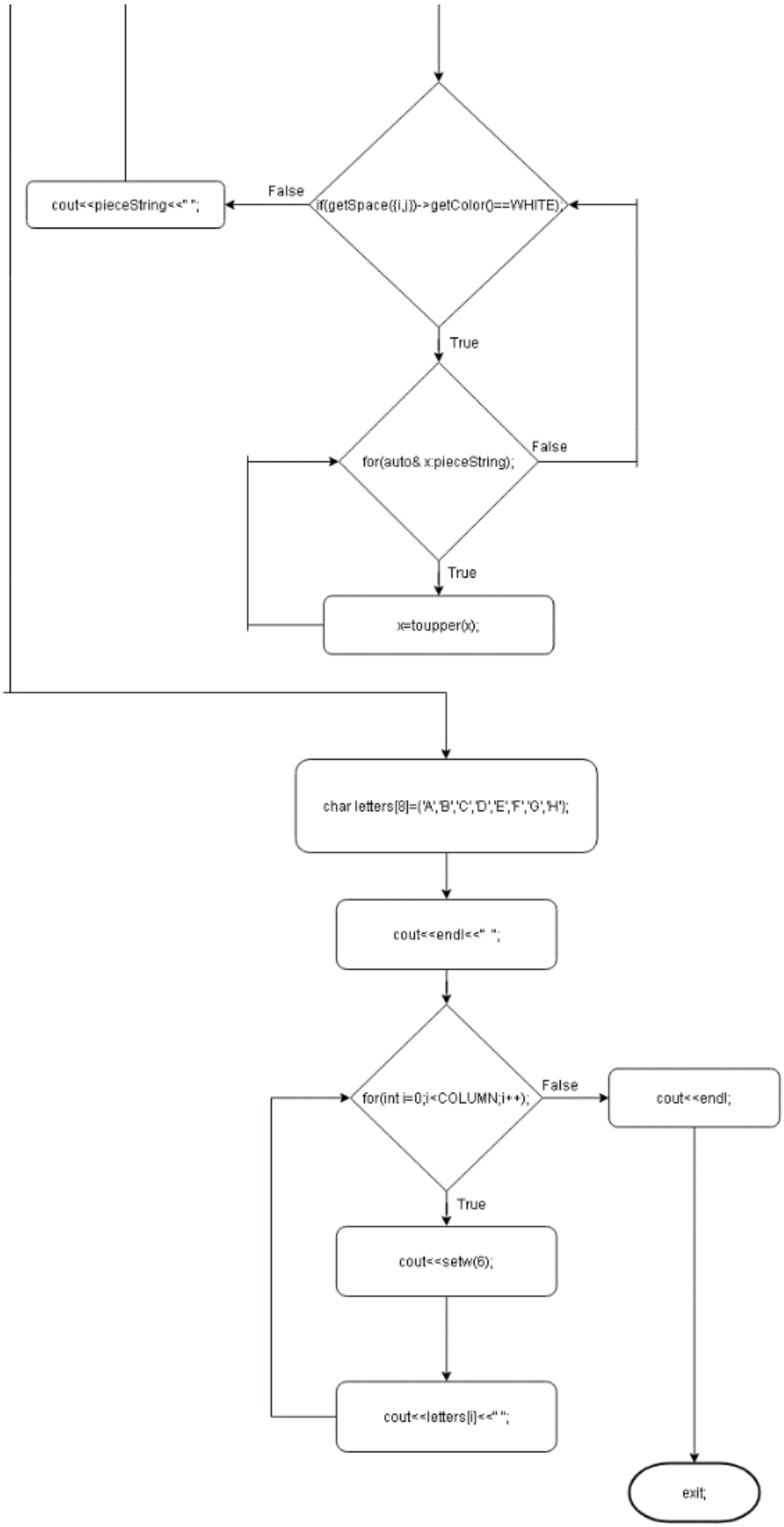
```
Player WHITE, enter which piece to move (enter row and column with space in between i.e (E 2))[Or enter 'M 0' for menu display]f 3
Player WHITE, enter where to move to (enter row and column with space in between i.e (E 3))f 7
-------------------------------------------------------
8 |  rok |  kgh |  bsh |  qun |  kng |  bsh |  kgh |  rok
-------------------------------------------------------
7 | ____ |  pwn |  pwn | ____ | ____ |  QUN |  pwn |  pwn
-------------------------------------------------------
6 |  pwn | ____ | ____ |  pwn | ____ | ____ | ____ | ____
-------------------------------------------------------
5 | ____ | ____ | ____ | ____ |  pwn | ____ | ____ | ____
-------------------------------------------------------
4 | ____ | ____ |  BSH | ____ |  PWN | ____ | ____ | ____
-------------------------------------------------------
3 | ____ | ____ | ____ | ____ | ____ | ____ | ____ | ____
-------------------------------------------------------
2 |  PWN |  PWN |  PWN |  PWN | ____ |  PWN |  PWN |  PWN
-------------------------------------------------------
1 |  ROK |  KGH |  BSH | ____ |  KNG | ____ |  KGH |  ROK

       A      B      C      D      E      F      G      H
CHECK!
Congratulations! The winner is John, who is playing the color WHITE.
```

Flowchart

Flowchart of printBoard() function




if(getSpace({i,j})->getColor()==WHITE);
False
cout<<pieceString<<" ";
True
for(auto& x:pieceString);
False
True
x=toupper(x);
char letters[8]={'A','B','C','D','E','F','G','H'};
cout<<endl<<" ";
for(int i=0;i<COLUMN;i++);
False
cout<<endl;
True
cout<<setw(6);
cout<<letters[i]<<" ";
exit;

Pseudocode

Describe the structure of Chess

Present rules of chess to players

Initialize the board

Display the initial board

While game is not over:

Player1 Move:

If currently under check:

If no available responding move:

Game over

Get user input for coordinate to move from and to move to

If invalid input (such as trying to move opponent piece):

Show error message and restart user input

Get all locations where this piece can move to

If user's 'move to' square is in the list of where this piece can move to:

Perform move

Else:

Show error message and restart user input

If current move causes user to be under check:

Retract move, show error message, and restart user input

Check if current move causes opponent to be under check

Player2 Move:

Repeat all steps above

Variables

const int ROW=8;

- The name of the variable is called ROW.
- Its type is a constant integer.
- It represents the number of rows in the Chess board.
- It is located on line 22.

const int COLUMN=8;

- The name of the variable is called COLUMN.
- Its type is a constant integer.
- It represents the number of columns in the Chess board.
- It is located on line 24.

const string pieceNames[7]={"pwn","rok","bsh","kgh","qun","kng","____"};

- The name of the variable is called pieceNames.
- Its type is a one-dimensional array that holds 7 strings.
- It represents the display names for different pieces for board use.
- It is located on line 26.

Piece value;

- The name of the variable is called value.
- Its type is a Piece, which is an enumerator in the program located on line 28.
- It represents the value of a space in the Space class which is on line 47.
- It is located on line 49.

Color color;

- The name of the variable is called color.
- Its type is a Color, which is an enumerator in the program located on line 40.
- It represents the color of a space in the Space class which is on line 47.
- It is located on line 50.

int x;

- The name of the variable is called x.
- Its type is an integer, which is part of the Coordinate struct located on line 65.
- It represents a x/row coordinate on the Chess board.
- It is located on line 68.

int y;

- The name of the variable is called y.
- Its type is an integer, which is part of the Coordinate struct located on line 65.
- It represents a y/column coordinate on the Chess board.
- It is located on line 70.

string player1="";

- The name of the variable is called player1.
- Its type is a string, which is part of the Game class located on line 77.
- It represents player1’s name.
- It is located on line 81.

string player2="";

- The name of the variable is called player2.
- Its type is a string, which is part of the Game class located on line 77.
- It represents player1's name.
- It is located on line 83.

string winnerName="";

- The name of the variable is called winnerName.
- Its type is a string, which is part of the announceWinner() function on line 87.
- It represents the winner's name.
- It is located on line 89.

Space board[ROW][COLUMN];

- The name of the variable is called board.
- Its type is a Space, which is a class in the program located on line 47 and it's part of the Board class (line 167), which inherits from the Game class.
- It represents the Chess board to be played on.
- It is located on line 170.

