```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

```

```c
    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);
```

```c
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},
```

```
136             {-40,-20,  0,  0,  0,  0,-20,-40},
137             {-30,  0, 10, 15, 15, 10,  0,-30},
138             {-30,  5, 15, 20, 20, 15,  5,-30},
139             {-30,  0, 15, 20, 20, 15,  0,-30},
140             {-30,  5, 10, 15, 15, 10,  5,-30},
141             {-40,-20,  0,  5,  5,  0,-20,-40},
142             {-50,-40,-30,-30,-30,-30,-40,-50}};
143         int bishopPositions[8][8] = {
144             {-20,-10,-10,-10,-10,-10,-10,-20},
145             {-10,  0,  0,  0,  0,  0,  0,-10},
146             {-10,  0,  5, 10, 10,  5,  0,-10},
147             {-10,  5,  5, 10, 10,  5,  5,-10},
148             {-10,  0, 10, 10, 10, 10,  0,-10},
149             {-10, 10, 10, 10, 10, 10, 10,-10},
150             {-10,  5,  0,  0,  0,  0,  5,-10},
151             {-20,-10,-10,-10,-10,-10,-10,-20}};
152         int rookPositions[8][8] = {
153             {0,  0,  0,  0,  0,  0,  0,  0},
154             {5, 10, 10, 10, 10, 10, 10,  5},
155             {-5,  0,  0,  0,  0,  0,  0, -5},
156             {-5,  0,  0,  0,  0,  0,  0, -5},
157             {-5,  0,  0,  0,  0,  0,  0, -5},
158             {-5,  0,  0,  0,  0,  0,  0, -5},
159             {-5,  0,  0,  0,  0,  0,  0, -5},
160             {0,  0,  0,  5,  5,  0,  0,  0}};
161         int queenPositions[8][8] = {
162             {-20,-10,-10, -5, -5,-10,-10,-20},
163             {-10,  0,  0,  0,  0,  0,  0,-10},
164             {-10,  0,  5,  5,  5,  5,  0,-10},
165             {-5,  0,  5,  5,  5,  5,  0, -5 },
166             {0,  0,  5,  5,  5,  5,  0, -5  },
167             {-10,  5,  5,  5,  5,  5,  0,-10},
168             {-10,  0,  5,  0,  0,  0,  0,-10},
169             {-20,-10,-10, -5, -5,-10,-10,-20}};
170         int kingPositionsMid[8][8] = {
171             {-30,-40,-40,-50,-50,-40,-40,-30},
172             {-30,-40,-40,-50,-50,-40,-40,-30},
173             {-30,-40,-40,-50,-50,-40,-40,-30},
174             {-30,-40,-40,-50,-50,-40,-40,-30},
175             {-20,-30,-30,-40,-40,-30,-30,-20},
176             {-10,-20,-20,-20,-20,-20,-20,-10},
177             {20, 20, -5, -10, -10, -5, 20,20},
178             {20, 30, 10,  0,  0, 10, 30, 20}};
179         int kingPositionsEnd[8][8] = {
180             {-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)
```

```
226         {
227             return false;
228         }
229         if (noecho() == ERR)
230         {
231             endwin();
232             return false;
233         }
234         if (raw() == ERR)
235         {
236             endwin();
237             return false;
238         }
239         if (keypad(stdscr, true) == ERR)
240         {
241             endwin();
242             return false;
243         }
244         top = 3;
245         left = 3;
246 
247         // drawboard variables
248         computer_eval = 0;
249         difficulty = 4;
250         highlightedDifficulty = 4;
251         deselectPiece();
252 
253         // initialize color pairs
254         start_color();
255         init_color(COLOR_BLACK, 1000, 0, 0);
256         init_pair(WHITEPAIR, COLOR_WHITE, COLOR_WHITE);
257         init_pair(BLACKPAIR, COLOR_WHITE, COLOR_BLACK);
258         init_pair(BLUEPAIR, COLOR_WHITE, COLOR_CYAN);
259         init_pair(BOARDPAIR, COLOR_WHITE, COLOR_YELLOW);
260 
261 
262         highlightedPiece[0] = 0;
263         highlightedPiece[1] = 0;
264 
265         drawBoard(realPosition.board);
266 
267         // main loop
268         while (true)
269         {
270 
```

```
271         // create pointer for realPosition
272
273         position * realPosition_ptr = &realPosition;
274
275
276         // computer makes black move
277         if (realPosition.turn == 'b')
278         {
279             ply thisPly = findBestMove(realPosition_ptr, difficulty, -999999, 999999);
280
281             computer_eval = thisPly.evaluation;
282
283             playMove(realPosition_ptr, thisPly.move[0], thisPly.move[1]);
284
285         }
286         else
287         {
288             // get player move
289             int ch;
290             while (realPosition.turn == 'w')
291             {
292                 // get user's input
293                 refresh();
294                 ch = getch();
295
296                 // capitalize input to simplify cases
297                 ch = toupper(ch);
298
299                 switch (ch)
300                 {
301
302                     // quit game
303                     case 'Q':
304                         endwin();
