Main

Header Code

Class implementations

//  
// Created by bento on 11/20/2024.  
//  
  
#ifndef BISHOP\_H  
#define BISHOP\_H  
#include "Position.h"  
  
//the bishop class holds our bishop's getMove Function  
class Bishop {  
public:  
 //this stores the position from which the bishop class is created. We use this so we can access the board from inside the bishop class  
 Position \*position;  
 //this coordinate is the coordinate of the bishop we are getting moves for  
 const Coordinate location;  
 Bishop(Position \*position, const Coordinate &location);  
  
 void getMoves(Move moves[], int &length);  
};  
  
  
  
#endif //BISHOP\_H

//  
// Created by bento on 11/21/2024.  
//  
  
#ifndef PAWN\_H  
#define PAWN\_H  
#include "Position.h"  
  
//this class holds all our Pawn logic  
class Pawn {  
public:  
 //this allows us to access the chessboard from within the Pawn class  
 Position \*position;  
 //This holds the location of the pawn  
 const Coordinate &location;  
  
 Pawn(Position \*position, const Coordinate &location);  
  
 //adds the pawns legal moves to moves[] and adjusts length accordingly  
 void getMoves(Move moves[], int &length);  
  
};  
  
  
  
#endif //PAWN\_H

//  
// Created by bento on 11/11/2024.  
//  
  
/\*  
 \*This class will be used to pass around potential moves. It stores the start and end coordinates so that they can  
 \*easily be checked for legality.  
 \*There are no getters or setters because that requires an extra level of abstraction which slows the engine down.  
\*/  
#ifndef MOVE\_H  
#define MOVE\_H  
#include "Coordinate.h"  
  
  
class Move {  
 public:  
 //this holds the start square of the move  
 Coordinate start;  
 //this holds the end square where the piece will end up  
 Coordinate end;  
  
 //these are used when the Position::unMakeMove function is called to unmake a move.  
 int capturingPiece;  
 int capturedPiece;  
 //  
  
 Move(Coordinate start, Coordinate end);  
 Move();  
 //this returns all the variables from the move class in a nice format to be printed  
 std::string toString();  
 static std::string *toString*(Move moves[]);  
};  
  
  
  
#endif //MOVE\_H

//  
// Created by bento on 11/11/2024.  
//  
  
  
/\*  
 \*this class will hold a full position of the chess board, including the piece location,  
 \*who's turn it is(color), and the last move that was made.  
 \*  
\*/  
#ifndef POSITION\_H  
#define POSITION\_H  
#include "Move.h"  
  
  
class Position {  
 public:  
 int (&board)[8][8];  
 //should be -1 for black,and 1 for white  
 int color = 0;  
 //the lastMove is needed for en-pessant-ing pawns  
 Move lastMove;  
  
  
 Position(int (&board)[8][8], Move lastMove, int color);  
  
 //outputs the position class in a nice format  
 void print();  
  
  
 //adds all the legal moves to the moves[] array  
 int getLegalMoves(Move moves[]);  
  
 //this checks whether the king is in check. It first makes the move it is passed then checks  
 bool checkForKingDanger(Move &move);  
 //this checks whether the king is in check. It runs on the position as is, without making any moves  
 bool checkForKingDanger();  
  
 //returns a double value that represents the evaluation for a position  
 double getEvaluation();  
 //takes a move and makes it on the board  
 void makeMove(Move &move);  
 //takes a move and un-makes it  
 void unMakeMove(Move &move);  
  
 //used to access coordinates on the chess board  
 int getCoordinate(const Coordinate &coordinate);  
 void setCoordinate(const Coordinate &coordinate, const int &piece);  
  
 //we use this function to check whether two squares hold the same color piece. It helps when determining whether a move is legal  
 bool areSameColor(const Coordinate &first, const Coordinate &second);  
  
 //this is used by the checkForKingDanger functions to find the king position  
 Coordinate findKingPosition();  
  
 //this is the Master function if you will. It is what you call on an instance of position to find the best move.  
 Move getBestMove(int depth);  
  
 //this is a helper function to the getBestMove. It uses recursion to search to a depth.  
 int search(int depth);  
};  
  
  
#endif //POSITION\_H

//  
// Created by Will on 11/21/2024.  
//  
  
#ifndef QUEEN\_H  
#define QUEEN\_H  
#include "Position.h"  
//this class holds all our Queen logic  
class Queen {  
 public:  
 //this allows us to access the chessboard from within the Queen class  
 Position\* position;  
 //holds this queens position  
 Coordinate location;  
 Queen(Position \*position, const Coordinate &location);  
  
 //adds all the moves the queen can make to moves[] and adjusts length accordingly  
 void getMoves(Move moves[], int &length);  
};  
  
#endif //QUEEN\_H

//  
// Created by Will on 11/20/2024.  
//  
  
#ifndef KNIGHT\_H  
#define KNIGHT\_H  
#include "Position.h"  
  
//holds all our logic used for interacting with knight pieces  
class Knight {  
public:  
 //this lets us access the board from the position where the knight was created  
 Position\* position;  
 //this holds the location of the knight on the chess board  
 Coordinate location;  
 Knight(Position \*position, const Coordinate &location);  
 //adds all the legal knight moves to moves[] and adjusts length accordingly  
 void getMoves(Move moves[], int &length);  
};  
  
  
  
#endif //KNIGHT\_H

//  
// Created by Will on 11/21/2024.  
//  
  
#ifndef KING\_H  
#define KING\_H  
#include "Position.h"  
  
  
//this holds all our king logic  
class King {  
public:  
 //stores the position class from which the King is created. it lets us access the board from within the king class  
 Position\* position;  
 //holds the kings location  
 Coordinate location;  
 King(Position \*position, const Coordinate &location);  
 //adds all the kings moves to the moves[] and increases length accordingly  
 void getMoves(Move moves[], int &length);  
};  
  