Color turn=WHITE;

- The name of the variable is called turn.
- Its type is a Color, which is an enum in the program located on line 40 and it's part of the Board class (line 167), which inherits from the Game class.
- It represents the color's turn that is in effect.
- It is located on line 172.

bool check=false;

- The name of the variable is called check.
- Its type is a Boolean and it's part of the Board class (line 167), which inherits from the Game class.
- It checks if a check has occurred in the Chess game.
- It is located on line 174.

int numberOfTurns=0;

- The name of the variable is called numberOfTurns.
- Its type is an integer and it's part of the Board class (line 167), which inherits from the Game class.
- It represents the number of turns of the total game played so far.
- It is located on line 176.

bool castleAble[2][2]={{true,true},{true,true}};

- The name of the variable is called castleAble.
- Its type is a Boolean 2-dimensional array and it's part of the Board class (line 167), which inherits from the Game class.

- It tracks if it's castleable for all four locations during the current game.
- It is located on line 178.

int winner=-1;

- The name of the variable is called winner.
- Its type is an integer and it’s part of the Board class (line 167), which inherits from the Game class.
- It tracks the winner during the current game.
- It is located on line 181.

string winnerName="";

- The name of the variable is called winnerName.
- Its type is a string, which is part of the announceWinner() function on line 224.
- It represents the winner’s name.
- It is located on line 226.

string player1=getPlayerName(0);

- The name of the variable is called player1.
- Its type is a string, which is part of the announceWinner() function on line 224.
- It represents the player1’s name derived from the input given.
- It is located on line 227.

string player2=getPlayerName(1);

- The name of the variable is called player2.
- Its type is a string, which is part of the announceWinner() function on line 224.
- It represents the player2’s name derived from the input given.
- It is located on line 228.

string color="";

- The name of the variable is called color.
- Its type is a string, which is part of the announceWinner() function on line 224.
- It stores the winning player’s color at the end of the game.
- It is located on line 229.

string player1=getPlayerName(0);

- The name of the variable is called player1.
- Its type is a string, which is part of the menu() function on line 246.
- It represents the player1’s name derived from the input given.
- It is located on line 248.

string player2=getPlayerName(1);

- The name of the variable is called player2.
- Its type is a string, which is part of the menu() function on line 246.

- It represents the player2’s name derived from the input given.
- It is located on line 249.

int colANDrow[]={8,5,3,6,2,7,1,4};

- The name of the variable is called colANDrow.
- Its type is an integer array which takes an array of integers.
- It labels how many rows and columns there are in the game of chess.
- It is located on line 290.

int size=sizeof(colANDrow)/sizeof(colANDrow[0]);

- The name of the variable is called size.
- Its type is an integer which takes the size of the colANDrow array.
- It produces a size value to perform bubble sort executions.
- It is located on line 291.

map<string,string> pieceMove;

- The name of the variable is called pieceMove.
- Its type is a map which takes two strings as input.
- It maps each chess piece to what moves it can do in the game of chess.
- It is located on line 300.

map<string,string>::iterator itr;

- The name of the variable is called itr.
- Its type is an iterator linked to a map which takes two strings as input.
- It helps in iterating over the map to print its elements to show tips for chess.
- It is located on line 309.

Board board;

- The name of the variable is called board.
- Its type is a Board, which is a class located on line 167.
- It represents a board created from the Board class that will be used for the Chess game.
- It is located on line 319.

string player1=“”;

- The name of the variable is called player1.
- Its type is a string, which is part of the initializeBoard() function on line 316 from the Board class.
- It represents player1’s name.
- It is located on line 348.

string player2=“”;

- The name of the variable is called player2.

- Its type is a string, which is part of the initializeBoard() function on line 316 from the Board class.
- It represents player2’s name.
- It is located on line 350.

string pieceString=pieceNames[getSpace({i,j})->getValue()];

- The name of the variable is called pieceString.
- Its type is a string, which is part of the printBoard() function on line 369 from the Board class.
- It stores each chess piece’s name derived from getting its value using getSpace(), which is a function on line 184.
- It is located on line 407.

char letters[8]={'A','B','C','D','E','F','G','H'};

- The name of the variable is called letters.
- Its type is a character array holding 8 elements, which is part of the printBoard() function on line 369 from the Board class.
- It represents the row names that will be printed on the Chess board for better game display.
- It is located on line 419.

bool foundMove=false;

- The name of the variable is called foundMove.
- Its type is a Boolean, which is part of the movePiece() function on line 433 from the Board class.
- It represents checks to see if a move done by a player on the board results in check mate.
- It is located on line 467.

Coordinate allMovable[64];

- The name of the variable is called allMovable.
- Its type is a Coordinate array, which is part of the movePiece() function on line 433 from the Board class.
- It creates a coordinate array that holds all the movable positions on the board which is 64 (8*8).
- It is located on line 469.

int size=0;

- The name of the variable is called size.
- Its type is an integer, which is part of the movePiece() function on line 433 from the Board class.
- It creates a size and initializes it to zero.
- It is located on line 471.

bool isCastleAttempt=false;

- The name of the variable is called isCastleAttempt.
- Its type is a Boolean, which is part of the movePiece() function on line 433 from the Board class.
- It stores info on whether user is attempting to castle.
- It is located on line 495.

bool valid=false;

- The name of the variable is called valid.
- Its type is a Boolean, which is part of the movePiece() function on line 433 from the Board class.
- It holds true if user input is acceptable and should not input again.
- It is located on line 497.

char letter;

- The name of the variable is called letter.
- Its type is a character, which is part of the movePiece() function on line 433 from the Board class.
- It stores user input for column.
- It is located on line 499.

Space* startSpace=new Space();

- The name of the variable is called startSpace.
- Its type is a Space, which is part of the movePiece() function on line 433 from the Board class.
- It represents the space element where user moved a piece from.
- It is located on line 501.

Space* endSpace=new Space();

- The name of the variable is called endSpace.
- Its type is a Space, which is part of the movePiece() function on line 433 from the Board class.
- It represents the space element where user moved a piece to end.
- It is located on line 503.

Coordinate start;

- The name of the variable is called start.
- Its type is a Coordinate, which is part of the movePiece() function on line 433 from the Board class.
- It holds a start coordinate.
- It is located on line 505.

Coordinate end;

- The name of the variable is called end.

- Its type is a Coordinate, which is part of the movePiece() function on line 433 from the Board class.
- It holds an end coordinate.
- It is located on line 507.

Piece startPiece;

- The name of the variable is called startPiece.
- Its type is a Piece, which is part of the movePiece() function on line 433 from the Board class.
- It holds a start piece.
- It is located on line 509.

Color enemyColor;

- The name of the variable is called enemyColor.
- Its type is a Color, which is part of the movePiece() function on line 433 from the Board class.
- It stores the opposing color.
- It is located on line 511.

string colorString="";

- The name of the variable is called colorString.
- Its type is a string, which is part of the movePiece() function on line 433 from the Board class.
- It stores the color.
- It is located on line 521.

Coordinate ableToMove[64];

- The name of the variable is called ableToMove.
- Its type is a Coordinate array, which is part of the movePiece() function on line 433 from the Board class.
- It loads in all movable spaces of a piece.
- It is located on line 560.

int size=0;

- The name of the variable is called size.
- Its type is an integer, which is part of the movePiece() function on line 433 from the Board class.
- It saves target location color info in case a reset is needed.
- It is located on line 561.

Piece replacedPiece;

- The name of the variable is called replacedPiece.

- Its type is a Piece, which is part of the movePiece() function on line 433 from the Board class.
- It saves the target location piece info in case a reset is needed.
- It is located on line 595.

Color replacedColor;

- The name of the variable is called replacecColor.
- Its type is a Color, which is part of the movePiece() function on line 433 from the Board class.
- It creates a size and initializes it to zero.
- It is located on line 596.

bool found=false;

- The name of the variable is called found.
- Its type is a Boolean, which is part of the movePiece() function on line 433 from the Board class.
- It tracks if the location user wants to move to is a location that user can move to.
- It is located on line 598.