305                         return 0;
306                         break;
307
308                     // skip your turn (for testing purposes)
309                     case 'N':
310                         realPosition.turn = 'b';
311                         break;
312
313                     // user moves the cursor with arrows
314                     case KEY_UP:
315                         highlightedPiece[1] += 1;
```

```
316                         clearBoard();
317                         drawBoard(realPosition.board);
318                         break;
319                     case KEY_DOWN:
320                         highlightedPiece[1] -= 1;
321                         clearBoard();
322                         drawBoard(realPosition.board);
323                         break;
324                     case KEY_LEFT:
325                         highlightedPiece[0] -= 1;
326                         clearBoard();
327                         drawBoard(realPosition.board);
328                         break;
329                     case KEY_RIGHT:
330                         highlightedPiece[0] += 1;
331                         clearBoard();
332                         drawBoard(realPosition.board);
333                         break;
334 
335                     // enter key, user makes a selection
336                     case 10:
337                         // if the user is changing the difficulty, not moving pieces
338                         if (highlightedPiece[1] == 8)
339                         {
340                             difficulty = highlightedDifficulty;
341                             break;
342                         }
343 
344                         // if the user is moving pieces
345                         inputPiece[0] = highlightedPiece[0];
346                         inputPiece[1] = highlightedPiece[1];
347 
348 
349                         // if player selected a valid square (square is on the board)
350                         // this should always be true, but just in case
351 
352                         if (inputPiece[0] > -1 && inputPiece[0] < 8 && inputPiece[1] > -1 && inputPiece[1] < 8)
353                         {
354 
355                             // if no piece is previously selected
356 
357                             if (selectedPiece[0] == -1)
358                             {
359                                 // checks that the square is not empty
360 
```

```
                                    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"
```

```
405       ply thisPly;
406
407       // evaluate board if reached end of search tree
408
409       if (depth == 0)
410       {
411           thisPly.evaluation = evaluate(position_ptr);
412           return thisPly;
413       }
414
415       // make default evaluation low
416
417       thisPly.move[0][0] = -1;
418       thisPly.evaluation = -100000000;
419       if ((*position_ptr).turn == 'b')
420       {
421           thisPly.evaluation = 100000000;
422       }
423
424       bool continueSearch = true;
425
426       listAllMoves(position_ptr);
427
428       // iterate over every possible move in input position
429       for (int i = 0; i < (*position_ptr).movesLength; i ++)
430       {
431           if (continueSearch)
432           {
433
434               // creates new position in which a new move is played
435
436               position newPosition = (*position_ptr);
437               position * newPosition_ptr = &newPosition;
438               playMove(newPosition_ptr, (*position_ptr).moves[i][0], (*position_ptr).moves[i][1]);
439
440               // if the king was captured, that was the best choice and do not look further
441
442               if  (abs(evaluate(newPosition_ptr)) > 10000)
443               {
444                   continueSearch = false;
445                   thisPly.evaluation = evaluate(newPosition_ptr);
446                   thisPly.move[0][0] = (*position_ptr).moves[i][0][0];
447                   thisPly.move[0][1] = (*position_ptr).moves[i][0][1];
448                   thisPly.move[1][0] = (*position_ptr).moves[i][1][0];
449                   thisPly.move[1][1] = (*position_ptr).moves[i][1][1];
```

```
450                 (*position_ptr).bestMove = i;
451             }
452             else
453             {
454 
455                 // newPly is the best move of the new position
456                 ply newPly = findBestMove(newPosition_ptr, depth - 1, alpha, beta);
457 
458                 if ((*position_ptr).turn == 'w')
459                 {
460                     if (newPly.evaluation > thisPly.evaluation)
461                     {