  
  
#endif //KING\_H

//  
// Created by bento on 11/11/2024.  
//  
  
/\*  
this class will hold our coordinate for a piece. anytime you want to specify a square on the chess  
board you use this class. Normally we would use getter and setter functions to access the rank and file variables  
in the class, but that adds one extra layer of abstraction. Because we will need to access these thousands of times  
a second that will slow the engine down. That's why there aren't any getters or setters.  
\*/  
  
#ifndef COORDINATE\_H  
#define COORDINATE\_H  
#include <string>  
  
class Coordinate {  
public:  
 int rank;  
 int file;  
 Coordinate(int file, int rank);  
 Coordinate();  
 //this gives a representation like this "(R,F)" to make it easier to print the contents of a coordinate.  
 std::string toString() const;  
 //checks whether the coordinate falls within the eight by eight chessboard  
 bool inBounds() const;  
};  
  
  
  
#endif //COORDINATE\_H

//  
// Created by bento on 11/21/2024.  
//  
  
#ifndef ROOK\_H  
#define ROOK\_H  
#include "Position.h"  
  
//this class holds all our Rook piece logic  
class Rook {  
public:  
 //this allows us to access the chessboard from within the Rook class  
 Position \*position;  
 //stores the rooks location on the board  
 const Coordinate &location;  
 Rook(Position \*position, const Coordinate &location);  
  
 //adds all the rooks legal moves to moves[]  
 void getMoves(Move moves[], int &length);  
};  
  
  
  
#endif //ROOK\_H

//  
// Created by bento on 11/21/2024.  
//  
  
#include "Rook.h"  
  
Rook::Rook(Position \*position, const Coordinate &location): position(position), location(location) {}  
  
  
  
  
void Rook::getMoves(Move moves[], int &length) {  
 //these for loops iterate in each direction the rook can travel until the rook hits something at which point a break statement is used to pop out of that direction  
 //rank positive  
 for(int i = 1; i < 7; i++) {  
 Coordinate endCoordinate = Coordinate(location.file, location.rank + i);  
 if(endCoordinate.inBounds()) {  
 if(position->areSameColor(location,endCoordinate)) {  
 break;  
 }  
 if(position->getCoordinate(endCoordinate) != 0){  
 moves[length++] = Move(location, endCoordinate);  
 break;  
 }  
 moves[length++] = Move(location, endCoordinate);  
 }else {  
 break;  
 }  
 }  
 //rank negative  
 for(int i = 1; i < 7; i++) {  
 Coordinate endCoordinate = Coordinate(location.file, location.rank - i);  
 if(endCoordinate.inBounds()) {  
 if(position->areSameColor(location,endCoordinate)) {  
 break;  
 }  
 if(position->getCoordinate(endCoordinate) != 0){  
 moves[length++] = Move(location, endCoordinate);  
 break;  
 }  
 moves[length++] = Move(location, endCoordinate);  
 }else {  
 break;  
 }  
 }  
 //file positive  
 for(int i = 1; i < 7; i++) {  
 Coordinate endCoordinate = Coordinate(location.file + i, location.rank);  
 if(endCoordinate.inBounds()) {  
 if(position->areSameColor(location,endCoordinate)) {  
 break;  
 }  
 if(position->getCoordinate(endCoordinate) != 0){  
 moves[length++] = Move(location, endCoordinate);  
 break;  
 }  
 moves[length++] = Move(location, endCoordinate);  
 }else {  
 break;  
 }  
 }  
 //file negative  
 for(int i = 1; i < 7; i++) {  
 Coordinate endCoordinate = Coordinate(location.file - i, location.rank);  
 if(endCoordinate.inBounds()) {  
 if(position->areSameColor(location,endCoordinate)) {  
 break;  
 }  
 if(position->getCoordinate(endCoordinate) != 0){  
 moves[length++] = Move(location, endCoordinate);  
 break;  
 }  
 moves[length++] = Move(location, endCoordinate);  
 }else {  
 break;  
 }  
 }  
  
}

//  
// Created by Will on 11/21/2024.  
//  
  
#include "Queen.h"  
Queen::Queen(Position \*position, const Coordinate &location): position(position), location(location) {}  
  
