```c
// chess.c by Ilya Sherstyuk

#include <ctype.h>
#include <ncurses.h>
#include <curses.h>
#include <signal.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cs50.h>
#include <time.h>
#include <wchar.h>
#include <unistd.h>

#define MAX_PIECE_MOVES 30
#define MAX_TOTAL_MOVES 16 * MAX_PIECE_MOVES

// color pair names

#define WHITEPAIR 1
#define BLACKPAIR 2
#define BLUEPAIR 3
#define BOARDPAIR 4


typedef struct
{
    char board[8][8];
    // board[file][rank]
    // board[0][0] == a1
    // board[7][7] == h8
    // board[4][3] == e4

    char turn;
    // 'w': white's turn
    // 'b': black's turn

    int enPassant[2];
    // en passant square

    int moves[MAX_TOTAL_MOVES][2][2];
    int movesLength;
    // list of all possible moves in this position

    int bestMove;
    // index of the best move in moves[]

    int castlingRights[4];
    // castlingRights[0] : white, kingside
    // castlingRights[1] : white, queenside
    // castlingRights[2] : black, kingside
    // castlingRights[3] : black, queenside

} position;

typedef struct
{
    int coords[2];
    // [0]: rank
    // [1]: file

    char color;
    // 'w' or 'b'

    int listMoves[MAX_PIECE_MOVES][2];
    int listMovesLength;

    char type;
    // piece type: 'K', 'q', 'R', etc

} piece;

typedef struct
{
    int evaluation;
    int move[2][2];

} ply;


void drawBoard(char board[][8]);
void clearBoard(void);
void playMove(position *position_ptr, int moveFrom[], int moveTo[]);
void deselectPiece(void);
bool checkLegalMove(position *position_ptr, int moveFrom[], int moveTo[]);
void listPawnMoves(position *position_ptr, piece *piece_ptr);
void listKnightMoves(position *position_ptr, piece *piece_ptr);
void listKingMoves(position *position_ptr, piece *piece_ptr);
void listRookMoves(position *position_ptr, piece *piece_ptr);
void listBishopMoves(position *position_ptr, piece *piece_ptr);
void append(piece *piece_ptr, int file, int rank);
char getColor(char testPiece);
void listAllMoves(position *position_ptr);
int evaluate(position *position_ptr);
ply findBestMove(position *position_ptr, int depth, int alpha, int beta);
int charToUnicode(char piece);
int difficulty;
int highlightedDifficulty;

// the actual board state
position realPosition;

// full computer evaluation
int computer_eval;


int highlightedPiece[2];
int selectedPiece[2];
int inputPiece[2];
// [0]: file (a-h)
// [1]: rank (1-8)

// for ncurses
int top;
int left;

// makeshift dictionary of piece ascii art
char pieces_index[7] = {'p', 'r', 'n', 'b', 'q', 'k', 'x'};
char pieces_ascii[7][7] = {"_O_", "[\"]", "{`\\", "(\\)", "\\^/", "\\+/", "[\"]"};


// piece-square tables taken from https://chessprogramming.wikispaces.com/Simplified%20evaluation%20function
// this website is a very good resource
int pawnPositions[8][8] =  {
        {0,  0,  0,  0,  0,  0,  0,  0},
        {50, 50, 50, 50, 50, 50, 50, 50},
        {10, 10, 20, 30, 30, 20, 10, 10},
        {5,  5, 10, 25, 25, 10,  5,  5},
        {0,  0,  0, 20, 20,  0,  0,  0},
        {5, -5,-10,  0,  0,-10, -5,  5},
        {5, 10, 10,-20,-20, 10, 10,  5},
        {0,  0,  0,  0,  0,  0,  0,  0}};
    int knightPositions[8][8] = {
        {-50,-40,-30,-30,-30,-30,-40,-50},
        {-40,-20,  0,  0,  0,  0,-20,-40},
        {-30,  0, 10, 15, 15, 10,  0,-30},
        {-30,  5, 15, 20, 20, 15,  5,-30},
        {-30,  0, 15, 20, 20, 15,  0,-30},
        {-30,  5, 10, 15, 15, 10,  5,-30},
        {-40,-20,  0,  5,  5,  0,-20,-40},
        {-50,-40,-30,-30,-30,-30,-40,-50}};
    int bishopPositions[8][8] = {
        {-20,-10,-10,-10,-10,-10,-10,-20},
        {-10,  0,  0,  0,  0,  0,  0,-10},
        {-10,  0,  5, 10, 10,  5,  0,-10},
        {-10,  5,  5, 10, 10,  5,  5,-10},
        {-10,  0, 10, 10, 10, 10,  0,-10},
        {-10, 10, 10, 10, 10, 10, 10,-10},
        {-10,  5,  0,  0,  0,  0,  5,-10},
        {-20,-10,-10,-10,-10,-10,-10,-20}};
    int rookPositions[8][8] = {
        {0,  0,  0,  0,  0,  0,  0,  0},
        {5, 10, 10, 10, 10, 10, 10,  5},
        {-5,  0,  0,  0,  0,  0,  0, -5},
        {-5,  0,  0,  0,  0,  0,  0, -5},
        {-5,  0,  0,  0,  0,  0,  0, -5},
        {-5,  0,  0,  0,  0,  0,  0, -5},
        {-5,  0,  0,  0,  0,  0,  0, -5},
        {0,  0,  0,  5,  5,  0,  0,  0}};
    int queenPositions[8][8] = {
        {-20,-10,-10, -5, -5,-10,-10,-20},
        {-10,  0,  0,  0,  0,  0,  0,-10},
        {-10,  0,  5,  5,  5,  5,  0,-10},
        {-5,  0,  5,  5,  5,  5,  0, -5 },
        {0,  0,  5,  5,  5,  5,  0, -5  },
        {-10,  5,  5,  5,  5,  5,  0,-10},
        {-10,  0,  5,  0,  0,  0,  0,-10},
        {-20,-10,-10, -5, -5,-10,-10,-20}};
    int kingPositionsMid[8][8] = {
        {-30,-40,-40,-50,-50,-40,-40,-30},
        {-30,-40,-40,-50,-50,-40,-40,-30},
        {-30,-40,-40,-50,-50,-40,-40,-30},
        {-30,-40,-40,-50,-50,-40,-40,-30},
        {-20,-30,-30,-40,-40,-30,-30,-20},
        {-10,-20,-20,-20,-20,-20,-20,-10},
        {20, 20, -5, -10, -10, -5, 20,20},
        {20, 30, 10,  0,  0, 10, 30, 20}};
    int kingPositionsEnd[8][8] = {
        {-50,-40,-30,-20,-20,-30,-40,-50},
        {-30,-20,-10,  0,  0,-10,-20,-30},
        {-30,-10, 20, 30, 30, 20,-10,-30},
        {-30,-10, 30, 40, 40, 30,-10,-30},
        {-30,-10, 30, 40, 40, 30,-10,-30},
        {-30,-10, 20, 30, 30, 20,-10,-30},
        {-30,-30,  0,  0,  0,  0,-30,-30},