462                         thisPly.evaluation = newPly.evaluation;
463                         thisPly.move[0][0] = (*position_ptr).moves[i][0][0];
464                         thisPly.move[0][1] = (*position_ptr).moves[i][0][1];
465                         thisPly.move[1][0] = (*position_ptr).moves[i][1][0];
466                         thisPly.move[1][1] = (*position_ptr).moves[i][1][1];
467                         (*position_ptr).bestMove = i;
468 
469                         if (thisPly.evaluation > beta)
470                         {
471                             continueSearch = false;
472                         }
473 
474                         alpha = thisPly.evaluation;
475                     }
476                 }
477                 else
478                 {
479                     if (newPly.evaluation < thisPly.evaluation)
480                     {
481                         thisPly.evaluation = newPly.evaluation;
482                         thisPly.move[0][0] = (*position_ptr).moves[i][0][0];
483                         thisPly.move[0][1] = (*position_ptr).moves[i][0][1];
484                         thisPly.move[1][0] = (*position_ptr).moves[i][1][0];
485                         thisPly.move[1][1] = (*position_ptr).moves[i][1][1];
486                         (*position_ptr).bestMove = i;
487 
488                         if (thisPly.evaluation < alpha)
489                         {
490                             continueSearch = false;
491                         }
492 
493                         beta = thisPly.evaluation;
494                     }
```

```
                }
            }
        }
    }

    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];
```

```
            }
            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
```

```
585         // discourages throwing away castling rights
586         // position boost from castling should overpower this
587 
588         if (testPosition.castlingRights[0] == 1)
589         {
590             evaluation += 50;
591         }
592         if (testPosition.castlingRights[2] == 1)
593         {
594             evaluation -= 50;
595         }
596 
597         // prevents you from castling in check and through check
598 
599         if (testPosition.castlingRights[0] == -1)
600         {
601             evaluation -= 200000;
602         }
603         if (testPosition.castlingRights[2] == -1)
604         {
605             evaluation += 200000;
606         }
607 
608         // more possible moves = better
609         // I excluded this because it makes the program a lot slower
610         // although I suspect is still has merit, so I left it commented
611         // just in case I want to include it later
612         // "2" is just a modifier, could be between 1 and 10 or possibly even more
613 
614         // testPosition.turn = 'w';
615         // listAllMoves(&testPosition);
616         // evaluation += 2 * testPosition.movesLength;
617         // testPosition.turn = 'b';
618         // listAllMoves(&testPosition);
619         // evaluation -= 2 * testPosition.movesLength;
620 
621         return evaluation;
622     }
623 
624     void listAllMoves(position *position_ptr)
625     {
626         (*position_ptr).movesLength = 0;
627         for (int i = 0; i < 8; i ++)
628         {
629             for (int j = 0; j < 8; j ++)
```

```
630         {
631             // if the piece is the correct color
632             if (getColor((*position_ptr).board[i][j]) == (*position_ptr).turn )
633             {
634                 piece testPiece;
635                 testPiece.type = (*position_ptr).board[i][j];
636                 testPiece.color = (*position_ptr).turn;
637                 testPiece.coords[0] = i;
638                 testPiece.coords[1] = j;
639                 testPiece.listMovesLength = 0;
640 
641                 // create pointer to the piece
642                 piece * piece_ptr = &testPiece;
643 
644                 if (testPiece.type == 'P' || testPiece.type == 'p')
645                 {
646                     listPawnMoves(position_ptr, piece_ptr);
647                 }
648                 else if (testPiece.type == 'K' || testPiece.type == 'k')
649                 {
650                     listKingMoves(position_ptr, piece_ptr);
651                 }
652                 else if (testPiece.type == 'R' || testPiece.type == 'r')
653                 {
654                     listRookMoves(position_ptr, piece_ptr);
655                 }
656                 else if (testPiece.type == 'N' || testPiece.type == 'n')
657                 {
658                     listKnightMoves(position_ptr, piece_ptr);