Coordinate currentCoord=startCoord;

- The name of the variable is called startCoord.
- Its type is a Coordinate, which is part of the searchAndAdd() function on line 675 from the Board class.
- It acts as a "pointer" that goes through the board.
- It is located on line 707.

Color playerColor=getSpace(startCoord)->getColor();

- The name of the variable is called playerColor.
- Its type is a Color, which is part of the searchAndAdd() function on line 675 from the Board class.
- It gets the color of the piece being checked.
- It is located on line 709.

Coordinate currentCoord={i,j};

- The name of the variable is called currentCoord.
- Its type is a Coordinate, which is part of the findAttackSquares() function on line 707 from the Board class.
- It acts as the current viewing coordinate.
- It is located on line 740.

Piece currentPiece=getSpace(currentCoord)->getValue();

- The name of the variable is called currentPiece.

- Its type is a Piece, which is part of the findAttackSquares() function on line 707 from the Board class.
- It acts as the current viewing piece type.
- It is located on line 741.

Space* currentSpace=getSpace(currentCoord);

- The name of the variable is called currentSpace.
- Its type is a Space, which is part of the findAttackSquares() function on line 707 from the Board class.
- It acts as the current viewing space.
- It is located on line 742.

int offset;

- The name of the variable is called offset.
- Its type is an integer, which is part of the findAttackSquares() function on line 707 from the Board class.
- It acts as a placeholder for either WHITE or BLACK.
- It is located on line 749.

int availablePos[][2]={{1,2},{2,1},{-2,1},{-2,-1},{-1,2},{2,-1},{-1,-2},{1,-2}};

- The name of the variable is called availablePos.
- Its type is an integer 2-dimensional array, which is part of the findAttackSquares() function on line 707 from the Board class.
- It holds the positional offsets for the knight.
- It is located on line 780.

Coordinate
usedToCheck={currentCoord.x+availablePos[i][0],currentCoord.y+availablePos[i][1]};

- The name of the variable is called usedToCheck.
- Its type is a Coordinate, which is part of the findAttackSquares() function on line 707 from the Board class.
- It holds the space that is being checked.
- It is located on line 782.

int availablePos[][2]={{1,1},{1,0},{-1,0},{-1,-1},{1,-1},{-1,1},{0,1},{0,-1}};

- The name of the variable is called availablePos.
- Its type is an integer 2-dimensional array, which is part of the findAttackSquares() function on line 707 from the Board class.
- It holds the positional offsets for king to move (i.e. 1,1 means one row up and one column right).
- It is located on line 802.

Color enemyColor;

- The name of the variable is called enemyColor.
- Its type is a Color, which is part of the isUnderCheck() function on line 793 from the Board class.
- It stores the enemy color.
- It is located on line 824.

Coordinate myKing;

- The name of the variable is called myKing.
- Its type is a Coordinate, which is part of the isUnderCheck() function on line 793 from the Board class.
- It acts as the coordinate to hold location of the king.
- It is located on line 832.

Coordinate attackMoves[64];

- The name of the variable is called attackMoves.
- Its type is a Coordinate 1-dimensional array, which is part of the isUnderCheck() function on line 793 from the Board class.
- It holds all spaces that is being attacked.
- It is located on line 842.

int attackMoveSize=0;

- The name of the variable is called attackMoveSize.
- Its type is an integer, which is part of the isUnderCheck() function on line 793 from the Board class.
- It stores the size of the attackMoves array.
- It is located on line 843.

Coordinate ableToMove[64];

- The name of the variable is called attackMoves.
- Its type is a Coordinate 1-dimensional array, which is part of the getMovableLocations() function on line 844 from the Board class.
- It records all spaces the piece can move to.
- It is located on line 876.

int size=0;

- The name of the variable is called size.
- Its type is an integer, which is part of the getMovableLocations() function on line 844 from the Board class.
- It stores the size of the ableToMove array.
- It is located on line 878.

Piece startPiece=getSpace(start)->getValue();

- The name of the variable is called startPiece.

- Its type is a Piece, which is part of the getMovableLocations() function on line 844 from the Board class.
- It stores the piece that is being moved.
- It is located on line 880.

Space* startSpace=getSpace(start);

- The name of the variable is called startSpace.
- Its type is a Space, which is part of the getMovableLocations() function on line 844 from the Board class.
- It stores a space where a piece is moved from.
- It is located on line 882.

int offset;

- The name of the variable is called offset.
- Its type is an integer, which is part of the getMovableLocations() function on line 844 from the Board class.
- It acts as the row off set for different color pawn.
- It is located on line 885.

Coordinate attackAble[64];

- The name of the variable is called attackAble.
- Its type is a Coordinate 1-dimensional array, which is part of the getMovableLocations() function on line 844 from the Board class.
- It stores all squares that enemy are attacking.
- It is located on line 975.

int attackSize=0;

- The name of the variable is called attackSize.
- Its type is an integer, which is part of the getMovableLocations() function on line 844 from the Board class.
- It stores the size of the attackAble array.
- It is located on line 977.

Color enemyColor;

- The name of the variable is called enemyColor.
- Its type is a Color, which is part of the getMovableLocations() function on line 844 from the Board class.
- It holds the enemy color.
- It is located on line 979.

int scoreValues[]={1,5,3,3,9,0};

- The name of the variable is called scoreValues.

- Its type is an integer array, which is part of the displayWinningStatus() function on line 980 from the Board class.
- It scores a value for each piece, ordered by its index number in enum.
- It is located on line 1011.

int playerScores[2]={0,0};

- The name of the variable is called playerScores.
- Its type is an integer array, which is part of the displayWinningStatus() function on line 980 from the Board class.
- It records score for player one and two.
- It is located on line 1013.

string playerNames[2]={getPlayerName(0),getPlayerName(1)};

- The name of the variable is called playerNames.
- Its type is a string array, which is part of the displayWinningStatus() function on line 980 from the Board class.
- It holds name of players, can be accessed by player index, 0 for player1 and 1 for player2.
- It is located on line 1015.

int winningPlayerNumber=maxItem(playerScores[0],playerScores[1]);

- The name of the variable is called winningPlayerNumber.
- Its type is an integer, which is part of the displayWinningStatus() function on line 980 from the Board class.
- It stores 0 if player1 is winning, 1 if player2 is winning, -1 if tie.
- It is located on line 1033.

int scoreDifference=abs(playerScores[0]-playerScores[1]);

- The name of the variable is called scoreDifference.
- Its type is an integer, which is part of the displayWinningStatus() function on line 980 from the Board class.
- It scores the score difference between players.
- It is located on line 1039.

Concepts

I used enum Piece to represent all possible pieces of a square on the board can be. It's located on line 28. Each square on the board is represented using a Space class (starting on line 47) with class methods such as get the Piece at this square, get the Color (enum created on line 40) of this piece, and set the value of this piece. In addition, a Board class (starts on line 164), which inherits from a Game class (starts on line 77), is created to store an 8 by 8 array of Spaces, along with useful methods, including polymorphism (starts on line 221). To better refer to the coordinates, a Coordinate struct (on line 65) was created, which also utilizes an operator overload method "operator==" (on line 71) to compare if two coordinates are the same. Moreover, the user can also use a menu function (on line 143) to get useful information such as

which player is winning currently, which uses a template function (on line 127) to compare the scores of the two players. I used a recursion sort known as bubble sort to show how many rows and columns are in the game of chess on the board of 64 spaces (on line 293). I used a map and iterator on lines 300 and 309 to be able to show before the game even starts, on what the rules of chess are and what movements each piece can make in case the players involved are playing chess for the first time.