// this is a predetermined list of all possible queen moves... basically just a combination of rook and bishop  
void Queen::getMoves(Move moves[], int &length) {  
 //see rook/bishop classes to see how these for loops work  
 //rank positive  
 for(int i = 1; i < 7; i++) {  
 Coordinate endCoordinate = Coordinate(location.file, location.rank + i);  
 if(endCoordinate.inBounds()) {  
 if(position->areSameColor(location,endCoordinate)) {  
 break;  
 }  
 if(position->getCoordinate(endCoordinate) != 0){  
 moves[length++] = Move(location, endCoordinate);  
 break;  
 }  
 moves[length++] = Move(location, endCoordinate);  
 }else {  
 break;  
 }  
 }  
 //rank negative  
 for(int i = 1; i < 7; i++) {  
 Coordinate endCoordinate = Coordinate(location.file, location.rank - i);  
 if(endCoordinate.inBounds()) {  
 if(position->areSameColor(location,endCoordinate)) {  
 break;  
 }  
 if(position->getCoordinate(endCoordinate) != 0){  
 moves[length++] = Move(location, endCoordinate);  
 break;  
 }  
 moves[length++] = Move(location, endCoordinate);  
 }else {  
 break;  
 }  
 }  
 //file positive  
 for(int i = 1; i < 7; i++) {  
 Coordinate endCoordinate = Coordinate(location.file + i, location.rank);  
 if(endCoordinate.inBounds()) {  
 if(position->areSameColor(location,endCoordinate)) {  
 break;  
 }  
 if(position->getCoordinate(endCoordinate) != 0){  
 moves[length++] = Move(location, endCoordinate);  
 break;  
 }  
 moves[length++] = Move(location, endCoordinate);  
 }else {  
 break;  
 }  
 }  
 //file negative  
 for(int i = 1; i < 7; i++) {  
 Coordinate endCoordinate = Coordinate(location.file - i, location.rank);  
 if(endCoordinate.inBounds()) {  
 if(position->areSameColor(location,endCoordinate)) {  
 break;  
 }  
 if(position->getCoordinate(endCoordinate) != 0){  
 moves[length++] = Move(location, endCoordinate);  
 break;  
 }  
 moves[length++] = Move(location, endCoordinate);  
 }else {  
 break;  
 }  
 }  
 //positive and positive  
 for(int i = 1; i < 7; i++) {  
 Coordinate endCoordinate = Coordinate(location.file + i, location.rank + i);  
 if(endCoordinate.inBounds()) {  
 if(position->areSameColor(location,endCoordinate)) {  
 break;  
 }  
 if(position->getCoordinate(endCoordinate) != 0){  
 moves[length++] = Move(location, endCoordinate);  
 break;  
 }  
 moves[length++] = Move(location, endCoordinate);  
 }else {  
 break;  
 }  
 }  
 //negative and negative  
 for(int i = 1; i < 7; i++) {  
 Coordinate endCoordinate = Coordinate(location.file - i, location.rank - i);  
 if(endCoordinate.inBounds()) {  
 if(position->areSameColor(location,endCoordinate)) {  
 break;  
 }  
 if(position->getCoordinate(endCoordinate) != 0){  
 moves[length++] = Move(location, endCoordinate);  
 break;  
 }  
 moves[length++] = Move(location, endCoordinate);  
 }else {  
 break;  
 }  
 }  
 //negative and positive  
 for(int i = 1; i < 7; i++) {  
 Coordinate endCoordinate = Coordinate(location.file - i, location.rank + i);  
 if(endCoordinate.inBounds()) {  
 if(position->areSameColor(location,endCoordinate)) {  
 break;  
 }  
 if(position->getCoordinate(endCoordinate) != 0){  
 moves[length++] = Move(location, endCoordinate);  
 break;  
 }  
 moves[length++] = Move(location, endCoordinate);  
 }else {  
 break;  
 }  
 }  
 //positive and negative  
 for(int i = 1; i < 7; i++) {  
 Coordinate endCoordinate = Coordinate(location.file + i, location.rank - i);  
 if(endCoordinate.inBounds()) {  
 if(position->areSameColor(location,endCoordinate)) {  
 break;  
 }  
 if(position->getCoordinate(endCoordinate) != 0){  
 moves[length++] = Move(location, endCoordinate);  
 break;  
 }  
 moves[length++] = Move(location, endCoordinate);  
 }else {  
 break;  
 }  
 }  
}

//  
// Created by bento on 11/21/2024.  
//  
  
#include "Pawn.h"  
  
#include <iostream>  
  
  
Pawn::Pawn(Position \*position, const Coordinate &location): position(position), location(location) {}  
  
void Pawn::getMoves(Move moves[], int &length) {  
  
 //if the pawn is white it has different moves that the black pawns  
 if (position->getCoordinate(location) > 0) { //white pawns  
 Coordinate upOne = Coordinate(location.file,location.rank + 1);  
 Coordinate upTwo = Coordinate(location.file,location.rank + 2);  
 Coordinate captureLeft = Coordinate(location.file -1,location.rank + 1);  
 Coordinate captureRight = Coordinate(location.file + 1,location.rank + 1);  
 if(upOne.inBounds() && position->getCoordinate(upOne) == 0) {  
 moves[length++] = Move(location, upOne);  
 if(upTwo.inBounds() && position->getCoordinate(upTwo) == 0 && location.rank == 1) {  
 moves[length++] = Move(location, upTwo);  
 }  
 }  
 if(captureLeft.inBounds() && position->getCoordinate(captureLeft) < 0) {  
 moves[length++] = Move(location, captureLeft);  
 }  
 if(captureRight.inBounds() && position->getCoordinate(captureRight) < 0) {  
 moves[length++] = Move(location, captureRight);  
 }  
  
  
 //En-pessant  
  
 if(position->getCoordinate(position->lastMove.end) == -100 //Checks whether the last move was a black pawn  
 && position->lastMove.end.rank - position->lastMove.start.rank == -2 //checks that the pawn moved two spaces  
 && abs(position->lastMove.end.file - location.file) == 1 //checks that it is on either side of the current pawn  
 && position->lastMove.end.rank == location.rank) {  
 moves[length++] = Move(location, Coordinate(position->lastMove.end.file, position->lastMove.end.rank+1));  
 }  
  
 //we're finished with white pawn logic now it is black pawns turn  
 }else { //black pawns  
 //these are the predetermined available moves for pawns  
 Coordinate upOne = Coordinate(location.file,location.rank - 1);  
 Coordinate upTwo = Coordinate(location.file,location.rank - 2);  
 Coordinate captureLeft = Coordinate(location.file -1,location.rank - 1);  
 Coordinate captureRight = Coordinate(location.file + 1,location.rank - 1);  
 //  
  
 //checks all the moves to see if they apply to this board.  
 if(upOne.inBounds() && position->getCoordinate(upOne) == 0) {  
 moves[length++] = Move(location, upOne);  
 if(upTwo.inBounds() && position->getCoordinate(upTwo) == 0 && location.rank == 6) {  
 moves[length++] = Move(location, upTwo);  
 }  
 }  
 if(captureLeft.inBounds() && position->getCoordinate(captureLeft) > 0) {  
 moves[length++] = Move(location, captureLeft);  
 }  
 if(captureRight.inBounds() && position->getCoordinate(captureRight) > 0) {  
 moves[length++] = Move(location, captureRight);  
 }  
  