659                 }
660                 else if (testPiece.type == 'B' || testPiece.type == 'b')
661                 {
662                     listBishopMoves(position_ptr, piece_ptr);
663                 }
664                 else if (testPiece.type == 'Q' || testPiece.type == 'q')
665                 {
666                     listBishopMoves(position_ptr, piece_ptr);
667                     listRookMoves(position_ptr, piece_ptr);
668                 }
669 
670                 // transcribes the moves from the piece struct to the position struct
671                 for (int k = 0; k < testPiece.listMovesLength; k ++)
672                 {
673                     (*position_ptr).moves[(*position_ptr).movesLength][0][0] = i;
674                     (*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)
```

```
720                     {
721                         pieceIndex = k;
722                     }
723                 }
724                 mvaddstr(top + 22 - 3 * i, left + 2 + 6 * j, pieces_ascii[pieceIndex]);
725 
726                 // print the bottom of the piece
727                 // all bottoms are the same
728                 mvaddstr(top + 23 - 3 * i, left + 2 + 6 * j, "(_)");
729 
730                 // color to distinguish between black and white pieces
731                 if (getColor(board[j][i]) == 'w')
732                 {
733                     attron(COLOR_PAIR(WHITEPAIR));
734                     mvaddstr(top + 23 - 3 * i, left + 3 + 6 * j, "_");
735                     attroff(COLOR_PAIR(WHITEPAIR));
736                 }
737                 else
738                 {
739                     attron(COLOR_PAIR(BLACKPAIR));
740                     mvaddstr(top + 23 - 3 * i, left + 3 + 6 * j, "_");
741                     attroff(COLOR_PAIR(BLACKPAIR));
742                 }
743 
744                 // if the piece is highlighted or select it, color it accordingly
745                 if (highlightedPiece[0] == j && highlightedPiece[1] == i)
746                 {
747                     attron(COLOR_PAIR(BLUEPAIR));
748                     mvaddstr(top + 22 - 3 * i, left + 2 + 6 * j, pieces_ascii[pieceIndex]);
749                     mvaddstr(top + 23 - 3 * i, left + 2 + 6 * j, "(_)");
750                     attroff(COLOR_PAIR(BLUEPAIR));
751                 }
752                 if (selectedPiece[0] == j && selectedPiece[1] == i)
753                 {
754                     attron(COLOR_PAIR(WHITEPAIR));
755                     mvaddstr(top + 22 - 3 * i, left + 2 + 6 * j, pieces_ascii[pieceIndex]);
756                     mvaddstr(top + 23 - 3 * i, left + 2 + 6 * j, "(_)");
757                     attroff(COLOR_PAIR(BLUEPAIR));
758                 }
759             }
760             // if an empty square is highlighted
761             else if (highlightedPiece[0] == j && highlightedPiece[1] == i)
762             {
763                 attron(COLOR_PAIR(BLUEPAIR));
764                 mvaddstr(top + 22 - 3 * i, left + 2 + 6 * j, "   ");
```

```
765                 mvaddstr(top + 23 - 3 * i, left + 2 + 6 * j, "    ");
766                 attroff(COLOR_PAIR(BLUEPAIR));
767             }
768         }
769     }
770 
771     // gives a few evaluations above the board
772     char message[128] = "hello";
773     sprintf(message, "Position Eval: %i \t\t Computer Eval: %i \t\t\t", evaluate(&realPosition), computer_eval);
774     mvaddstr(top - 1, left, message);
775 
776     // displays difficulty level selection
777 
778     if (highlightedPiece[1] == 8)
779     {
780         if (highlightedPiece[0] % 3 == 0)
781         {
782             attron(COLOR_PAIR(BLUEPAIR));
783             mvaddstr(top + 25, left + 12, "Easy");
784             attroff(COLOR_PAIR(BLUEPAIR));
785             highlightedDifficulty = 3;
786         }
787         else
788         {
789             attron(COLOR_PAIR(BLACKPAIR));
790             mvaddstr(top + 25, left + 12, "Easy");
791             attroff(COLOR_PAIR(BLACKPAIR));
792         }
793         mvaddstr(top + 25, left + 16, "     ");
794         if (highlightedPiece[0] % 3 == 1)
795         {
796             attron(COLOR_PAIR(BLUEPAIR));
797             mvaddstr(top + 25, left + 20, "Medium");
798             attroff(COLOR_PAIR(BLUEPAIR));
799             highlightedDifficulty = 4;
800         }
801         else
802         {
803             attron(COLOR_PAIR(BLACKPAIR));
804             mvaddstr(top + 25, left + 20, "Medium");
805             attroff(COLOR_PAIR(BLACKPAIR));
806         }
807         mvaddstr(top + 25, left + 26, "     ");
808         if (highlightedPiece[0] % 3 == 2)