Program

// Chess

```
/**
 * @file:   main.cpp
 * @author: Attila Koksal
 * @date: Created on May 28, 2022, 4:45 PM
 * Purpose:  CPP Template
 * Chess Version 2
 */

///System Libraries
///I/O Library
#include <iostream>
///Iomanip Library
#include<iomanip>
///ValArray Library
#include <valarray>
///CMath Library
#include <cmath>
#include <iterator>
#include <map>
using namespace std;
///number of rows in the chess board
const int ROW=8;
///number of columns in the chess board
const int COLUMN=8;
///display name for different pieces for board use
const string pieceNames[7]={"pwn","rok","bsh","kgh","qun","kng","____"};

///Piece represents all possible values of a spot on a board can be, in traditional board, its spot can be either red or
yellow, or empty. Here we use X and O to represent yellow and empty
enum Piece{
    Pawn=0,
    Rook=1,
    Bishop=2,
    Knight=3,
    Queen=4,
    King=5,
    Empty=6
};

///@class
///@brief the player that on a space/square (can be none if empty)
enum Color{
    WHITE,
```

```
    BLACK,
    NONE
};

///a space on the board
class Space{
    Piece value;
    Color color;

public:
    ///@brief get which player is this space on the board
    ///@return the color user requested
    Color getColor();

    ///@brief get which piece is this space on the board
    ///@return the piece at that location
    Piece getValue();

    ///@brief set what piece is on this space
    void setValue(Piece,Color);

};

struct Coordinate{
    ///a x/row coordinate
    int x;
    ///a y/column coordinate
    int y;
    ///compares to coordinates
    bool operator==(const Coordinate& other)const{
        return (x==other.x&&y==other.y);
    }
};

///@class a super class that be inherited on any game including the previews project one if desired
class Game{
private:
    ///player1's name
    string player1="";
    ///player2's name
    string player2="";
public:
    ///declare the winner of the game
    ///@brief display who the winner is
    ///@param winner 0 is player 1 won, 1 if player 2 won
    void announceWinner(int winner){
        string winnerName="";
        if(winner==0){
            ///set winner name
            winnerName=player1;
        }
        else{
            ///set winner name
            winnerName=player2;
        }
        ///declare winner
```

```
    cout<<"Congratulations, the winner is "<<winnerName<<endl;
}

///@brief set player name
///@param playerNumber 0 for player 1, 1 for player2
///@param name the name to set to
void setPlayerName(int playerNumber,string name){
    if(playerNumber==0){
        player1=name;
    }
    else if(playerNumber==1){
        player2=name;
    }
}

///@brief get player name
///@param playerNumber 0 for player 1, 1 for player 2
///@return name of the string
string getPlayerName(int playerNumber){
    if(playerNumber==0){
        return player1;
    }
    else if(playerNumber==1){
        return player2;
    }
    else{
        return "Player Name";
    }
}
///@brief find out which item is larger
///@tparam T is template param
///@returns 0 if first parameter is larger, 1 if second parameter is larger, -1 if tie
template<typename T>
int maxItem(T a,T b){
    if(a>b){
        return 0;
    }
    else if(b>a){
        return 1;
    }
    else{
        return -1;
    }
}

///@brief menu for users to interact
///@param input is number for which menu item to be called
///this function can be overwritten by subclasses if desired
void menu(int input){
    if(input==0){
        cout<<"Welcome to this game. For information, please select one of the following options below: "<<endl;
        cout<<"'M 1': Display Game Creator Information"<<endl;
        cout<<"'M 2': Display Players"<<endl;
    }
    else if(input==1){
        cout<<"The game was developed by Attila Koksal for the CSC-17A final project."<<endl;
```

```
        }
        else if(input==2){
            cout<<"The players of this game are: "<<endl;
            cout<<"Player 1: "<<player1<<endl;
            cout<<"Player 2: "<<player2<<endl;
        }
        else{
            cout<<"Sorry, incorrect input."<<endl;
        }
    }
};

///@class this class simulates the game board using multiple functions
class Board:public Game{
    ///the board
    Space board[ROW][COLUMN];
    ///The color's turn that is in effect
    Color turn=WHITE;
    ///Checks if check has occurred
    bool check=false;
    ///Number of turns of total game played so far
    int numberOfTurns=0;
    ///Tracks if it's castleable for all four locations
    bool castleAble[2][2]={{true,true},{true,true}};
public:
    ///Tracks the winner
    int winner=-1;
    //initializes the game board
    void initializeBoard();
    //returns a specific space on the board
    Space* getSpace(Coordinate);
    //moves a piece according to the player's decision
    void movePiece(Color);
    //prints the game board
    void printBoard();
    //sets a piece with a designated coordinate, piece type, and color on the game board
    void setPiece(Coordinate,Piece,Color);
    //adds space coordinates that a piece can move to in a direction
    void searchAndAdd(Coordinate[],int &,Coordinate,int,int,bool);
    //finds the attack squares of all the pieces on the board of the same color
    void findAttackSquares(Coordinate[],int &,Color);
    //checks if one of the two colors in the game is in check
    bool isUnderCheck(Color);
    //gets the movable locations where a piece can move
    void getMovableLocations(Coordinate[],int &,Coordinate,Color);
    //gets all the movable locations where a piece can move
    void getAllMovableLocations(Coordinate[],int &,Color);
    //displays the winning player's status to the screen
    void displayWinningStatus();
    ///@brief check if the move that is made is an attempt to castle
    ///@param from coordpiece moved from
    ///@param toY y coordinate to move to
    ///@return true if ttrying to castle
    bool castleAttempt(Coordinate from,int toY){
        Piece movingPiece=getSpace(from)->getValue();
        if(movingPiece!=King){
```

```
        return false;
    }
    else if(movingPiece==King){
        if(from.y!=4){
            return false;
        }
        else if(toY==6||toY==2){
            return true;
        }
    }
    return true;
}
        ///@brief announces the winner of the game
        ///@param winner the player who is victorious at the end of the game
void announceWinner(int winner){
    string winnerName="";
    string player1=getPlayerName(0);
    string player2=getPlayerName(1);
    string color="";

    if(winner==0){
        winnerName=player1;
        color="WHITE";
    }
    else{
        winnerName=player2;
        color="BLACK";
    }
    cout<<"Congratulations! The winner is "<<winnerName<<", who is playing the color "<<color<<"."<<endl;
}
        ///@brief increases the number of turns
void increaseNumberOfTurns(){
    numberOfTurns++;
}
        ///@brief displays a game menu on the screen
        ///@param input gets an input from the user depending on the user's desire
void menu(int input){
    string player1=getPlayerName(0);
    string player2=getPlayerName(1);
    if(input==0){
        cout<<"Welcome to this game. For information, please select one of the following options below: "<<endl;
        cout<<"'M 1': Display Game Creator Information"<<endl;
        cout<<"'M 2': Display Players"<<endl;
        cout<<"'M 3': Display Current Winning Player Lead Status"<<endl;
        cout<<"'M 4': Display Current Number of Turns"<<endl;
        cout<<"'M 5': Display Board"<<endl;
        cout<<"'M 6': Display Castle Instruction"<<endl;
    }
    else if(input==1){
        cout<<"The game was developed by Attila Koksal for the CSC-17A final project."<<endl;
    }
    else if(input==2){
        cout<<"The players of this game are: "<<endl;
        cout<<"Player 1: "<<player1<<endl;
        cout<<"Player 2: "<<player2<<endl;
    }
```

```
        else if(input==3){
            displayWinningStatus();
        }
        else if(input==4){
            cout<<"Current Number of Turns: "<<numberOfTurns<<endl;
        }
        else if(input==5){
            printBoard();
        }
        else if(input==6){
            cout<<"Castle Instructions: "<<endl;
            cout<<"Short Castle: Move WHITE King to G 1 OR Move BLACK King to G 8 when allowed to castle"<<endl;
            cout<<"Long Castle: Move WHITE King to C 1 OR Move BLACK King to C 8 when allowed to castle"<<endl;
        }
        else{
            cout<<"Sorry, incorrect input."<<endl;
        }
    }
};
void bubbleSort(int [],int);
//main function
int main(int argc, char** argv) {

    int colANDrow[]={8,5,3,6,2,7,1,4};
    int size=sizeof(colANDrow)/sizeof(colANDrow[0]);