        {-50,-30,-30,-30,-30,-30,-30,-50}};


// finally
int main(void)
{
    char initialBoard[8][8] = {{'R', 'P', ' ', ' ', ' ', ' ', 'p', 'r'} ,
                              {'N', 'P', ' ', ' ', ' ', ' ', 'p', 'n'} ,
                              {'B', 'P', ' ', ' ', ' ', ' ', 'p', 'b'} ,
                              {'Q', 'P', ' ', ' ', ' ', ' ', 'p', 'q'} ,
                              {'K', 'P', ' ', ' ', ' ', ' ', 'p', 'k'} ,
                              {'B', 'P', ' ', ' ', ' ', ' ', 'p', 'b'} ,
                              {'N', 'P', ' ', ' ', ' ', ' ', 'p', 'n'} ,
                              {'R', 'P', ' ', ' ', ' ', ' ', 'p', 'r'}};



    // initialize real Position
    // inputs the initial board into the real Position

    for (int i = 0; i < 8; i++)
    {
        for (int j = 0; j < 8; j++)
        {
            realPosition.board[i][j] = initialBoard[i][j];
        }
    }
    realPosition.turn = 'w';
    realPosition.enPassant[0] = -1;
    realPosition.enPassant[1] = -1;

    // set up castling rights
    for (int i = 0; i < 4; i++)
    {
        realPosition.castlingRights[i] = 1;
    }

    // initialize ncurses
    if (initscr() == NULL)
    {
        return false;
    }
    if (noecho() == ERR)
    {
        endwin();
        return false;
    }
    if (raw() == ERR)
    {
        endwin();
        return false;
    }
    if (keypad(stdscr, true) == ERR)
    {
        endwin();
        return false;
    }
    top = 3;
    left = 3;

    // drawboard variables
    computer_eval = 0;
    difficulty = 4;
    highlightedDifficulty = 4;
    deselectPiece();

    // initialize color pairs
    start_color();
    init_color(COLOR_BLACK, 1000, 0, 0);
    init_pair(WHITEPAIR, COLOR_WHITE, COLOR_WHITE);
    init_pair(BLACKPAIR, COLOR_WHITE, COLOR_BLACK);
    init_pair(BLUEPAIR, COLOR_WHITE, COLOR_CYAN);
    init_pair(BOARDPAIR, COLOR_WHITE, COLOR_YELLOW);


    highlightedPiece[0] = 0;
    highlightedPiece[1] = 0;

    drawBoard(realPosition.board);

    // main loop
    while (true)
    {

        // create pointer for realPosition

        position * realPosition_ptr = &realPosition;


        // computer makes black move
        if (realPosition.turn == 'b')
        {
            ply thisPly = findBestMove(realPosition_ptr, difficulty, -999999, 999999);

            computer_eval = thisPly.evaluation;

            playMove(realPosition_ptr, thisPly.move[0], thisPly.move[1]);

        }
        else
        {
            // get player move
            int ch;
            while (realPosition.turn == 'w')
            {
                // get user's input
                refresh();
                ch = getch();

                // capitalize input to simplify cases
                ch = toupper(ch);

                switch (ch)
                {

                    // quit game
                    case 'Q':
                        endwin();
                        return 0;
                        break;

                    // skip your turn (for testing purposes)
                    case 'N':
                        realPosition.turn = 'b';
                        break;

                    // user moves the cursor with arrows
                    case KEY_UP:
                        highlightedPiece[1] += 1;
                        clearBoard();
                        drawBoard(realPosition.board);
                        break;
                    case KEY_DOWN:
                        highlightedPiece[1] -= 1;
                        clearBoard();
                        drawBoard(realPosition.board);
                        break;
                    case KEY_LEFT:
                        highlightedPiece[0] -= 1;
                        clearBoard();
                        drawBoard(realPosition.board);
                        break;
                    case KEY_RIGHT:
                        highlightedPiece[0] += 1;
                        clearBoard();
                        drawBoard(realPosition.board);
                        break;

                    // enter key, user makes a selection
                    case 10:
                        // if the user is changing the difficulty, not moving pieces
                        if (highlightedPiece[1] == 8)
                        {
                            difficulty = highlightedDifficulty;
                            break;
                        }

                        // if the user is moving pieces
                        inputPiece[0] = highlightedPiece[0];
                        inputPiece[1] = highlightedPiece[1];


                        // if player selected a valid square (square is on the board)
                        // this should always be true, but just in case

                        if (inputPiece[0] > -1 && inputPiece[0] < 8 && inputPiece[1] > -1 && inputPiece[1] < 8)
                        {

                            // if no piece is previously selected

                            if (selectedPiece[0] == -1)
                            {
                                // checks that the square is not empty

```

```c
                                if (realPosition.board[inputPiece[0]][inputPiece[1]] != ' ')
                                {
                                    selectedPiece[0] = inputPiece[0];
                                    selectedPiece[1] = inputPiece[1];