809         {
```

```
810                 attron(COLOR_PAIR(BLUEPAIR));
811                 mvaddstr(top + 25, left + 30, "Hard");
812                 attroff(COLOR_PAIR(BLUEPAIR));
813                 highlightedDifficulty = 5;
814             }
815             else
816             {
817                 attron(COLOR_PAIR(BLACKPAIR));
818                 mvaddstr(top + 25, left + 30, "Hard");
819                 attroff(COLOR_PAIR(BLACKPAIR));
820             }
821         }
822         else
823         {
824             attron(COLOR_PAIR(BLACKPAIR));
825             mvaddstr(top + 25, left + 12, "Easy    Medium    Hard");
826             attroff(COLOR_PAIR(BLACKPAIR));
827         }
828 
829         // move actual cursor away
830         move(0, 0);
831 
832     }
833 
834     void playMove(position *position_ptr, int moveFrom[], int moveTo[])
835     {
836         char pieceType = (*position_ptr).board[moveFrom[0]][moveFrom[1]];
837 
838         // move the piece
839         (*position_ptr).board[moveTo[0]][moveTo[1]] = pieceType;
840 
841         // clear the spot where the piece was
842         (*position_ptr).board[moveFrom[0]][moveFrom[1]] = ' ';
843 
844         // pawn promotion
845         // will only promote to queen
846         if ((pieceType == 'P' && moveTo[1] == 7) || (pieceType == 'p' && moveTo[1] == 0))
847         {
848             (*position_ptr).board[moveTo[0]][moveTo[1]] ++;
849         }
850 
851 
852         // if captured en Passant, remove piece
853         if (pieceType == 'P' || pieceType == 'p')
854         {
```

```
855             // set en passant square
856             if (abs(moveTo[1] - moveFrom[1]) == 2)
857             {
858                 (*position_ptr).enPassant[0] = moveFrom[0];
859                 (*position_ptr).enPassant[1] = (moveFrom[1] + moveTo[1]) / 2;
860             }
861             // check if en passant capture took place
862             else if (moveTo[0] == (*position_ptr).enPassant[0] && (*position_ptr).enPassant[1] == moveTo[1])
863             {
864                 // black pawn captured
865                 if (moveTo[1] == 5)
866                 {
867                     (*position_ptr).board[moveTo[0]][4] = ' ';
868                 }
869                 else
870                 // white pawn captured
871                 {
872                     (*position_ptr).board[moveTo[0]][3] = ' ';
873                 }
874 
875                 // clear en passant
876                 (*position_ptr).enPassant[0] = -1;
877             }
878             else
879             // clear en passant
880             {
881                 (*position_ptr).enPassant[0] = -1;
882             }
883         }
884         else
885         {
886             (*position_ptr).enPassant[0] = -1;
887         }
888 
889         // if white castled on the previous move
890         // reset pieces to normal
891         if ((*position_ptr).castlingRights[0] == -1)
892         {
893             if ((*position_ptr).board[5][0] == 'X')
894             {
895                 (*position_ptr).board[5][0] = 'R';
896             }
897             if ((*position_ptr).board[4][0] == 'C')
898             {
899                 (*position_ptr).board[4][0] = ' ';
```

```
900             }
901             if ((*position_ptr).board[3][0] == 'X')
902             {
903                 (*position_ptr).board[3][0] = 'R';
904             }
905             (*position_ptr).castlingRights[0] = 0;
906         }
907 
908         // if black castled on the previous move
909         // reset pieces to normal
910         if ((*position_ptr).castlingRights[2] == -1)
911         {
912             if ((*position_ptr).board[5][7] == 'x')
913             {
914                 (*position_ptr).board[5][7] = 'r';
915             }
916             if ((*position_ptr).board[4][7] == 'c')
917             {
918                 (*position_ptr).board[4][7] = ' ';
919             }
920             if ((*position_ptr).board[3][7] == 'x')
921             {
922                 (*position_ptr).board[3][7] = 'r';
923             }
924             (*position_ptr).castlingRights[2] = 0;
925         }
926 
927         if (pieceType == 'K')
928         {
929             // forfeir castling rights
930             (*position_ptr).castlingRights[0] = 0;
931             (*position_ptr).castlingRights[1] = 0;
932 
933             // if castled kingside
934             if (moveTo[0] == 6 && moveFrom[0] == 4)
935             {
936                 (*position_ptr).board[7][0] = ' ';
937                 (*position_ptr).board[5][0] = 'X';
938                 (*position_ptr).board[4][0] = 'C';