    bubbleSort(colANDrow,size);
    cout<<"The game of chess has 8 rows and 8 columns, which totals to 64 squares of space! EX: ";
    for(int i=0;i<size;i++){
        cout<<colANDrow[i]<<" ";
    }
    cout<<endl;

    map<string,string> pieceMove;

    pieceMove.insert(pair<string, string>("Pawn", "Can move once forward on captures or moves"));
    pieceMove.insert(pair<string, string>("Rook", "Can move horizontally or vertically any number of spaces but can't
jump over pieces"));
    pieceMove.insert(pair<string, string>("Knight", "Moves in an L shape with a limit of two spaces and then one space
but in any direction"));
    pieceMove.insert(pair<string, string>("Bishop", "Can move diagonally only for any number of spaces but can't jump
over pieces"));
    pieceMove.insert(pair<string, string>("Queen", "Can move diagonally, horizontally, or vertically for any number of
spaces but can't jump over pieces"));
    pieceMove.insert(pair<string, string>("King", "Can move one space any direction so long as that one space is not
being attacked by an enemy piece"));

    map<string,string>::iterator itr;
    cout<<"Here is some useful information about what each piece in chess can do and cannot."<<endl;
    cout<<"\tPiece\tMovement"<<endl;

    for (itr = pieceMove.begin(); itr != pieceMove.end(); ++itr) {
        cout<<'\t'<<itr->first<<'\t'<<itr->second<<endl;
    }
    cout<<endl;
```

```
    //creates a board from the Board class
    Board board;
    //initializes the board
    board.initializeBoard();
            //prints the board
    board.printBoard();
            //inserts a newline
    cout<<endl;
             //creates a while loop in which it simulates the game between the two colors/players
             //checks if either has emerged victorious and at the end of each turn increases the number of turns
    while(true){
        board.movePiece(WHITE);
        if(board.winner!=-1){
            board.announceWinner(board.winner);
            return 0;
        }
        board.movePiece(BLACK);
        if(board.winner!=-1){
            board.announceWinner(board.winner);
            return 0;
        }
        board.increaseNumberOfTurns();
    }
             //inserts a newline
    cout<<endl;

}
///@brief initializes the game board
void Board::initializeBoard(){
    ///player1 name
    string player1="";
    ///player2 name
    string player2="";
    cout<<"Welcome to the amazing Chess game! You're in for a treat!"<<endl;
    cout<<"The chess pieces all capitalized are the WHITE pieces and the lowercase chess pieces are the BLACK
pieces."<<endl;
    cout<<"Enter a player display name for the WHITE piece: ";
    cin>>player1;
    cout<<endl;
    cout<<"Enter a player display name for the BLACK piece: ";
    cin>>player2;
    cout<<endl;
    //sets player1 name
    setPlayerName(0,player1);
    //sets player2 name
    setPlayerName(1,player2);
    //creates two nested for loops (which loops through the entire board) in which it assigns all spaces on the game
board to the piece value to Empty and the Color to NONE
    for(int i=0;i<ROW;i++){
        for(int j=0;j<COLUMN;j++){
            Space defaultSpace;
            defaultSpace.setValue(Empty,NONE);
            board[i][j]=defaultSpace;
        }
    }
             //initialize all the pieces on the board to default locations
```

```
    setPiece({0,0},Rook,WHITE);
    setPiece({0,1},Knight,WHITE);
    setPiece({0,2},Bishop,WHITE);
    setPiece({0,3},Queen,WHITE);
    setPiece({0,4},King,WHITE);
    setPiece({0,5},Bishop,WHITE);
    setPiece({0,6},Knight,WHITE);
    setPiece({0,7},Rook,WHITE);
    for(int i=0;i<COLUMN;i++){
        setPiece({1,i},Pawn,WHITE);
        setPiece({6,i},Pawn,BLACK);
        for(int j=2;j<=5;j++){
            setPiece({j,i},Empty,NONE);
        }
    }
    setPiece({7,0},Rook,BLACK);
    setPiece({7,1},Knight,BLACK);
    setPiece({7,2},Bishop,BLACK);
    setPiece({7,3},Queen,BLACK);
    setPiece({7,4},King,BLACK);
    setPiece({7,5},Bishop,BLACK);
    setPiece({7,6},Knight,BLACK);
    setPiece({7,7},Rook,BLACK);


}
///@brief prints the game board
void Board::printBoard(){
    //formats the game board
    for(int i=ROW-1;i>=0;i--){
        cout<<"-----------------------------------------------------------"<<endl;
        cout<<i+1<<" ";
        for(int j=0;j<COLUMN;j++){
            cout<<"|";
            cout<<setw(5);
            string pieceString=pieceNames[getSpace({i,j})->getValue()];
            if(getSpace({i,j})->getColor()==WHITE){
                for(auto& x:pieceString){
                    x=toupper(x);
                }
            }
            cout<<pieceString<<" ";

        }
        cout<<endl;
    }
    //prints the row of letters for the rows on the bottom of the board for better game display
    char letters[8]={'A','B','C','D','E','F','G','H'};
    cout<<endl<<"  ";
    for(int i=0;i<COLUMN;i++){
        cout<<setw(6);
        cout<<letters[i]<<" ";
    }
    cout<<endl;
}
///@brief sets a piece with a designated coordinate, piece type, and color on the game board
```

```
///@param coor takes in a coordinate
///@param pece takes in a piece
///@param clr takes in a color
void Board::setPiece(Coordinate coor,Piece pece,Color clr){
    getSpace(coor)->setValue(pece,clr);
}
///@brief gets the space of the given coordinate
///@param coor takes in a coordinate
///@return specific space on the board
Space* Board::getSpace(Coordinate coor){
    return &board[coor.x][coor.y];
}

//-------------SPACE--------------------------------
///@brief gets the value of the space on the board
///@return a piece value
Piece Space::getValue(){
    return value;
}
///@brief gets the color of the space on the board
///@return a color
Color Space::getColor(){
    return color;
}

///@brief sets the value of a specific piece of a color
///@param pece takes in a piece
///@param clr takes in a color
void Space::setValue(Piece pece,Color clr){
    value = pece;
    color = clr;

}
///@brief moves a specific piece on the board for a color
///@param myColor takes in a color
void Board::movePiece(Color myColor){
    //checks if a check is in existence in the game currently
            if(check){
            ///checks to see if a move done by a player on the board results in check mate
        bool foundMove=false;
        ///creates a coordinate array that holds all the movable positions on the board which is 64 (8*8)
            Coordinate allMovable[64];
        ///creates a size and initializes it to zero
            int size=0;
        getAllMovableLocations(allMovable,size,myColor);
        for(int i=0;i<size;i++){
            Piece tempPiece=getSpace(allMovable[i])->getValue();
            Color tempColor=getSpace(allMovable[i])->getColor();
            setPiece(allMovable[i],Pawn,myColor);

            if(!isUnderCheck(myColor)){
                foundMove=true;
            }
            setPiece(allMovable[i],tempPiece,tempColor);
        }
            //if a check mate is found
```

```
        if(!foundMove){
            if(myColor==WHITE){//if the color is WHITE, then WHITE is the winner
                winner=1;
            }
            else{//if the color is BLACK, then BLACK is the winner
                winner=0;
            }
            return;//return
        }
    }
    ///stores info on whether user is attempting to castle
    bool isCastleAttempt=false;
    ///true if user input is acceptable and should not input again
    bool valid=false;
    ///to store user input for column
    char letter;
    ///the space element where user moved a piece from
    Space* startSpace=new Space();
    ///the space element where user moved a piece to end
    Space* endSpace=new Space();
    ///create a start coordinate
    Coordinate start;
    ///create a end coordinate
    Coordinate end;
    ///create a piece called startPiece
    Piece startPiece;
    ///create a color called enemyColor to store the opposing color
    Color enemyColor;
    if(myColor==WHITE){//if the current color is WHITE, then the enemy color is BLACK
        enemyColor=BLACK;
    }
    else{//if the current color is BLACK, then the enemy color is WHITE
        enemyColor=WHITE;
    }
    while(!valid){//while valid is true
        cout<<endl;
            ///creates a string to hold the color
        string colorString="";
        if(myColor==WHITE){//if current color is WHITE, set colorString to "WHITE"
            colorString="WHITE";
        }
        else if(myColor==BLACK){//if current color is BLACK, set colorString to "BLACK"
            colorString="BLACK";
        }
        cout<<"Player "<<colorString<<", enter which piece to move (enter row and column with space in between i.e
(E 2))";
        cout<<"[Or enter 'M 0' for menu display]";