                                }
                            }
                            else
                            // a piece is already selected
                            {
                                // check if user manually deselected by selecting same piece again
                                if (!((selectedPiece[0] == inputPiece[0]) && (selectedPiece[1] == inputPiece[1])))
                                {
                                    // user did not deselect

                                    //try to move the piece
                                    if (checkLegalMove(realPosition_ptr, selectedPiece, inputPiece))
                                    {
                                        playMove(realPosition_ptr, selectedPiece, inputPiece);
                                    }
                                }
                                deselectPiece();
                            }
                        }
                        else
                        {
                            deselectPiece();
                        }
                        drawBoard(realPosition.board);
                        break;
                }
            }
        }
        clearBoard();
        drawBoard(realPosition.board);
    }
}

// recursive function to find the best move given a certain search depth
// uses minimax with alpha-beta pruning

ply findBestMove(position *position_ptr, int depth, int alpha, int beta)
{
    // thisPly contains the best move in the input position and its evaluation
    // maybe a better name is "bestmove"
    ply thisPly;

    // evaluate board if reached end of search tree

    if (depth == 0)
    {
        thisPly.evaluation = evaluate(position_ptr);
        return thisPly;
    }

    // make default evaluation low

    thisPly.move[0][0] = -1;
    thisPly.evaluation = -100000000;
    if ((*position_ptr).turn == 'b')
    {
        thisPly.evaluation = 100000000;
    }

    bool continueSearch = true;

    listAllMoves(position_ptr);

    // iterate over every possible move in input position
    for (int i = 0; i < (*position_ptr).movesLength; i ++)
    {
        if (continueSearch)
        {

            // creates new position in which a new move is played

            position newPosition = (*position_ptr);
            position * newPosition_ptr = &newPosition;
            playMove(newPosition_ptr, (*position_ptr).moves[i][0], (*position_ptr).moves[i][1]);

            // if the king was captured, that was the best choice and do not look further

            if  (abs(evaluate(newPosition_ptr)) > 10000)
            {
                continueSearch = false;
                thisPly.evaluation = evaluate(newPosition_ptr);
                thisPly.move[0][0] = (*position_ptr).moves[i][0][0];
                thisPly.move[0][1] = (*position_ptr).moves[i][0][1];
                thisPly.move[1][0] = (*position_ptr).moves[i][1][0];
                thisPly.move[1][1] = (*position_ptr).moves[i][1][1];
                (*position_ptr).bestMove = i;
            }
            else
            {

                // newPly is the best move of the new position
                ply newPly = findBestMove(newPosition_ptr, depth - 1, alpha, beta);

                if ((*position_ptr).turn == 'w')
                {
                    if (newPly.evaluation > thisPly.evaluation)
                    {
                        thisPly.evaluation = newPly.evaluation;
                        thisPly.move[0][0] = (*position_ptr).moves[i][0][0];
                        thisPly.move[0][1] = (*position_ptr).moves[i][0][1];
                        thisPly.move[1][0] = (*position_ptr).moves[i][1][0];
                        thisPly.move[1][1] = (*position_ptr).moves[i][1][1];
                        (*position_ptr).bestMove = i;

                        if (thisPly.evaluation > beta)
                        {
                            continueSearch = false;
                        }

                        alpha = thisPly.evaluation;
                    }
                }
                else
                {
                    if (newPly.evaluation < thisPly.evaluation)
                    {
                        thisPly.evaluation = newPly.evaluation;
                        thisPly.move[0][0] = (*position_ptr).moves[i][0][0];
                        thisPly.move[0][1] = (*position_ptr).moves[i][0][1];
                        thisPly.move[1][0] = (*position_ptr).moves[i][1][0];
                        thisPly.move[1][1] = (*position_ptr).moves[i][1][1];
                        (*position_ptr).bestMove = i;

                        if (thisPly.evaluation < alpha)
                        {
                            continueSearch = false;
                        }

                        beta = thisPly.evaluation;
                    }
                }
            }
        }
    }

    return thisPly;
}

int evaluate(position *position_ptr)
{
    position testPosition = *position_ptr;
    int evaluation = 0;

    // adds the material value of each piece
    // also takes into account where the piece is on the board
    // some pieces are more valuable in the center, while the king is best in the corner
    for (int i = 0; i < 8; i ++)
    {
        for (int j = 0; j < 8; j ++)
        {