939                 (*position_ptr).castlingRights[0] = -1;
940             }
941 
942             // if castled queenside
943             if (moveTo[0] == 2 && moveFrom[0] == 4)
944             {
```

```
945             (*position_ptr).board[0][0] = ' ';
946             (*position_ptr).board[3][0] = 'X';
947             (*position_ptr).board[4][0] = 'C';
948             (*position_ptr).castlingRights[0] = -1;
949         }
950 
951     }
952 
953     if (pieceType == 'k')
954     {
955         // forfeit castling rights
956         (*position_ptr).castlingRights[2] = 0;
957         (*position_ptr).castlingRights[3] = 0;
958 
959         // if castled kingside
960         if (moveTo[0] == 6 && moveFrom[0] == 4)
961         {
962             (*position_ptr).board[7][7] = ' ';
963             (*position_ptr).board[5][7] = 'x';
964             (*position_ptr).board[4][7] = 'c';
965             (*position_ptr).castlingRights[2] = -1;
966         }
967 
968         // if castled queenside
969         if (moveTo[0] == 2 && moveFrom[0] == 4)
970         {
971             (*position_ptr).board[0][7] = ' ';
972             (*position_ptr).board[3][7] = 'x';
973             (*position_ptr).board[4][7] = 'c';
974             (*position_ptr).castlingRights[2] = -1;
975         }
976     }
977 
978     // forfeit castling rights if you moved the appropriate rook
979 
980     if (pieceType == 'R' && moveFrom[0] == 7 && moveFrom[1] == 0)
981     {
982         (*position_ptr).castlingRights[0] = 0;
983     }
984     if (pieceType == 'R' && moveFrom[0] == 0 && moveFrom[1] == 0)
985     {
986         (*position_ptr).castlingRights[1] = 0;
987     }
988     if (pieceType == 'r' && moveFrom[0] == 7 && moveFrom[1] == 7)
989     {
```

```
990             (*position_ptr).castlingRights[2] = 0;
991         }
992         if (pieceType == 'r' && moveFrom[0] == 0 && moveFrom[1] == 7)
993         {
994             (*position_ptr).castlingRights[3] = 0;
995         }
996 
997         // switch whose turn it is
998 
999         if ((*position_ptr).turn == 'w')
1000        {
1001            (*position_ptr).turn = 'b';
1002        }
1003        else
1004        {
1005            (*position_ptr).turn = 'w';
1006        }
1007    }
1008
1009    bool checkLegalMove(position *position_ptr, int moveFrom[], int moveTo[])
1010    {
1011        // create type piece that is the selected piece
1012
1013        piece testPiece;
1014        testPiece.type = (*position_ptr).board[moveFrom[0]][moveFrom[1]];
1015        testPiece.coords[0] = moveFrom[0];
1016        testPiece.coords[1] = moveFrom[1];
1017        testPiece.listMovesLength = 0;
1018
1019        // create pointer to the piece
1020
1021        piece * piece_ptr = &testPiece;
1022
1023        testPiece.color = getColor(testPiece.type);
1024
1025        if ((*position_ptr).turn != testPiece.color)
1026        {
1027            return false;
1028        }
1029
1030        if (testPiece.color == 'w') // white piece
1031        {
1032            switch (testPiece.type)
1033            {
1034                case 'P':
```

```
1035
1036                    listPawnMoves(position_ptr, piece_ptr);
1037
1038                    break;
1039
1040                case 'N':
1041
1042                    listKnightMoves(position_ptr, piece_ptr);
1043
1044                    break;
1045
1046                case 'K':
1047
1048                    listKingMoves(position_ptr, piece_ptr);
1049
1050                    break;
1051
1052                case 'R':
1053
1054                    listRookMoves(position_ptr, piece_ptr);
1055
1056                    break;
1057
1058                case 'B':
1059
1060                    listBishopMoves(position_ptr, piece_ptr);
1061
1062                    break;
1063
1064                case 'Q':
1065
1066                    listRookMoves(position_ptr, piece_ptr);
1067                    listBishopMoves(position_ptr, piece_ptr);
1068
1069                    break;
1070
1071            }
1072        }
1073        else // black piece
1074        {
1075            switch (testPiece.type)
1076            {
1077                case 'p':
1078
1079                    listPawnMoves(position_ptr, piece_ptr);
```

```
1080
1081                    break;
1082
1083                case 'n':
1084
1085                    listKnightMoves(position_ptr, piece_ptr);
1086
1087                    break;
1088
1089                case 'k':
1090
1091                    listKingMoves(position_ptr, piece_ptr);
1092
1093                    break;
1094
1095                case 'r':
1096
1097                    listRookMoves(position_ptr, piece_ptr);
1098
1099                    break;