        cin>>letter;//get moving from column
        cin>>start.x;//get moving from row
        if(letter=='m'||letter=='M'){// if user entered m, call menu
            menu(start.x);//pass in menu option
            continue;
        }
        start.x--;//reduce the number to match array index
        start.y=tolower(letter)-'a';//get column number to be moved from
```

```
        cout<<"Player "<<colorString<<", enter where to move to (enter row and column with space in between i.e (E
3))";
        cin>>letter;//get input column to move to
        cin>>end.x;//get moving to row
        end.x--;//reduce the number to match array index
        end.y=tolower(letter)-'a';//get column number to be moved to

        isCastleAttempt=castleAttempt(start,end.y);//check if user is trying to castle

        startSpace=getSpace(start);//get space of moving square
        endSpace=getSpace(end);//get space of the target square
        startPiece=startSpace->getValue();//get piece type of the moving piece

        if(endSpace->getColor()==myColor){//if the moving target location is already moving player's color, that move is
not allowed
            cout<<"how can you move there if there is already a piece there of yours";
            continue;
        }
        if(startSpace->getColor()!=myColor){//if the moving piece is not moving player's color, that move is not allowed
            cout<<"Sorry you can only move your piece"<<endl;
            continue;
        }
        ///array to load in all movable spaces of a piece
        Coordinate ableToMove[64];
        int size=0;
        getMovableLocations(ableToMove,size,start,myColor);//load movable locations into array for a piece

        if(myColor==WHITE){
            if(castleAble[0][0]==true){//if white can long castle
                if(getSpace({0,3})->getValue()==Empty&&getSpace({0,2})->getValue()==Empty&&getSpace({0,1})-
>getValue()==Empty&&getSpace({0,0})->getValue()==Rook){
                    ableToMove[size]={0,2};//user can move to row 0 column 2 by king if wants to castle left
                    size++;
                }
            }
            if(castleAble[0][1]==true){//if white can short  castle
                if(getSpace({0,5})->getValue()==Empty&&getSpace({0,6})->getValue()==Empty&&getSpace({0,7})-
>getValue()==Rook){
                    ableToMove[size]={0,6};//user can move to row 0 column 6 by king if wants to castle right
                    size++;
                }
            }
        }
        else if(myColor==BLACK){//if user is playing black
            if(castleAble[1][0]==true){//if black can long castle
                if(getSpace({7,3})->getValue()==Empty&&getSpace({7,2})->getValue()==Empty&&getSpace({7,1})-
>getValue()==Empty&&getSpace({7,0})->getValue()==Rook){
                    ableToMove[size]={7,2};//user can move to row 8 column 2 by king if wants to castle left
                    size++;
                }
            }
            if(castleAble[1][1]==true){//if black can short castle
                if(getSpace({7,5})->getValue()==Empty&&getSpace({7,6})->getValue()==Empty&&getSpace({7,7})-
>getValue()==Rook){
                    //user can move to row 8 column 7 by king if wants to castle right
                        ableToMove[size]={7,6};
```

```
                size++;//increase array size
            }
        }
    }
    //----------------------check if valid move
    //end.x end.y
    Piece replacedPiece;//save target location piece info in case a reset is needed
    Color replacedColor;//save target location color info in case a reset is needed
    ///tracks if the location user wants to move to is a location that user can move to
        bool found=false;

    for(int i=0;i<size;i++){
        if(ableToMove[i].x==end.x&&ableToMove[i].y==end.y){// the location that user wants to move to is in the list
of where user can move to
            found=true;
            break;
        }
    }
    if(found){
        replacedPiece=endSpace->getValue();//get piece of the space to be moved to
        replacedColor=endSpace->getColor();//get color of the space to be moved to
        endSpace->setValue(startPiece,myColor);//move piece
        startSpace->setValue(Empty,NONE);//move piece
        if(isCastleAttempt){
            if(end.y==6){//castle is to the right
                getSpace({start.x,7})->setValue(Empty,NONE);//perform the castle
                getSpace({start.x,5})->setValue(Rook,myColor);//perform the castle
            }
            else if(end.y==2){//castle is to the left
                getSpace({start.x,0})->setValue(Empty,NONE);//perform the castle
                getSpace({start.x,3})->setValue(Rook,myColor);//perform the castle
            }
        }
        valid=true;//can stop looping
    }
    else{
        cout<<"Sorry you cannot move to that location."<<endl;
        continue;//restart the function, make new move
    }
    if(check&&isUnderCheck(myColor)){
        startSpace->setValue(startPiece,myColor);//space to be moved from
        endSpace->setValue(replacedPiece,replacedColor);//space to be moved to
        if(isCastleAttempt){//current move is trying to castle
            if(end.y==6){//castling right side
                getSpace({start.x,5})->setValue(Empty,NONE);//move back the castle
                getSpace({start.x,7})->setValue(Rook,myColor);//move back the castle
            }
            else if(end.y==2){//castling left sid e
                getSpace({start.x,3})->setValue(Empty,NONE);//move back the castle
                getSpace({start.x,0})->setValue(Rook,myColor);//move back the castle
            }
        }
        cout<<"You cannot do that while under check."<<endl;
        valid=false;
        continue;//restart the funciton
    }
```

```
        if(isUnderCheck(myColor)){//player playing is under check
            startSpace->setValue(startPiece,myColor);//the space to be moved from
            endSpace->setValue(replacedPiece,replacedColor);//the space to be moved to
            if(isCastleAttempt){//was trying to castle
                if(end.y==6){//castle was right side
                    getSpace({start.x,5})->setValue(Empty,NONE);//move back the castle
                    getSpace({start.x,7})->setValue(Rook,myColor);//move back the castle
                }
                else if(end.y==2){//castle was left side
                    getSpace({start.x,3})->setValue(Empty,NONE);//move back the castle
                    getSpace({start.x,0})->setValue(Rook,myColor);//move back the castle
                }
            }
            cout<<"You cannot do that, that move causes a check from the opponent."<<endl;
            valid=false;//restart the function
            continue;//restart
        }
        if(valid){
            if(startPiece==King){// if moving piece is a king
                if(myColor==WHITE){// if checking color si white
                    castleAble[0][0]=false;//king moved so no castleing allowed on that side
                    castleAble[0][1]=false;//king moved so no castleing allowed on that side
                }
                else if(myColor==BLACK){//if checking color is black
                    castleAble[1][0]=false;//king moved so no castleing allowed on that side
                    castleAble[1][1]=false;//king moved so no castleing allowed on that side
                }
            }
            else if(startPiece==Rook){//if the piece moving is a rook
                if(start.y==7){// if left size of board
                    if(myColor==WHITE){// if it is white color
                        castleAble[0][1]=false;//rook moved so no castleing allowed on that side
                    }
                    else if(myColor==BLACK){// if it is black color
                        castleAble[1][1]=false;//rook moved so no castleing allowed on that side
                    }
                }
                else if(start.y==0){// if left size of board
                    if(myColor==WHITE){// if it is white color
                        castleAble[0][0]=false;//rook moved so no castleing allowed on that side
                    }
                    else if(myColor==BLACK){// if it is black color
                        castleAble[1][0]=false;//rook moved so no castleing allowed on that side
                    }
                }
            }
        }
        check=isUnderCheck(enemyColor);//check if enemy is under check
        Board::printBoard();//print board
        if(check){
            cout<<"CHECK!"<<endl;//display message to tell user that he is under check
        }
    }
}
///@brief goes through the board in one direction, and adds spaces to array that is on the path until incounter a piece