            char testPiece = testPosition.board[i][j];

            if (testPiece == 'P')
            {
                evaluation += 100 + pawnPositions[7 - j][i];
            }
            else if (testPiece == 'K')
            {
                evaluation += 100000 + kingPositionsMid[7 - j][i];
            }
            else if (testPiece == 'C')
            {
                evaluation += 100000;
            }
            else if (testPiece == 'X')
            {
                evaluation += 100500 + rookPositions[7 - j][i];
            }
            else if (testPiece == 'R')
            {
                evaluation += 500 + rookPositions[7 - j][i];
            }
            else if (testPiece == 'N')
            {
                evaluation += 300 + knightPositions[7 - j][i];
```

```c
            }
            else if (testPiece == 'B')
            {
                evaluation += 310 + bishopPositions[7 - j][i];
            }
            else if (testPiece == 'Q')
            {
                evaluation += 900 + queenPositions[7 - j][i];
            }
            if (testPiece == 'p')
            {
                evaluation -= 100 + pawnPositions[j][i];
            }
            else if (testPiece == 'k')
            {
                evaluation -= 100000 + kingPositionsMid[j][i];
            }
            else if (testPiece == 'c')
            {
                evaluation -= 100000;
            }
            else if (testPiece == 'x')
            {
                evaluation -= 100500 + rookPositions[j][i];
            }
            else if (testPiece == 'r')
            {
                evaluation -= 500 + rookPositions[j][i];
            }
            else if (testPiece == 'n')
            {
                evaluation -= 300 + knightPositions[j][i];
            }
            else if (testPiece == 'b')
            {
                evaluation -= 310 + bishopPositions[j][i];
            }
            else if (testPiece == 'q')
            {
                evaluation -= 900 + queenPositions[j][i];
            }
        }
    }

    // bonus points if you can still caslte
    // discourages throwing away castling rights
    // position boost from castling should overpower this

    if (testPosition.castlingRights[0] == 1)
    {
        evaluation += 50;
    }
    if (testPosition.castlingRights[2] == 1)
    {
        evaluation -= 50;
    }

    // prevents you from castling in check and through check

    if (testPosition.castlingRights[0] == -1)
    {
        evaluation -= 200000;
    }
    if (testPosition.castlingRights[2] == -1)
    {
        evaluation += 200000;
    }

    // more possible moves = better
    // I excluded this because it makes the program a lot slower
    // although I suspect is still has merit, so I left it commented
    // just in case I want to include it later
    // "2" is just a modifier, could be between 1 and 10 or possibly even more

    // testPosition.turn = 'w';
    // listAllMoves(&testPosition);
    // evaluation += 2 * testPosition.movesLength;
    // testPosition.turn = 'b';
    // listAllMoves(&testPosition);
    // evaluation -= 2 * testPosition.movesLength;

    return evaluation;
}

void listAllMoves(position *position_ptr)
{
    (*position_ptr).movesLength = 0;
    for (int i = 0; i < 8; i ++)
    {
        for (int j = 0; j < 8; j ++)
```

```
            {
                // if the piece is the correct color
                if (getColor((*position_ptr).board[i][j]) == (*position_ptr).turn )
                {
                    piece testPiece;
                    testPiece.type = (*position_ptr).board[i][j];
                    testPiece.color = (*position_ptr).turn;
                    testPiece.coords[0] = i;
                    testPiece.coords[1] = j;
                    testPiece.listMovesLength = 0;

                    // create pointer to the piece
                    piece * piece_ptr = &testPiece;

                    if (testPiece.type == 'P' || testPiece.type == 'p')
                    {
                        listPawnMoves(position_ptr, piece_ptr);
                    }
                    else if (testPiece.type == 'K' || testPiece.type == 'k')
                    {
                        listKingMoves(position_ptr, piece_ptr);
                    }
                    else if (testPiece.type == 'R' || testPiece.type == 'r')
                    {
                        listRookMoves(position_ptr, piece_ptr);
                    }
                    else if (testPiece.type == 'N' || testPiece.type == 'n')
                    {
                        listKnightMoves(position_ptr, piece_ptr);
                    }
                    else if (testPiece.type == 'B' || testPiece.type == 'b')
                    {
                        listBishopMoves(position_ptr, piece_ptr);
                    }
                    else if (testPiece.type == 'Q' || testPiece.type == 'q')
                    {
                        listBishopMoves(position_ptr, piece_ptr);
                        listRookMoves(position_ptr, piece_ptr);
                    }

                    // transcribes the moves from the piece struct to the position struct
                    for (int k = 0; k < testPiece.listMovesLength; k ++)
                    {
                        (*position_ptr).moves[(*position_ptr).movesLength][0][0] = i;