  
  
 //En-pessant  
 if(position->getCoordinate(position->lastMove.end) == 100 //Checks whether the last move was a black pawn  
 && position->lastMove.end.rank - position->lastMove.start.rank == 2 //checks that the pawn moved two spaces  
 && abs(position->lastMove.end.file - location.file) == 1 //checks that it is on either side of the current pawn  
 && position->lastMove.end.rank == location.rank  
 ) {  
 moves[length++] = Move(location, Coordinate(position->lastMove.end.file, position->lastMove.end.rank-1));  
 }  
 }  
}

//  
// Created by bento on 11/11/2024.  
//  
  
/\*  
 \*This class will be used to pass around potential moves. It stores the start and end coordinates so that they can  
 \*easily be checked for legality.  
 \*There are no getters or setters because that requires an extra level of abstraction which slows the engine down.  
\*/  
#include "Move.h"  
  
Move::Move(Coordinate start, Coordinate end): start(-1,-1), end(-1,-1) {  
 this->start = start;  
 this->end = end;  
  
}  
Move::Move(): start(), end() {}  
  
//toStrings return a nicely formatted string representation of the class  
std::string Move::toString() {  
 return start.toString() + " " + end.toString();  
}  
std::string Move::*toString*(Move moves[]) {  
 std::string result = "";  
 for(int i = 0; i < 50; i++) {  
 result += moves[i].toString() + "\n";  
 }  
 return result;  
}

//  
// Created by Will on 11/20/2024.  
//  
  
#include "Knight.h"  
  
Knight::Knight(Position\* position, const Coordinate &location): position(position), location(location) {}  
  
void Knight::getMoves(Move moves[], int &length) {  
  
 // this is a predetermined list of all possible knight moves  
 Coordinate allEndCoordinates[] = {  
 Coordinate(location.file+2, location.rank-1),  
 Coordinate(location.file+2, location.rank+1),  
 Coordinate(location.file+1, location.rank-2),  
 Coordinate(location.file+1, location.rank+2),  
 Coordinate(location.file-1, location.rank-2),  
 Coordinate(location.file-1, location.rank+2),  
 Coordinate(location.file-2, location.rank-1),  
 Coordinate(location.file-2, location.rank+1),  
 };  
 // adds all the moves that are actually legal  
 for (int i = 0; i < 8; i++) {  
 if (allEndCoordinates[i].inBounds()) {  
 if(!position->areSameColor(location, allEndCoordinates[i])){  
 moves[length] = Move(location, allEndCoordinates[i]);  
 length++;  
 }  
  
 }  
 }  
}

//  
// Created by Will on 11/21/2024.  
//  
  
#include "King.h"  
  
  
King::King(Position\* position, const Coordinate &location): position(position), location(location) {}  
  
  
void King::getMoves(Move moves[], int &length) {  
 // this is a predetermined list of all possible king moves  
 Coordinate allEndCoordinates[] = {  
 Coordinate(location.file-1, location.rank+1),  
 Coordinate(location.file+0, location.rank+1),  
 Coordinate(location.file+1, location.rank+1),  
 Coordinate(location.file+1, location.rank+0),  
 Coordinate(location.file+1, location.rank-1),  
 Coordinate(location.file+0, location.rank-1),  
 Coordinate(location.file-1, location.rank-1),  
 Coordinate(location.file-1, location.rank+0),  
 };  
 // adds all the moves that are actually legal  
 for (int i = 0; i < 8; i++) {  
 if (allEndCoordinates[i].inBounds()) {  
 if(!position->areSameColor(location, allEndCoordinates[i])){  
 moves[length] = Move(location, allEndCoordinates[i]);  
 length++;  
 }  
 }  
 }  
}

//  
// Created by bento on 11/11/2024.  
//  
/\*  
this class will hold our coordinate for a piece. anytime you want to specify a square on the chess  
board you use this class. Normally we would use getter and setter functions to access the rank and file variables  
in the class, but that adds one extra layer of abstraction. Because we will need to access these thousands of times  
a second that will slow the engine down. That's why there aren't any getters or setters.  
\*/  
#include "Coordinate.h"  
  
Coordinate::Coordinate(const int file, const int rank) {  
 this->rank = rank;  
 this->file = file;  
}  
Coordinate::Coordinate() {  
 this->rank = -1;  
 this->file = -1;  
}  
  
  
std::string Coordinate::toString() const {  
 return "(" + std::to\_string(rank) + "," + std::to\_string(file) + ")";  
}  
  
bool Coordinate::inBounds() const {  
 return (rank >= 0 && rank < 8 && file >= 0 && file < 8);  
}

//  
// Created by bento on 11/20/2024.  
//  
  
#include "Bishop.h"  
  
#include <iostream>  
#include <ostream>  
  
Bishop::Bishop(Position \*position, const Coordinate &location): position(position), location(location) {}  
  