///@param arr is the array to add spaces to
```

```
///@param size is the array size
///@param startCoord the coord to start from
///@param xd the x offset for direction to move to
///@param yd the y offset for direction to move to
///@param attackable is the bool that tracks if we should add our ally piece when encountered
void Board::searchAndAdd(Coordinate arr[],int &size,Coordinate startCoord,int xd,int yd,bool attackable){
    ///a "pointer" that goes through the board
    Coordinate currentCoord=startCoord;
    ///get the color of the piece being checked
    Color playerColor=getSpace(startCoord)->getColor();
    currentCoord.x+=xd;//x move to first offset
    currentCoord.y+=yd;//y move to first offset
    while(currentCoord.x<ROW&&currentCoord.y<COLUMN&&currentCoord.x>=0&&currentCoord.y>=0){// if in bound
        if(getSpace(currentCoord)->getValue()==Empty){//empty space
            arr[size]=currentCoord;//add current coord to array of pieces attacked
            size++;//increase array size
        }
        else if(getSpace(currentCoord)->getColor()!=playerColor&&getSpace(currentCoord)-
>getColor()!=NONE){//enemy
            arr[size]=currentCoord;//add current coord to array of pieces attacked
            size++;//increase array size
            break;//stop the loop
        }
        else if(getSpace(currentCoord)->getColor()==playerColor){// if the piece is ally color
            if(attackable){// if under attack mode, ally piece counts
                arr[size]=currentCoord;//add current coord to attacking spaces array
                size++;//increase array size
            }
            break;//stop checking next one because we cannot pass thorugh our own piece
        }
        currentCoord.x+=xd;//move to next coord
        currentCoord.y+=yd;//move to next coord
    }
}
///@param find all squares that is a color is attacking
///@param ableToMove array to hold all squares that a color is attacking
///@param size is size of the array
///@param myColor is the color that is being evaluated
void Board::findAttackSquares(Coordinate ableToMove[],int &size,Color myColor){
    for(int i=0;i<ROW;i++){
        for(int j=0;j<COLUMN;j++){
            Coordinate currentCoord={i,j};//current viewing coord
            Piece currentPiece=getSpace(currentCoord)->getValue();//current viewing piece type
            Space* currentSpace=getSpace(currentCoord);//current viewing space
            if(getSpace(currentCoord)->getColor()!=myColor){// if the space being checked is not the color concerned
skip the space
                continue;
            }

            if(currentPiece==Pawn){// if the piece being checked is a pawn

                int offset;
                if(currentSpace->getColor()==WHITE){//if viewing black white, front is one row up
                    offset=1;
                }
                else if(currentSpace->getColor()==BLACK){//if viewing black pawn, front is one row down
```

```
                offset=-1;
            }
            if((currentCoord.y-1)>0){
                    // add the current viewing coord to movable array
                ableToMove[size]={currentCoord.x+offset,currentCoord.y-1};
                size++;//increase array size
            }
            if(currentCoord.y+1<COLUMN){
                    // add the current viewing coord to movable array
                ableToMove[size]={currentCoord.x+offset,currentCoord.y+1};
                size++;//increase array size
            }
        }
        else if(currentPiece==Rook){// if the piece being checked is a rook
            searchAndAdd(ableToMove,size,currentCoord,1,0,true); // up
            searchAndAdd(ableToMove,size,currentCoord,0,1,true); // right
            searchAndAdd(ableToMove,size,currentCoord,0,-1,true); // left
            searchAndAdd(ableToMove,size,currentCoord,-1,0,true); // down
        }
        else if(currentPiece==Bishop){// if the piece being checked is a bishop
            searchAndAdd(ableToMove,size,currentCoord,1,1,true); // up right
            searchAndAdd(ableToMove,size,currentCoord,-1,-1,true); // bottom left
            searchAndAdd(ableToMove,size,currentCoord,1,-1,true); // up left
            searchAndAdd(ableToMove,size,currentCoord,-1,1,true); // bottom right
        }
        else if(currentPiece==Knight){// if the piece being checked is a knight
            int availablePos[][2]={{1,2},{2,1},{-2,1},{-2,-1},{-1,2},{2,-1},{-1,-2},{1,-2}};//positional offsets for the knight
            for(int i=0;i<8;i++){
                Coordinate usedToCheck={currentCoord.x+availablePos[i][0],currentCoord.y+availablePos[i][1]};//space
that is being checked

if(usedToCheck.x<ROW&&usedToCheck.y<COLUMN&&usedToCheck.x>=0&&usedToCheck.y>=0){//check for out of
bounds
                    if(getSpace(usedToCheck)->getColor()!=NONE){
                        ableToMove[size]=usedToCheck;//add currently checking space into the movable space array
                        size++;//increase array size
                    }
                }
            }
        }
        else if(currentPiece==Queen){// if the piece being checked is a queen
            searchAndAdd(ableToMove,size,currentCoord,1,0,true); // up
            searchAndAdd(ableToMove,size,currentCoord,0,1,true); // right
            searchAndAdd(ableToMove,size,currentCoord,0,-1,true); // left
            searchAndAdd(ableToMove,size,currentCoord,-1,0,true); // down
            searchAndAdd(ableToMove,size,currentCoord,1,1,true); // up right
            searchAndAdd(ableToMove,size,currentCoord,-1,-1,true); // bottom left
            searchAndAdd(ableToMove,size,currentCoord,1,-1,true); // up left
            searchAndAdd(ableToMove,size,currentCoord,-1,1,true); // bottom right
        }
        else if(currentPiece==King){// if the piece being checked is a king
            int availablePos[][2]={{1,1},{1,0},{-1,0},{-1,-1},{1,-1},{-1,1},{0,1},{0,-1}};///positional offsets for king to
move (i.e. 1,1 means one row up and one column right)
            for(int i=0;i<8;i++){
                    ///coordinate that is being checked
                Coordinate usedToCheck={currentCoord.x+availablePos[i][0],currentCoord.y+availablePos[i][1]};
```

```
                if(usedToCheck.x<ROW&&usedToCheck.y<COLUMN&&usedToCheck.x>=0&&usedToCheck.y>=0){// if
not out of bound
                    //if the piece is enemy piece or empty
                    if(getSpace(usedToCheck)->getColor()!=myColor){
                        //add the space found into array of movable spaces
                            ableToMove[size]=usedToCheck;
                        size++;//size of array
                    }
                }
            }
        }
        }
    }
}
///@brief check if a color is being checked
///@param testColor that color that is being verified of check status
///@return true if test color is being checked
bool Board::isUnderCheck(Color testColor){
    Color enemyColor;//get enemy color
    if(testColor==WHITE){
        enemyColor=BLACK;
    }
    else{
        enemyColor=WHITE;
    }
    ///coordinate to hold location of the king
    Coordinate myKing;
    for(int i=0;i<ROW;i++){
        for(int j=0;j<COLUMN;j++){
            if(getSpace({i,j})->getValue()==King&&getSpace({i,j})->getColor()==testColor){// if a king found of the
color
                myKing={i,j};//location of the king
                break;//since the king is found, stop looping
            }
        }
    }
    ///holds all spaces that is being attacked
    Coordinate attackMoves[64];
    int attackMoveSize=0;//size of array
    findAttackSquares(attackMoves,attackMoveSize,enemyColor);//gets all spaces that is under attack from enemy
pieces
    for(int i=0;i<attackMoveSize;i++){
        if(attackMoves[i]==myKing){//if king is under attack
            return true;
            break;
        }
    }
    return false;
}
///@brief loads into param array all spaces that a color can move to
///@param ableToMove array to store all spaces that a color can move to
///@param size tracks array size
///@param myColor tracks the color that is being observed
void Board::getAllMovableLocations(Coordinate ableToMove[],int& size,Color myColor){
    for(int i=0;i<ROW;i++){
```

```
        for(int j=0;j<COLUMN;j++){
            if(getSpace({i,j})->getColor()!=myColor){//if the piece is not the color that is being checking, ignore
                continue;
            }
            //get all space that a piece can move to, and load it into the able to move array
            getMovableLocations(ableToMove,size,{i,j},myColor);// load all position that this piece can move to
        }
    }
}

///@brief add all space the piece can move to to a array