                        (*position_ptr).moves[(*position_ptr).movesLength][0][1] = j;
                        (*position_ptr).moves[(*position_ptr).movesLength][1][0] = testPiece.listMoves[k][0];
                        (*position_ptr).moves[(*position_ptr).movesLength][1][1] = testPiece.listMoves[k][1];
                        (*position_ptr).movesLength ++;
                    }

                }
            }
        }
    }

    void clearBoard(void)
    {
        printf("\033[2J");
        printf("\033[%d;%dH", 0, 0);
    }

    void drawBoard(char board[][8])
    {
        // makes the cursor "wrap around"
        highlightedPiece[1] = (highlightedPiece[1] + 9) % 9;
        highlightedPiece[0] = (highlightedPiece[0] + 8) % 8;

        // print grid
        attron(COLOR_PAIR(BLACKPAIR));
        for (int i = 0 ; i < 8 ; i++ )
        {
            mvaddstr(top + 0 + 3 * i, left, "+-----+-----+-----+-----+-----+-----+-----+-----+");
            mvaddstr(top + 1 + 3 * i, left, "|     |     |     |     |     |     |     |     |");
            mvaddstr(top + 2 + 3 * i, left, "|     |     |     |     |     |     |     |     |");
        }
        mvaddstr(top + 8 * 3, left, "+-----+-----+-----+-----+-----+-----+-----+-----+" );
        attroff(COLOR_PAIR(BLACKPAIR));

        // print pieces
        for (int i = 0 ; i < 8 ; i++ )
        {
            for (int j = 0 ; j < 8 ; j++ )
            {
                if (board[j][i] != ' ' && board[j][i] != 'C' && board[j][i] != 'c')
                {
                    // use makeshift ascii art dictionary to print the top of the piece
                    int pieceIndex = 0;
                    for (int k = 0; k < 7; k ++)
                    {
                        if (pieces_index[k] == board[j][i] || pieces_index[k] == board[j][i] + 32)
```

```
                        {
                            pieceIndex = k;
                        }
                    }
                    mvaddstr(top + 22 - 3 * i, left + 2 + 6 * j, pieces_ascii[pieceIndex]);

                    // print the bottom of the piece
                    // all bottoms are the same
                    mvaddstr(top + 23 - 3 * i, left + 2 + 6 * j, "(_)");

                    // color to distinguish between black and white pieces
                    if (getColor(board[j][i]) == 'w')
                    {
                        attron(COLOR_PAIR(WHITEPAIR));
                        mvaddstr(top + 23 - 3 * i, left + 3 + 6 * j, "_");
                        attroff(COLOR_PAIR(WHITEPAIR));
                    }
                    else
                    {
                        attron(COLOR_PAIR(BLACKPAIR));
                        mvaddstr(top + 23 - 3 * i, left + 3 + 6 * j, "_");
                        attroff(COLOR_PAIR(BLACKPAIR));
                    }

                    // if the piece is highlighted or select it, color it accordingly
                    if (highlightedPiece[0] == j && highlightedPiece[1] == i)
                    {
                        attron(COLOR_PAIR(BLUEPAIR));
                        mvaddstr(top + 22 - 3 * i, left + 2 + 6 * j, pieces_ascii[pieceIndex]);
                        mvaddstr(top + 23 - 3 * i, left + 2 + 6 * j, "(_)");
                        attroff(COLOR_PAIR(BLUEPAIR));
                    }
                    if (selectedPiece[0] == j && selectedPiece[1] == i)
                    {
                        attron(COLOR_PAIR(WHITEPAIR));
                        mvaddstr(top + 22 - 3 * i, left + 2 + 6 * j, pieces_ascii[pieceIndex]);
                        mvaddstr(top + 23 - 3 * i, left + 2 + 6 * j, "(_)");
                        attroff(COLOR_PAIR(BLUEPAIR));
                    }
                }
                // if an empty square is highlighted
                else if (highlightedPiece[0] == j && highlightedPiece[1] == i)
                {
                    attron(COLOR_PAIR(BLUEPAIR));
                    mvaddstr(top + 22 - 3 * i, left + 2 + 6 * j, "   ");
                mvaddstr(top + 23 - 3 * i, left + 2 + 6 * j, "   ");
                attroff(COLOR_PAIR(BLUEPAIR));
            }
        }
    }

    // gives a few evaluations above the board
    char message[128] = "hello";
    sprintf(message, "Position Eval: %i \t\t Computer Eval: %i \t\t\t", evaluate(&realPosition), computer_eval);
    mvaddstr(top - 1, left, message);

    // displays difficulty level selection

    if (highlightedPiece[1] == 8)
    {
        if (highlightedPiece[0] % 3 == 0)
        {
            attron(COLOR_PAIR(BLUEPAIR));
            mvaddstr(top + 25, left + 12, "Easy");
            attroff(COLOR_PAIR(BLUEPAIR));
            highlightedDifficulty = 3;
        }
        else
        {
            attron(COLOR_PAIR(BLACKPAIR));
            mvaddstr(top + 25, left + 12, "Easy");
            attroff(COLOR_PAIR(BLACKPAIR));
        }
        mvaddstr(top + 25, left + 16, "    ");
        if (highlightedPiece[0] % 3 == 1)
        {
            attron(COLOR_PAIR(BLUEPAIR));
            mvaddstr(top + 25, left + 20, "Medium");
            attroff(COLOR_PAIR(BLUEPAIR));
            highlightedDifficulty = 4;
        }
        else
        {
            attron(COLOR_PAIR(BLACKPAIR));
            mvaddstr(top + 25, left + 20, "Medium");
            attroff(COLOR_PAIR(BLACKPAIR));
        }
        mvaddstr(top + 25, left + 26, "    ");