  
  
  
void Bishop::getMoves(Move moves[], int &length) {  
 //positive and positive  
 for(int i = 1; i < 7; i++) {  
 Coordinate endCoordinate = Coordinate(location.file + i, location.rank + i);  
 if(endCoordinate.inBounds()) {  
 if(position->areSameColor(location,endCoordinate)) {  
 break;  
 }  
 if(position->getCoordinate(endCoordinate) != 0){  
 moves[length++] = Move(location, endCoordinate);  
 break;  
 }  
 moves[length++] = Move(location, endCoordinate);  
 }else {  
 break;  
 }  
 }  
 //negative and negative  
 for(int i = 1; i < 7; i++) {  
 Coordinate endCoordinate = Coordinate(location.file - i, location.rank - i);  
 if(endCoordinate.inBounds()) {  
 if(position->areSameColor(location,endCoordinate)) {  
 break;  
 }  
 if(position->getCoordinate(endCoordinate) != 0){  
 moves[length++] = Move(location, endCoordinate);  
 break;  
 }  
 moves[length++] = Move(location, endCoordinate);  
 }else {  
 break;  
 }  
 }  
 //negative and positive  
 for(int i = 1; i < 7; i++) {  
 Coordinate endCoordinate = Coordinate(location.file - i, location.rank + i);  
 if(endCoordinate.inBounds()) {  
 if(position->areSameColor(location,endCoordinate)) {  
 break;  
 }  
 if(position->getCoordinate(endCoordinate) != 0){  
 moves[length++] = Move(location, endCoordinate);  
 break;  
 }  
 moves[length++] = Move(location, endCoordinate);  
 }else {  
 break;  
 }  
 }  
 //positive and negative  
 for(int i = 1; i < 7; i++) {  
 Coordinate endCoordinate = Coordinate(location.file + i, location.rank - i);  
 if(endCoordinate.inBounds()) {  
 if(position->areSameColor(location,endCoordinate)) {  
 break;  
 }  
 if(position->getCoordinate(endCoordinate) != 0){  
 moves[length++] = Move(location, endCoordinate);  
 break;  
 }  
 moves[length++] = Move(location, endCoordinate);  
 }else {  
 break;  
 }  
 }  
}

//  
// Created by bento on 11/11/2024.  
//  
  
#include "Position.h"  
#include <iostream>  
#include <set>  
  
#include "Bishop.h"  
#include "King.h"  
#include "Knight.h"  
#include "Pawn.h"  
#include "Queen.h"  
#include "Rook.h"  
  
// The position class is the class that holds the 2D array that represents the chess board.  
// All access and change of values on the board should flow through this class.  
// It holds all the getters and setters for the chess board.  
  
Position::Position(int (&board)[8][8], Move lastMove, int color): board(board), lastMove(lastMove), color(color) {}  
  
// this is what is called when printing the board  
void Position::print() {  
 for (int rank = 0; rank < 8; rank++) {  
 for (int file = 0; file < 8; file++) {  
 std::cout << board[rank][file] << "\t";  
 }  
 std::cout << std::endl;  
 }  
 std::cout << "Color = " << color << ", LastMove = " << lastMove.toString() << std::endl;  
}  
  
//this will get all the possible moves for the position  
//this should only return legal moves  
int Position::getLegalMoves(Move moves[]){  
 Move allLegalMoves[50];  
 int length = 0, finalLength = 0;  
 for(int i = 0; i < 8; i++) {  
 for(int j = 0; j < 8; j++) {  
 switch(board[i][j]\*color) { // the absolute value means that our switch case can be half as long we just need to calculate the color in each case then.  
 case 100: //pawn  
 Pawn(this, Coordinate(j,i)).getMoves(allLegalMoves, length);  
 break;  
 case 300: //Knight  
 Knight(this, Coordinate(j, i)).getMoves(allLegalMoves, length);  
 break;  
 case 350: //bishop  
 Bishop(this, Coordinate(j, i)).getMoves(allLegalMoves, length);  
 break;  
 case 500: //rook  
 Rook(this, Coordinate(j, i)).getMoves(allLegalMoves, length);  
 break;  
 case 900: //queen  
 Queen(this, Coordinate(j, i)).getMoves(allLegalMoves, length);  
 break;  
 case 10000: //king  
 King(this, Coordinate(j, i)).getMoves(allLegalMoves, length);  
 break;  
 default: //empty square  
 break;  
 }  
 }  
 }  
  
 for(int i = 0; i < length; i++) {  
 if(!checkForKingDanger(allLegalMoves[i])) {  
 moves[finalLength++] = allLegalMoves[i];  
 }  
 }  
 return finalLength;  
}  
  
//this checks whether the king can be captured after the move is made  
bool Position::checkForKingDanger(Move &move) {  
  
 makeMove(move);  
 Coordinate location = findKingPosition();  
  
 //KING  
 Coordinate allKEndCoordinates[] = {  
 Coordinate(location.file-1, location.rank+1),  
 Coordinate(location.file+0, location.rank+1),  
 Coordinate(location.file+1, location.rank+1),  
 Coordinate(location.file+1, location.rank+0),  
 Coordinate(location.file+1, location.rank-1),  
 Coordinate(location.file+0, location.rank-1),  
 Coordinate(location.file-1, location.rank-1),  
 Coordinate(location.file-1, location.rank+0),  
 };  
 for (int i = 0; i < 8; i++) {  
 if (allKEndCoordinates[i].inBounds()) {  
 getCoordinate(location); // current position  
 getCoordinate(allKEndCoordinates[i]); // this is the possible move  
 if (getCoordinate(allKEndCoordinates[i]) == 10000\*color) {  
 unMakeMove(move);  
 return true;  
 }  
  
  
 }  
 }  
  
 //KNIGHT  
 Coordinate allEndCoordinates[] = {  
 Coordinate(location.file+2, location.rank-1),  
 Coordinate(location.file+2, location.rank+1),  
 Coordinate(location.file+1, location.rank-2),  
 Coordinate(location.file+1, location.rank+2),  
 Coordinate(location.file-1, location.rank-2),  
 Coordinate(location.file-1, location.rank+2),  
 Coordinate(location.file-2, location.rank-1),  
 Coordinate(location.file-2, location.rank+1),  
 };  
 for (int i = 0; i < 8; i++) {  
 if (allEndCoordinates[i].inBounds() && getCoordinate(allEndCoordinates[i]) == 300\*color) {  
 unMakeMove(move);  
 return true;  
 }  
 }  
  