1100
1101                case 'b':
1102
1103                    listBishopMoves(position_ptr, piece_ptr);
1104
1105                    break;
1106
1107                case 'q':
1108
1109                    listRookMoves(position_ptr, piece_ptr);
1110                    listBishopMoves(position_ptr, piece_ptr);
1111
1112                    break;
1113
1114            }
1115        }
1116
1117        // look through generated list of moves to see if any of them is the test move
1118
1119        for (int i = 0; i < testPiece.listMovesLength; i++)
1120        {
1121            if (testPiece.listMoves[i][0] == moveTo[0])
1122            {
1123                if (testPiece.listMoves[i][1] == moveTo[1])
1124                {
```

```
1125                    return true;
1126                }
1127            }
1128        }
1129
1130        return false;
1131
1132    }
1133
1134    void listPawnMoves(position *position_ptr, piece *piece_ptr)
1135    {
1136        int pieceFile = (*piece_ptr).coords[0];
1137        int pieceRank = (*piece_ptr).coords[1];
1138
1139        // black pawns move backwards
1140        int colorModifier = -1;
1141
1142        if ((*piece_ptr).color == 'w')
1143        {
1144            colorModifier = 1;
1145        }
1146
1147        if ((*position_ptr).board[pieceFile][pieceRank + colorModifier] == ' ')// If square in front is empty
1148        {
1149            // move possible: one forward
1150            append(piece_ptr, pieceFile, pieceRank + colorModifier);
1151
1152            if (((7 - 5 * colorModifier) / 2) == pieceRank) // if on second rank
1153            {
1154                if ((*position_ptr).board[pieceFile][pieceRank + 2 * colorModifier] == ' ')// If square 2 in front
is empty
1155                {
1156                    // move possible: two forward
1157                    append(piece_ptr, pieceFile, pieceRank + 2 * colorModifier);
1158                }
1159            }
1160        }
1161        // if enemy piece is on the diagonals
1162
1163        // capture diagonally if piece there is of opposite color, and not on edge
1164        char testSquare;
1165        if (pieceFile != 7)
1166            {
1167
1168            testSquare = (*position_ptr).board[pieceFile + 1][pieceRank + colorModifier];
```

```
1169
1170            if ((getColor(testSquare) != (*piece_ptr).color) && testSquare != ' ')
1171            {
1172                append(piece_ptr, pieceFile + 1, pieceRank + colorModifier);
1173            }
1174
1175            // capture diagonally if en passant
1176
1177            if (pieceFile + 1 == (*position_ptr).enPassant[0] && pieceRank + colorModifier == (*position_ptr).enPass
ant[1])
1178            {
1179                append(piece_ptr, pieceFile + 1, pieceRank + colorModifier);
1180            }
1181        }
1182        if (pieceFile != 0)
1183        {
1184            testSquare = (*position_ptr).board[pieceFile - 1][pieceRank + colorModifier];
1185
1186            if ((getColor(testSquare) != (*piece_ptr).color) && testSquare != ' ')
1187            {
1188                append(piece_ptr, pieceFile - 1, pieceRank + colorModifier);
1189            }
1190
1191            // capture diagonally if en passant
1192
1193            if (pieceFile - 1 == (*position_ptr).enPassant[0] && pieceRank + colorModifier == (*position_ptr).enPass
ant[1])
1194            {
1195                append(piece_ptr, pieceFile - 1, pieceRank + colorModifier);
1196            }
1197        }
1198    }
1199
1200    void listKnightMoves(position *position_ptr, piece *piece_ptr)
1201    {
1202        int pieceFile = (*piece_ptr).coords[0];
1203        int pieceRank = (*piece_ptr).coords[1];
1204
1205        int knightMoves[8][2] = {{1, 2},  {2, 1},  {-1, 2},  {-2, 1},
1206                                 {1, -2}, {2, -1}, {-1, -2}, {-2, -1}};
1207
1208        for (int i = 0; i < 8; i++)
1209        {
1210            int newFile = pieceFile + knightMoves[i][0];
1211            int newRank = pieceRank + knightMoves[i][1];
```

```c
            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;
```

```
1257            int newRank = pieceRank;
1258            if ((*position_ptr).board[newFile][newRank] == ' ' && (*position_ptr).board[newFile - 1][newRank] == '
')
1259            {
1260                append(piece_ptr, newFile, newRank);
1261            }
1262        }
1263        if ((*position_ptr).castlingRights[1] == 1 && (*piece_ptr).color == 'w')
1264        {
1265            int newFile = pieceFile - 2;
1266            int newRank = pieceRank;
1267            if ((*position_ptr).board[newFile][newRank] == ' ' && (*position_ptr).board[newFile + 1][newRank] == ' '