        if (highlightedPiece[0] % 3 == 2)
        {
```

```c
            attron(COLOR_PAIR(BLUEPAIR));
            mvaddstr(top + 25, left + 30, "Hard");
            attroff(COLOR_PAIR(BLUEPAIR));
            highlightedDifficulty = 5;
        }
        else
        {
            attron(COLOR_PAIR(BLACKPAIR));
            mvaddstr(top + 25, left + 30, "Hard");
            attroff(COLOR_PAIR(BLACKPAIR));
        }
    }
    else
    {
        attron(COLOR_PAIR(BLACKPAIR));
        mvaddstr(top + 25, left + 12, "Easy    Medium    Hard");
        attroff(COLOR_PAIR(BLACKPAIR));
    }

    // move actual cursor away
    move(0, 0);

}

void playMove(position *position_ptr, int moveFrom[], int moveTo[])
{
    char pieceType = (*position_ptr).board[moveFrom[0]][moveFrom[1]];

    // move the piece
    (*position_ptr).board[moveTo[0]][moveTo[1]] = pieceType;

    // clear the spot where the piece was
    (*position_ptr).board[moveFrom[0]][moveFrom[1]] = ' ';

    // pawn promotion
    // will only promote to queen
    if ((pieceType == 'P' && moveTo[1] == 7) || (pieceType == 'p' && moveTo[1] == 0))
    {
        (*position_ptr).board[moveTo[0]][moveTo[1]] ++;
    }


    // if captured en Passant, remove piece
    if (pieceType == 'P' || pieceType == 'p')
    {
        // set en passant square
        if (abs(moveTo[1] - moveFrom[1]) == 2)
        {
            (*position_ptr).enPassant[0] = moveFrom[0];
            (*position_ptr).enPassant[1] = (moveFrom[1] + moveTo[1]) / 2;
        }
        // check if en passant capture took place
        else if (moveTo[0] == (*position_ptr).enPassant[0] && (*position_ptr).enPassant[1] == moveTo[1])
        {
            // black pawn captured
            if (moveTo[1] == 5)
            {
                (*position_ptr).board[moveTo[0]][4] = ' ';
            }
            else
            // white pawn captured
            {
                (*position_ptr).board[moveTo[0]][3] = ' ';
            }

            // clear en passant
            (*position_ptr).enPassant[0] = -1;
        }
        else
        // clear en passant
        {
            (*position_ptr).enPassant[0] = -1;
        }
    }
    else
    {
        (*position_ptr).enPassant[0] = -1;
    }

    // if white castled on the previous move
    // reset pieces to normal
    if ((*position_ptr).castlingRights[0] == -1)
    {
        if ((*position_ptr).board[5][0] == 'X')
        {
            (*position_ptr).board[5][0] = 'R';
        }
        if ((*position_ptr).board[4][0] == 'C')
        {
            (*position_ptr).board[4][0] = ' ';
        }
        if ((*position_ptr).board[3][0] == 'X')
        {
            (*position_ptr).board[3][0] = 'R';
        }
        (*position_ptr).castlingRights[0] = 0;
    }

    // if black castled on the previous move
    // reset pieces to normal
    if ((*position_ptr).castlingRights[2] == -1)
    {
        if ((*position_ptr).board[5][7] == 'x')
        {
            (*position_ptr).board[5][7] = 'r';
        }
        if ((*position_ptr).board[4][7] == 'c')
        {
            (*position_ptr).board[4][7] = ' ';
        }
        if ((*position_ptr).board[3][7] == 'x')
        {
            (*position_ptr).board[3][7] = 'r';
        }
        (*position_ptr).castlingRights[2] = 0;
    }

    if (pieceType == 'K')
    {
        // forfeir castling rights
        (*position_ptr).castlingRights[0] = 0;
        (*position_ptr).castlingRights[1] = 0;

        // if castled kingside
        if (moveTo[0] == 6 && moveFrom[0] == 4)
        {
            (*position_ptr).board[7][0] = ' ';
            (*position_ptr).board[5][0] = 'X';
            (*position_ptr).board[4][0] = 'C';
            (*position_ptr).castlingRights[0] = -1;
        }

        // if castled queenside
        if (moveTo[0] == 2 && moveFrom[0] == 4)
        {
```

```c
            (*position_ptr).board[0][0] = ' ';
            (*position_ptr).board[3][0] = 'X';
            (*position_ptr).board[4][0] = 'C';
            (*position_ptr).castlingRights[0] = -1;
        }

    }

    if (pieceType == 'k')
    {
        // forfeit castling rights
        (*position_ptr).castlingRights[2] = 0;
        (*position_ptr).castlingRights[3] = 0;

        // if castled kingside
        if (moveTo[0] == 6 && moveFrom[0] == 4)
        {
            (*position_ptr).board[7][7] = ' ';
            (*position_ptr).board[5][7] = 'x';
            (*position_ptr).board[4][7] = 'c';
            (*position_ptr).castlingRights[2] = -1;
        }

        // if castled queenside
        if (moveTo[0] == 2 && moveFrom[0] == 4)
        {
            (*position_ptr).board[0][7] = ' ';
            (*position_ptr).board[3][7] = 'x';
            (*position_ptr).board[4][7] = 'c';
            (*position_ptr).castlingRights[2] = -1;
        }
    }

    // forfeit castling rights if you moved the appropriate rook

    if (pieceType == 'R' && moveFrom[0] == 7 && moveFrom[1] == 0)
    {
        (*position_ptr).castlingRights[0] = 0;
    }
    if (pieceType == 'R' && moveFrom[0] == 0 && moveFrom[1] == 0)
    {
        (*position_ptr).castlingRights[1] = 0;
    }
    if (pieceType == 'r' && moveFrom[0] == 7 && moveFrom[1] == 7)
    {
        (*position_ptr).castlingRights[2] = 0;
    }
    if (pieceType == 'r' && moveFrom[0] == 0 && moveFrom[1] == 7)
    {