 //ROOK AND QUEEN  
 //rank positive  
 for(int i = 1; i < 7; i++) {  
 Coordinate endCoordinate = Coordinate(location.file, location.rank + i);  
 if(endCoordinate.inBounds()) {  
 if(areSameColor(location,endCoordinate)) {  
 break;  
 }  
 if(getCoordinate(endCoordinate) == 500\*color || getCoordinate(endCoordinate) == 900\*color) {  
 unMakeMove(move);  
 return true;  
 }  
 }else {  
 break;  
 }  
 }  
 //rank negative  
 for(int i = 1; i < 7; i++) {  
 Coordinate endCoordinate = Coordinate(location.file, location.rank - i);  
 if(endCoordinate.inBounds()) {  
 if(areSameColor(location,endCoordinate)) {  
 break;  
 }  
 if(getCoordinate(endCoordinate) == 500\*color || getCoordinate(endCoordinate) == 900\*color) {  
 unMakeMove(move);  
 return true;  
 }  
 }else {  
 break;  
 }  
 }  
 //file positive  
 for(int i = 1; i < 7; i++) {  
 Coordinate endCoordinate = Coordinate(location.file + i, location.rank);  
 if(endCoordinate.inBounds()) {  
 if(areSameColor(location,endCoordinate)) {  
 break;  
 }  
 if(getCoordinate(endCoordinate) == 500\*color || getCoordinate(endCoordinate) == 900\*color) {  
 unMakeMove(move);  
 return true;  
 }  
 }else {  
 break;  
 }  
 }  
 //file negative  
 for(int i = 1; i < 7; i++) {  
 Coordinate endCoordinate = Coordinate(location.file - i, location.rank);  
 if(endCoordinate.inBounds()) {  
 if(areSameColor(location,endCoordinate)) {  
 break;  
 }  
 if(getCoordinate(endCoordinate) == 500\*color || getCoordinate(endCoordinate) == 900\*color) {  
 unMakeMove(move);  
 return true;  
 }  
 }else {  
 break;  
 }  
 }  
  
 //BISHOP AND QUEEN  
 //positive and positive  
 for(int i = 1; i < 7; i++) {  
 Coordinate endCoordinate = Coordinate(location.file + i, location.rank + i);  
 if(endCoordinate.inBounds()) {  
 if(areSameColor(location,endCoordinate)) {  
 break;  
 }  
 if(getCoordinate(endCoordinate) == 350\*color || getCoordinate(endCoordinate) == 900\*color) {  
 unMakeMove(move);  
 return true;  
 }  
 }else {  
 break;  
 }  
 }  
 //negative and negative  
 for(int i = 1; i < 7; i++) {  
 Coordinate endCoordinate = Coordinate(location.file - i, location.rank - i);  
 if(endCoordinate.inBounds()) {  
 if(areSameColor(location,endCoordinate)) {  
 break;  
 }  
 if(getCoordinate(endCoordinate) == 350\*color || getCoordinate(endCoordinate) == 900\*color) {  
 unMakeMove(move);  
 return true;  
 }  
 }else {  
 break;  
 }  
 }  
 //negative and positive  
 for(int i = 1; i < 7; i++) {  
 Coordinate endCoordinate = Coordinate(location.file - i, location.rank + i);  
 if(endCoordinate.inBounds()) {  
 if(areSameColor(location,endCoordinate)) {  
 break;  
 }  
 if(getCoordinate(endCoordinate) == 350\*color || getCoordinate(endCoordinate) == 900\*color) {  
 unMakeMove(move);  
 return true;  
 }  
 }else {  
 break;  
 }  
 }  
 //positive and negative  
 for(int i = 1; i < 7; i++) {  
 Coordinate endCoordinate = Coordinate(location.file + i, location.rank - i);  
 if(endCoordinate.inBounds()) {  
 if(areSameColor(location,endCoordinate)) {  
 break;  
 }  
 if(getCoordinate(endCoordinate) == 350\*color || getCoordinate(endCoordinate) == 900\*color) {  
 unMakeMove(move);  
 return true;  
 }  
 }else {  
 break;  
 }  
 }  
  
 unMakeMove(move);  
 return false;  
}  
  
//this checks if the current position can be captured  
bool Position::checkForKingDanger() {  
 Coordinate location = findKingPosition();  
 //KNIGHT -- see knight class to see how this works  
 Coordinate allEndCoordinates[] = {  
 Coordinate(location.file+2, location.rank-1),  
 Coordinate(location.file+2, location.rank+1),  
 Coordinate(location.file+1, location.rank-2),  
 Coordinate(location.file+1, location.rank+2),  
 Coordinate(location.file-1, location.rank-2),  
 Coordinate(location.file-1, location.rank+2),  
 Coordinate(location.file-2, location.rank-1),  
 Coordinate(location.file-2, location.rank+1),  
 };  
 for (int i = 0; i < 8; i++) {  
 if (allEndCoordinates[i].inBounds() && getCoordinate(allEndCoordinates[i]) == 300\*color) {  
 return true;  
 }  
 }  
  
 //ROOK AND QUEEN -- see rook/queen class to see how this works  
 //rank positive  
 for(int i = 1; i < 7; i++) {  
 Coordinate endCoordinate = Coordinate(location.file, location.rank + i);  
 if(endCoordinate.inBounds()) {  
 if(areSameColor(location,endCoordinate)) {  
 break;  
 }  
 if(getCoordinate(endCoordinate) == 500\*color || getCoordinate(endCoordinate) == 900\*color) {  
 return true;  
 }  
 }else {  
 break;  
 }  
 }  
  