&& (*position_ptr).board[newFile - 1][newRank] == ' ')
1268            {
1269                append(piece_ptr, newFile, newRank);
1270            }
1271        }
1272        if ((*position_ptr).castlingRights[2] == 1 && (*piece_ptr).color == 'b')
1273        {
1274            int newFile = pieceFile + 2;
1275            int newRank = pieceRank;
1276            if ((*position_ptr).board[newFile][newRank] == ' ' && (*position_ptr).board[newFile - 1][newRank] == '
')
1277            {
1278                append(piece_ptr, newFile, newRank);
1279            }
1280        }
1281        if ((*position_ptr).castlingRights[3] == 1 && (*piece_ptr).color == 'b')
1282        {
1283            int newFile = pieceFile - 2;
1284            int newRank = pieceRank;
1285            if ((*position_ptr).board[newFile][newRank] == ' ' && (*position_ptr).board[newFile + 1][newRank] == ' '
&& (*position_ptr).board[newFile - 1][newRank] == ' ')
1286            {
1287                append(piece_ptr, newFile, newRank);
1288            }
1289        }
1290
1291    }
1292
1293    void listRookMoves(position *position_ptr, piece *piece_ptr)
1294    {
1295        int pieceFile = (*piece_ptr).coords[0];
1296        int pieceRank = (*piece_ptr).coords[1];
1297
```

```
1298        int rookMoves[4][2] = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};
1299
1300        for (int i = 0; i < 4; i++)
1301        {
1302            int j = 0;
1303            int newFile = pieceFile;
1304            int newRank = pieceRank;
1305            while (j < 7)
1306            {
1307
1308                newFile += rookMoves[i][0];
1309                newRank += rookMoves[i][1];
1310
1311                if (newFile >= 0 && newRank >= 0)
1312                {
1313                    if (newFile < 8 && newRank < 8)
1314                    {
1315                        // can't be the same color piece
1316                        if (getColor((*position_ptr).board[newFile][newRank]) != (*piece_ptr).color)
1317                        {
1318                            append(piece_ptr, newFile, newRank);
1319                        }
1320                        if ((*position_ptr).board[newFile][newRank] != ' ')
1321                        {
1322                            j = 10;
1323                        }
1324                    }
1325                    else j = 10;
1326                }
1327                else j = 10;
1328
1329                j++;
1330            }
1331        }
1332    }
1333
1334    void listBishopMoves(position *position_ptr, piece *piece_ptr)
1335    {
1336        int pieceFile = (*piece_ptr).coords[0];
1337        int pieceRank = (*piece_ptr).coords[1];
1338
1339        int bishopMoves[4][2] = {{1, 1}, {-1, 1}, {-1, -1}, {1, -1}};
1340
1341        for (int i = 0; i < 4; i++)
1342        {
```

```
1343            int j = 0;
1344            int newFile = pieceFile;
1345            int newRank = pieceRank;
1346            while (j < 7)
1347            {
1348
1349                newFile += bishopMoves[i][0];
1350                newRank += bishopMoves[i][1];
1351
1352                if (newFile >= 0 && newRank >= 0)
1353                {
1354                    if (newFile < 8 && newRank < 8)
1355                    {
1356                        // can't be the same color piece
1357                        if (getColor((*position_ptr).board[newFile][newRank]) != (*piece_ptr).color)
1358                        {
1359                            append(piece_ptr, newFile, newRank);
1360                        }
1361                        if ((*position_ptr).board[newFile][newRank] != ' ')
1362                        {
1363                            j = 10;
1364                        }
1365                    }
1366                    else j = 10;
1367                }
1368                else j = 10;
1369
1370                j++;
1371            }
1372        }
1373    }
1374
1375    void deselectPiece(void)
1376    {
1377        selectedPiece[0] = -1;
1378        selectedPiece[1] = -1;
1379    }
1380
1381    void append(piece *piece_ptr, int file, int rank)
1382    {
1383        ((*piece_ptr).listMoves)[(*piece_ptr).listMovesLength][0] = file;
1384        ((*piece_ptr).listMoves)[(*piece_ptr).listMovesLength][1] = rank;
1385        ((*piece_ptr).listMovesLength) ++;
1386    }
1387
```

```
1388  char getColor(char testPiece)
1389  {
1390      if (testPiece >= 'A' && testPiece <= 'Z')
1391      {
1392          return 'w';
1393      }
1394      if (testPiece >= 'a' && testPiece <= 'z')
1395      {
1396          return 'b';
1397      }
1398      return ' ';
1399  }
```