        (*position_ptr).castlingRights[3] = 0;
    }

    // switch whose turn it is

    if ((*position_ptr).turn == 'w')
    {
        (*position_ptr).turn = 'b';
    }
    else
    {
        (*position_ptr).turn = 'w';
    }
}

bool checkLegalMove(position *position_ptr, int moveFrom[], int moveTo[])
{
    // create type piece that is the selected piece

    piece testPiece;
    testPiece.type = (*position_ptr).board[moveFrom[0]][moveFrom[1]];
    testPiece.coords[0] = moveFrom[0];
    testPiece.coords[1] = moveFrom[1];
    testPiece.listMovesLength = 0;

    // create pointer to the piece

    piece * piece_ptr = &testPiece;

    testPiece.color = getColor(testPiece.type);

    if ((*position_ptr).turn != testPiece.color)
    {
        return false;
    }

    if (testPiece.color == 'w') // white piece
    {
        switch (testPiece.type)
        {
            case 'P':

                listPawnMoves(position_ptr, piece_ptr);

                break;

            case 'N':

                listKnightMoves(position_ptr, piece_ptr);

                break;

            case 'K':

                listKingMoves(position_ptr, piece_ptr);

                break;

            case 'R':

                listRookMoves(position_ptr, piece_ptr);

                break;

            case 'B':

                listBishopMoves(position_ptr, piece_ptr);

                break;

            case 'Q':

                listRookMoves(position_ptr, piece_ptr);
                listBishopMoves(position_ptr, piece_ptr);

                break;

        }
    }
    else // black piece
    {
        switch (testPiece.type)
        {
            case 'p':

                listPawnMoves(position_ptr, piece_ptr);

                break;

            case 'n':

                listKnightMoves(position_ptr, piece_ptr);

                break;

            case 'k':

                listKingMoves(position_ptr, piece_ptr);

                break;

            case 'r':

                listRookMoves(position_ptr, piece_ptr);

                break;

            case 'b':

                listBishopMoves(position_ptr, piece_ptr);

                break;

            case 'q':

                listRookMoves(position_ptr, piece_ptr);
                listBishopMoves(position_ptr, piece_ptr);

                break;

        }
    }

    // look through generated list of moves to see if any of them is the test move

    for (int i = 0; i < testPiece.listMovesLength; i++)
    {
        if (testPiece.listMoves[i][0] == moveTo[0])
        {
            if (testPiece.listMoves[i][1] == moveTo[1])
            {
                return true;
            }
        }
    }

    return false;

}

void listPawnMoves(position *position_ptr, piece *piece_ptr)
{
    int pieceFile = (*piece_ptr).coords[0];
    int pieceRank = (*piece_ptr).coords[1];

    // black pawns move backwards
    int colorModifier = -1;

    if ((*piece_ptr).color == 'w')
    {
        colorModifier = 1;
    }

    if ((*position_ptr).board[pieceFile][pieceRank + colorModifier] == ' ')// If square in front is empty
    {
        // move possible: one forward
        append(piece_ptr, pieceFile, pieceRank + colorModifier);

        if (((7 - 5 * colorModifier) / 2) == pieceRank) // if on second rank
        {
            if ((*position_ptr).board[pieceFile][pieceRank + 2 * colorModifier] == ' ')// If square 2 in front is empty
            {
                // move possible: two forward
                append(piece_ptr, pieceFile, pieceRank + 2 * colorModifier);
            }
        }
    }
    // if enemy piece is on the diagonals

    // capture diagonally if piece there is of opposite color, and not on edge
    char testSquare;
    if (pieceFile != 7)
        {

        testSquare = (*position_ptr).board[pieceFile + 1][pieceRank + colorModifier];

        if ((getColor(testSquare) != (*piece_ptr).color) && testSquare != ' ')
        {
            append(piece_ptr, pieceFile + 1, pieceRank + colorModifier);
        }

        // capture diagonally if en passant

        if (pieceFile + 1 == (*position_ptr).enPassant[0] && pieceRank + colorModifier == (*position_ptr).enPassant[1])
        {
            append(piece_ptr, pieceFile + 1, pieceRank + colorModifier);
        }
    }
    if (pieceFile != 0)
    {
        testSquare = (*position_ptr).board[pieceFile - 1][pieceRank + colorModifier];

        if ((getColor(testSquare) != (*piece_ptr).color) && testSquare != ' ')
        {
            append(piece_ptr, pieceFile - 1, pieceRank + colorModifier);
        }

        // capture diagonally if en passant

        if (pieceFile - 1 == (*position_ptr).enPassant[0] && pieceRank + colorModifier == (*position_ptr).enPassant[1])
        {
            append(piece_ptr, pieceFile - 1, pieceRank + colorModifier);
        }
    }
}

void listKnightMoves(position *position_ptr, piece *piece_ptr)
{
    int pieceFile = (*piece_ptr).coords[0];
    int pieceRank = (*piece_ptr).coords[1];

    int knightMoves[8][2] = {{1, 2},  {2, 1},  {-1, 2},  {-2, 1},
                            {1, -2}, {2, -1}, {-1, -2}, {-2, -1}};

    for (int i = 0; i < 8; i++)
    {
        int newFile = pieceFile + knightMoves[i][0];
        int newRank = pieceRank + knightMoves[i][1];
        if (newFile >= 0 && newRank >= 0)
        {
            if (newFile < 8 && newRank < 8)
            {