 //rank negative  
 for(int i = 1; i < 7; i++) {  
 Coordinate endCoordinate = Coordinate(location.file, location.rank - i);  
 if(endCoordinate.inBounds()) {  
 if(areSameColor(location,endCoordinate)) {  
 break;  
 }  
 if(getCoordinate(endCoordinate) == 500\*color || getCoordinate(endCoordinate) == 900\*color) {  
 return true;  
 }  
 }else {  
 break;  
 }  
 }  
 //file positive  
 for(int i = 1; i < 7; i++) {  
 Coordinate endCoordinate = Coordinate(location.file + i, location.rank);  
 if(endCoordinate.inBounds()) {  
 if(areSameColor(location,endCoordinate)) {  
 break;  
 }  
 if(getCoordinate(endCoordinate) == 500\*color || getCoordinate(endCoordinate) == 900\*color) {  
 return true;  
 }  
 }else {  
 break;  
 }  
 }  
 //file negative  
 for(int i = 1; i < 7; i++) {  
 Coordinate endCoordinate = Coordinate(location.file - i, location.rank);  
 if(endCoordinate.inBounds()) {  
 if(areSameColor(location,endCoordinate)) {  
 break;  
 }  
 if(getCoordinate(endCoordinate) == 500\*color || getCoordinate(endCoordinate) == 900\*color) {  
 return true;  
 }  
 }else {  
 break;  
 }  
 }  
  
 //BISHOP AND QUEEN -- see bishop/queen class to see how this works  
 //positive and positive  
 for(int i = 1; i < 7; i++) {  
 Coordinate endCoordinate = Coordinate(location.file + i, location.rank + i);  
 if(endCoordinate.inBounds()) {  
 if(areSameColor(location,endCoordinate)) {  
 break;  
 }  
 if(getCoordinate(endCoordinate) == 350\*color || getCoordinate(endCoordinate) == 900\*color) {  
 return true;  
 }  
 }else {  
 break;  
 }  
 }  
  
 //negative and negative  
 for(int i = 1; i < 7; i++) {  
 Coordinate endCoordinate = Coordinate(location.file - i, location.rank - i);  
 if(endCoordinate.inBounds()) {  
 if(areSameColor(location,endCoordinate)) {  
 break;  
 }  
 if(getCoordinate(endCoordinate) == 350\*color || getCoordinate(endCoordinate) == 900\*color) {  
 return true;  
 }  
 }else {  
 break;  
 }  
 }  
  
 //negative and positive  
 for(int i = 1; i < 7; i++) {  
 Coordinate endCoordinate = Coordinate(location.file - i, location.rank + i);  
 if(endCoordinate.inBounds()) {  
 if(areSameColor(location,endCoordinate)) {  
 break;  
 }  
 if(getCoordinate(endCoordinate) == 350\*color || getCoordinate(endCoordinate) == 900\*color) {  
 return true;  
 }  
 }else {  
 break;  
 }  
 }  
  
 //positive and negative  
 for(int i = 1; i < 7; i++) {  
 Coordinate endCoordinate = Coordinate(location.file + i, location.rank - i);  
 if(endCoordinate.inBounds()) {  
 if(areSameColor(location,endCoordinate)) {  
 break;  
 }  
 if(getCoordinate(endCoordinate) == 350\*color || getCoordinate(endCoordinate) == 900\*color) {  
 return true;  
 }  
 }else {  
 break;  
 }  
 }  
 return false;  
}  
  
//this adds up all the piece values on the board in the position.  
double Position::getEvaluation() {  
 double sum = 0;  
 for(int i = 0; i < 8; i++) {  
 for(int j = 0; j < 8; j++) {  
 sum += board[i][j];  
 }  
 }  
 return sum;  
}  
  
//takes a move and makes it on the board  
void Position::makeMove(Move &move) {  
 //this line of code saves whatever value is in the end coordinate of the move so that it can be restored if the move is unmade  
 move.capturedPiece = getCoordinate(move.end);  
 move.capturingPiece = getCoordinate(move.start);  
 setCoordinate(move.end, move.capturingPiece);  
 setCoordinate(move.start, 0);  
 this->lastMove = move;  
  
 //this is the promotion for white  
 if(move.end.rank == 7 && getCoordinate(move.end) == 100) {  
 setCoordinate(move.end, 900);  
 }  
 //this is the promotion for white  
 if(move.end.rank == 0 && getCoordinate(move.end) == -100) {  
 setCoordinate(move.end, -900);  
 }  
 color\*=-1;  
}  
  
//takes a move and unmakes it on the board  
void Position::unMakeMove(Move &move) {  
 setCoordinate(move.start, move.capturingPiece);  
 setCoordinate(move.end, move.capturedPiece);  
 color\*=-1;  
}  
  
//please use these functions to access coordinates of the board it can be kind of tricky because accessing the array is backwards from how it makes sense in our heads.  
int Position::getCoordinate(const Coordinate &coordinate) {  
 return board[coordinate.rank][coordinate.file];  
}  
void Position::setCoordinate(const Coordinate &coordinate, const int &piece) {  
 board[coordinate.rank][coordinate.file] = piece;  
}  
  
//simple function to determine whether two squares hold pieces of the same color  
bool Position::areSameColor(const Coordinate &first, const Coordinate &second) {  
 if(getCoordinate(first) > 0 && getCoordinate(second) > 0) {  
 return true;  
 }  
 if(getCoordinate(first) < 0 && getCoordinate(second) < 0) {  
 return true;  
 }  
 return false;  
}  
  