///@param abletoMoveResult an array to hold spaces that can be moved to
///@param resultsize array size
///@param start is the coordinate of the space that is being evaluated
///@param myColor is the color of the piece being evaluated
void Board::getMovableLocations(Coordinate ableToMoveResult[],int &resultSize,Coordinate start,Color myColor){
    ///array to record all spaces the piece can move to
    Coordinate ableToMove[64];
    ///array size
    int size=0;
    ///the piece that is being moved
    Piece startPiece=getSpace(start)->getValue();
    ///a space where a piece is moved from
    Space* startSpace=getSpace(start);
    if(startPiece==Pawn){//if user player is trying to move a pawn
        //row off set for different color pawn
        int offset;
        if(startSpace->getColor()==WHITE){// if white pawn, ahead is one row up
            offset=1;
        }
        else if(startSpace->getColor()==BLACK){//if black pawn, ahead is one row down
            offset=-1;
        }
        // if the space ahead of the pawn is empty
        if(getSpace({start.x+offset,start.y})->getValue()==Empty){
            //if the space can be moved to, add it to array
            ableToMove[size]={start.x+offset,start.y};
            size++;//increase array size
        }
        if((start.y-1)>0){
            // if it is enemy color, and is on one block on diagonal
            if(getSpace({start.x+offset,start.y-1})->getColor()!=myColor&&getSpace({start.x+offset,start.y-1})-
>getColor()!=NONE){
                //if the space can be moved to, add it to array
                        ableToMove[size]={start.x+offset,start.y-1};
                size++;//increase array size
            }
        }
        if(start.y+1<COLUMN){
            // if it is enemy color, and is on one block on diagonal
            if(getSpace({start.x+offset,start.y+1})->getColor()!=myColor&&getSpace({start.x+offset,start.y+1})-
>getColor()!=NONE){
                //if the space can be moved to, add it to array
                ableToMove[size]={start.x+offset,start.y+1};
                size++;//increase array size
            }
```

```
        }
        Coordinate firstCheck={start.x+offset,start.y};///< the space before the pawn
        Coordinate secondCheck={start.x+(offset*2),start.y}; ///< two space ahead of the pawn
        if(getSpace(firstCheck)->getValue()==Empty&&getSpace(secondCheck)->getValue()==Empty){
            if((getSpace(start)->getColor()==WHITE&&start.x==1)||(getSpace(start)-
>getColor()==BLACK&&start.x==6)){
                        //if the space can be moved to, add it to array
                ableToMove[size]=secondCheck;
                size++;//increase array size
            }
        }
    }
    else if(startPiece==Rook){//if user player is trying to move a rook
        searchAndAdd(ableToMove,size,start,1,0,false); // up
        searchAndAdd(ableToMove,size,start,0,1,false); // right
        searchAndAdd(ableToMove,size,start,0,-1,false); // left
        searchAndAdd(ableToMove,size,start,-1,0,false); // down
    }
    else if(startPiece==Bishop){//if user player is trying to move a bishop
        searchAndAdd(ableToMove,size,start,1,1,false); // up right
        searchAndAdd(ableToMove,size,start,-1,-1,false); // bottom left
        searchAndAdd(ableToMove,size,start,1,-1,false); // up left
        searchAndAdd(ableToMove,size,start,-1,1,false); // bottom right
    }
    else if(startPiece==Knight){//if user player is trying to move a Knight
        int availablePos[][2]={{1,2},{2,1},{-2,1},{-2,-1},{-1,2},{2,-1},{-1,-2},{1,-2}};
        for(int i=0;i<8;i++){
            /// coordinate that we are checking if a king can move to
            Coordinate usedToCheck={start.x+availablePos[i][0],start.y+availablePos[i][1]};
            if(usedToCheck.x<ROW&&usedToCheck.y<COLUMN&&usedToCheck.x>=0&&usedToCheck.y>=0){

                if(getSpace(usedToCheck)->getColor()!=myColor){
                    //if the space can be moved to, add it to array
                    ableToMove[size]=usedToCheck;
                    size++;//increase array size
                }
            }
        }
    }
    else if(startPiece==Queen){
        searchAndAdd(ableToMove,size,start,1,0,false); // up
        searchAndAdd(ableToMove,size,start,0,1,false); // right
        searchAndAdd(ableToMove,size,start,0,-1,false); // left
        searchAndAdd(ableToMove,size,start,-1,0,false); // down
        searchAndAdd(ableToMove,size,start,1,1,false); // up right
        searchAndAdd(ableToMove,size,start,-1,-1,false); // bottom left
        searchAndAdd(ableToMove,size,start,1,-1,false); // up left
        searchAndAdd(ableToMove,size,start,-1,1,false); // bottom right
    }
    else if(startPiece==King){// if user is trying to move a king
        int availablePos[][2]={{1,1},{1,0},{-1,0},{-1,-1},{1,-1},{-1,1},{0,1},{0,-1}}; ///< all the offsetes where a king can
move to(i.e 1,1 means one row up and one column right)
        for(int i=0;i<8;i++){
            /// coordinate that we are checking if a king can move to
            Coordinate usedToCheck={start.x+availablePos[i][0],start.y+availablePos[i][1]};
            if(usedToCheck.x<ROW&&usedToCheck.y<COLUMN&&usedToCheck.x>=0&&usedToCheck.y>=0){
```

```
                if(getSpace(usedToCheck)->getColor()!=myColor){
                    //if the space can be moved to, add it to array
                    ableToMove[size]=usedToCheck;
                    size++;
                }
            }
        }
            ///stores all squares that enemy are attacking
        Coordinate attackAble[64];
            ///stores the size of the array
        int attackSize=0;
            ///holds enemy color
        Color enemyColor;

        if(myColor==WHITE){//get enemy colors
            enemyColor=BLACK;
        }
        else{
            enemyColor=WHITE;
        }
        findAttackSquares(attackAble,attackSize,enemyColor);//add all enemy attacking spaces into the array
        /*
        for(int i=0;i<attackSize;i++){
            cout<<attackAble[i].x+1<<":"<<attackAble[i].y+1<<endl;
        }
        */
        for(int i=0;i<size;i++){
            for(int j=0;j<attackSize;j++){
                if(ableToMove[i]==attackAble[j]){// for all space that a kind can move to, if it is in the attack range of an
enemy, remove that moveable space from the list
                    ableToMove[i]=ableToMove[size-1];
                    size--;
                    i--;
                }
            }
        }
    }
    for(int i=0;i<size;i++){
        ableToMoveResult[resultSize]=ableToMove[i];// add able to move temp array to able to move input array passed
in this function
        resultSize++;//increase size
    }
}
///@brief displays who is winning and by how much
/// it is based on how many pieces each player have on the board
void Board::displayWinningStatus(){
    int scoreValues[]={1,5,3,3,9,0}; ///score value for each piece, ordered by its index number in enum
    /// records score for player one and two
    int playerScores[2]={0,0};
    /// holds name of players, can be accessed by player index, 0 for player1 1 for player2
    string playerNames[2]={getPlayerName(0),getPlayerName(1)};

    for(int i=0;i<ROW;i++){// loop through the board and calculate how many piece is on the board
        for(int j=0;j<COLUMN;j++){
            ///the piece that is currently being viewed
            Piece currentPiece=getSpace({i,j})->getValue();
```

```
            ///the color that is currently being viewed
            Color currentColor=getSpace({i,j})->getColor();

            if(currentColor==WHITE&&currentPiece!=Empty){//count score for white
              playerScores[0]+=scoreValues[currentPiece];
            }
            else if(currentColor==BLACK&&currentPiece!=Empty){//count score for black
                playerScores[1]+=scoreValues[currentPiece];
            }
        }
    }
    ///stores 0 if player1 is winning, 1 if player2 is winning, -1 if tie
    int winningPlayerNumber=maxItem(playerScores[0],playerScores[1]);
    if(winningPlayerNumber==-1){
        cout<<"Based on the pieces currently on the board, the game is in a tie position."<<endl;
        return;
    }
    ///scores the score difference between players
    int scoreDifference=abs(playerScores[0]-playerScores[1]);
    cout<<"Current winning player is "<<playerNames[winningPlayerNumber]<<endl;
    cout<<"Score difference is "<<scoreDifference<<" points (based on number of pieces still on the board)"<<endl;
}
void bubbleSort(int arr[],int number){
    if (number == 1)
        return;

    for (int i=0;i<number-1;i++)
        if (arr[i]>arr[i+1])
            swap(arr[i], arr[i+1]);

    bubbleSort(arr,number-1);
}
```