                // can't be the same color piece
                if (getColor((*position_ptr).board[newFile][newRank]) != (*piece_ptr).color)
                {
                    append(piece_ptr, newFile, newRank);
                }
            }
        }
    }
}

void listKingMoves(position *position_ptr, piece *piece_ptr)
{
    int pieceFile = (*piece_ptr).coords[0];
    int pieceRank = (*piece_ptr).coords[1];

    int kingMoves[8][2] = {{1, 0},  {1, 1},  {1, -1},
                             {0, 1},  {0, -1},
                             {-1, 0}, {-1, 1}, {-1, -1}};

    for (int i = 0; i < 8; i++)
    {
        int newFile = pieceFile + kingMoves[i][0];
        int newRank = pieceRank + kingMoves[i][1];
        if (newFile >= 0 && newRank >= 0)
        {
            if (newFile < 8 && newRank < 8)
            {
                // can't be the same color piece
                if (getColor((*position_ptr).board[newFile][newRank]) != (*piece_ptr).color)
                {
                    append(piece_ptr, newFile, newRank);
                }
            }
        }
    }

    // castling

    if ((*position_ptr).castlingRights[0] == 1 && (*piece_ptr).color == 'w')
    {
        int newFile = pieceFile + 2;
        int newRank = pieceRank;
        if ((*position_ptr).board[newFile][newRank] == ' ' && (*position_ptr).board[newFile - 1][newRank] == ' ')
        {
            append(piece_ptr, newFile, newRank);
        }
    }
    if ((*position_ptr).castlingRights[1] == 1 && (*piece_ptr).color == 'w')
    {
        int newFile = pieceFile - 2;
        int newRank = pieceRank;
        if ((*position_ptr).board[newFile][newRank] == ' ' && (*position_ptr).board[newFile + 1][newRank] == ' ' && (*position_ptr).board[newFile - 1][newRank] == ' ')
        {
            append(piece_ptr, newFile, newRank);
        }
    }
    if ((*position_ptr).castlingRights[2] == 1 && (*piece_ptr).color == 'b')
    {
        int newFile = pieceFile + 2;
        int newRank = pieceRank;
        if ((*position_ptr).board[newFile][newRank] == ' ' && (*position_ptr).board[newFile - 1][newRank] == ' ')
        {
            append(piece_ptr, newFile, newRank);
        }
    }
    if ((*position_ptr).castlingRights[3] == 1 && (*piece_ptr).color == 'b')
    {
        int newFile = pieceFile - 2;
        int newRank = pieceRank;
        if ((*position_ptr).board[newFile][newRank] == ' ' && (*position_ptr).board[newFile + 1][newRank] == ' ' && (*position_ptr).board[newFile - 1][newRank] == ' ')
        {
            append(piece_ptr, newFile, newRank);
        }
    }

}

void listRookMoves(position *position_ptr, piece *piece_ptr)
{
    int pieceFile = (*piece_ptr).coords[0];
    int pieceRank = (*piece_ptr).coords[1];

```

```c
    int rookMoves[4][2] = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

    for (int i = 0; i < 4; i++)
    {
        int j = 0;
        int newFile = pieceFile;
        int newRank = pieceRank;
        while (j < 7)
        {

            newFile += rookMoves[i][0];
            newRank += rookMoves[i][1];

            if (newFile >= 0 && newRank >= 0)
            {
                if (newFile < 8 && newRank < 8)
                {
                    // can't be the same color piece
                    if (getColor((*position_ptr).board[newFile][newRank]) != (*piece_ptr).color)
                    {
                        append(piece_ptr, newFile, newRank);
                    }
                    if ((*position_ptr).board[newFile][newRank] != ' ')
                    {
                        j = 10;
                    }
                }
                else j = 10;
            }
            else j = 10;

            j++;
        }
    }
}

void listBishopMoves(position *position_ptr, piece *piece_ptr)
{
    int pieceFile = (*piece_ptr).coords[0];
    int pieceRank = (*piece_ptr).coords[1];

    int bishopMoves[4][2] = {{1, 1}, {-1, 1}, {-1, -1}, {1, -1}};

    for (int i = 0; i < 4; i++)
    {
        int j = 0;
        int newFile = pieceFile;
        int newRank = pieceRank;
        while (j < 7)
        {

            newFile += bishopMoves[i][0];
            newRank += bishopMoves[i][1];

            if (newFile >= 0 && newRank >= 0)
            {
                if (newFile < 8 && newRank < 8)
                {
                    // can't be the same color piece
                    if (getColor((*position_ptr).board[newFile][newRank]) != (*piece_ptr).color)
                    {
                        append(piece_ptr, newFile, newRank);
                    }
                    if ((*position_ptr).board[newFile][newRank] != ' ')
                    {
                        j = 10;
                    }
                }
                else j = 10;
            }
            else j = 10;

            j++;
        }
    }
}

void deselectPiece(void)
{
    selectedPiece[0] = -1;
    selectedPiece[1] = -1;
}

void append(piece *piece_ptr, int file, int rank)
{
    ((*piece_ptr).listMoves)[(*piece_ptr).listMovesLength][0] = file;
    ((*piece_ptr).listMoves)[(*piece_ptr).listMovesLength][1] = rank;
    ((*piece_ptr).listMovesLength) ++;
}

```

```c
char getColor(char testPiece)
{
    if (testPiece >= 'A' && testPiece <= 'Z')
    {
        return 'w';
    }
    if (testPiece >= 'a' && testPiece <= 'z')
    {
        return 'b';
    }
    return ' ';
}
```