// this fxn is used by the fxn checkForKingDanger to find the king's position  
Coordinate Position::findKingPosition() {  
 for(int i = 0; i < 8; i++) {  
 for(int j = 0; j < 8; j++) {  
 if(board[i][j] == -10000\*color) {  
 return Coordinate(j,i);  
 }  
 }  
 }  
 return Coordinate();  
}  
  
// this fxn is used to find the best move  
Move Position::getBestMove(int depth) {  
 Move moves[50];  
 Move bestMove;  
 int bestMoveEvaluation = -100000000;  
 int length = this->getLegalMoves(moves);  
 for(int i = 0; i < length; i++) {  
  
 this->makeMove(moves[i]);  
 int currentEval = this->search(depth-1);  
 //this keeps track of the best move that has been found so far.  
 if (currentEval > bestMoveEvaluation) {  
 bestMoveEvaluation = currentEval;  
 bestMove = moves[i];  
 }  
 this->unMakeMove(moves[i]);  
 }  
 return bestMove;  
}  
  
//this is the vanilla search evaluation function without any extra optimization.  
int Position::search(int depth) {  
 //if we have reached the end of our depth then we return the static evaluation.  
 if (depth <= 0) {  
 return this->getEvaluation();  
 }  
 //otherwise we get the moves in the position.  
 Move moves[50];  
 int length = this->getLegalMoves(moves);  
 //check that there are available moves  
 if (length <= 0) {  
 //if there aren't any and the king is in danger then it is checkmate  
 this->color \*= -1; //this line switches the color to check the opposing kings danger not our own  
 if (checkForKingDanger()) {  
 this->color \*= -1; //this line switches the color to check the opposing kings danger not our own  
 return 1000000;  
 }  
 this->color \*= -1; //this line switches the color to check the opposing kings danger not our own  
 //otherwise it is stalemate which evaluates to 0  
 return 0;  
 }  
  
 int worstMoveEvaluation = 100000000;  
 for(int i = 0; i < length; i++) {  
 this->makeMove(moves[i]);  
 //this negative sign is very important because it flips the eval for black and white  
 int currentEval = -this->search(depth-1);  
  
 //we pick the evaluation that is closest to zero because that is what the opponent will always choose.  
 if (currentEval < worstMoveEvaluation) {  
 worstMoveEvaluation = currentEval;  
 }  
 this->unMakeMove(moves[i]);  
 }  
 return worstMoveEvaluation;  
}

#include <iostream>  
#include "Coordinate.h"  
#include "Position.h"  
  
//  
// Created by bento on 11/11/2024.  
//  
  
  
int main() {  
  
 // empty board  
 int board[8][8] = {  
 {0,0,0,0,0,0,0,0},  
 {0,0,0,0,0,0,0,0},  
 {0,0,0,0,0,0,0,0},  
 {0,0,0,0,0,0,0,0},  
 {0,0,0,0,0,0,0,0},  
 {0,0,0,0,0,0,0,0},  
 {0,0,0,0,0,0,0,0},  
 {0,0,0,0,0,0,0,0},  
 };  
  
 // initializing instance of position class  
 Position position = Position(board, Move(Coordinate(0,0), Coordinate(0,0)), 1);  
  
 //beginning of User Interface  
 // initial message/directions  
 std::cout << "WELCOME TO CHESS ENGINE!!!\n";  
 std::cout << "--------------------------\n";  
 std::cout << "Enter your chessboard by entering values for each position.\n";  
 std::cout << "--> pawns = 100, knights = 300, bishops = 350, rooks = 500, queens = 900, and kings = 10000\n";  
 std::cout << "--> NEGATIVE values represent black's pieces & POSITIVE values represent white's pieces\n";  
 std::cout << "--> Enter 0 for empty squares\n";  
 std::cout << "--> Coordinate (0,0) is the upper left-hand corner of the board & (7,7) is the lower right-hand corner.\n\n";  
  
 // loop to let user enter values for each position  
 for (int rank = 0; rank < 8; rank++) {  
 for (int file = 0; file < 8; file++) {  
 std::cout << "Enter value for position (" << rank << ", " << file << "): ";  
 std::cin >> board[rank][file];  
 }  
 }  
  
 // display the updated board to the user  
 std::cout << "\n\nThank you for entering your board!\n";  
 std::cout << "This is what it looks like:\n";  
 std::cout << "------------------------------------------------------------\n";  
 position.print();  
 std::cout << "------------------------------------------------------------\n";  
  
 // ask for who's turn it is and what depth level to search at  
 std::cout << "\nWhich color's turn is it? (enter -1 for black and 1 for white) : ";  
 std::cin >> position.color;  
 std::cout << "\n";  
 std::cout << "How deep do you want to search? (Recommended >= 3)";  
 int depth;  
 std::cin >> depth;  
  
 // output the best possible move  
 std::cout << "\nThe best possible move given this board is:\n";  
 std::cout << "------------------------------------------------------------\n";  
 Move bestMove = position.getBestMove(depth);  
 position.makeMove(bestMove);  
 position.print();  
 std::cout << "------------------------------------------------------------\n";  
  
 // thank you message  
 std::cout << "\nThank you for using our chess engine!\n";  
  
 return 0;  
  
}  
  
//Example Position 1 - Mate in One  
//Best Move - (2,3) - (0,3)  
//Position - 0 0 0 0 0 -10000 0 0 0 0 0 0 0 0 0 0 0 0 0 500 0 10000 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
//Color - 1  
//Depth - 3  
  
//Example Position 2 - Mate in Four  
//Best Move - (1,3) - (0,3)  
//Position - 0 0 0 0 0 -300 -10000 0 0 0 0 500 0 0 0 0 0 0 0 0 0 10000 0 0 0 0 0 0 0 0 0 0 0 0 -100 0 0 0 0 0 0 0 100 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
//Color - 1  
//Depth - 5