**Project 1**

**<Chess>**

**CSC-17a**

**Name: Mathew Briguglio**

**Date: 05/04/15**

**Introduction:**

Title: Chess

Chess is a strategy based game played with two players on a checkered game board. Each player has eight chess pieces: rook, knight, bishop, queen, king, and pawn. Each piece has moves on the chessboard according to a set of rules.

For example: The rook can move horizontally or vertically any amount of squares. However, it cannot leap over any other pieces, but it may capture, that is, replace the position of an opponent’s piece.

The chess game is over when one of the pieces, the king, doesn’t have any moves remaining due to the fact that if it did move to another square an opponent’s piece could capture it.

Chess is a complex game with many different outcomes, and it requires patience and strategizing. It can help strengthen logical thinking, and can be enjoyed by both children and adults.

**Summary:**

Project size: About 1860 lines

The number of variables: about 40

This project includes many concepts from chapters 1-12 in the Gaddis textbook. Some of the concepts involved include: variable declaration, primitive data types such as integers and characters, abstract data types or structures, looping constructs such as while, do while, and for loop, file operations, 1D arrays, 2D arrays, pointers to different variables, dynamic allocation of memory to both 1D and 2D arrays, character arrays, functions, function arguments/parameters, return statements, and more.

This project took about 2-3 weeks of time. A majority of the difficulty was in the logic behind a game of chess. The programming constructs were relatively simple to implement – structures, loops, variable assignments and initializations, pointers and dynamic allocation of memory, character and integer arrays, character testing and conversion, input validation, binary files, etc.

Coding a game is an amazing experience because it reveals the difficulty behind implementing logic via computation. The code has room for plenty of improvements, details that could make the game better for the user. However, the code runs and acts as a decent simulation for the game of chess.

**Description:**

Chess is a strategy-based board game played on a checkered game board. The board is an eight by eight grid thus it contains 64 squares. Chess is a two player game, and each player begins with 16 pieces: two rooks two knights, two bishops, one queen, one king, and eight pawns. Each game piece moves in a unique way. If player one moves one of their game pieces and it happens to land on a square occupied by their opponent's game piece, then player one captures player two's game piece and that captured piece is permanently removed from the board. Player one's game piece now occupies the square and remains on the board.) The only piece that cannot be captured is the king. The objective of the game is to checkmate the opponent's king, meaning that any move the opponent's king makes, it would be threatened by inevitable capture. If the opponent's king is still able to move to a square where it cannot be captured, then the game continues! Players also earn points by capturing the opponent's game pieces.

MOVING THE PIECES:

Rook: The rook can move any number of squares horizontally or vertically, but it cannot leap over any other pieces.

Knight: The knight can move two squares horizontally and then one square vertically OR two squares vertically and then one square horizontally. Unlike the other pieces, the knight can leap over any other piece and land on its destination square. The movement of the knight is often referred to as an L-shape.

Bishop: The bishop can move any number of squares diagonally, but it cannot leap over any other pieces.

Queen: The queen can move any number of squares horizontally, vertically, or diagonally, but it cannot leap over any other pieces.

King: The king can move one square horizontally, vertically, or diagonally.

Pawn: The pawn can move forward one square or, if it is the pawn's first move, it can move forward two squares. The pawn can also move diagonally forward one square only when capturing another piece. Lastly, if a pawn reaches the final row of the board on the opponent's side, then the pawn is promoted to the player's choice of either a rook, knight, bishop, or queen.

SPECIAL MOVES:

Castling: If one of the rooks and the king have not yet moved from their original positions, then they may castle. The king moves two squares toward the rook the player chooses, and the rook moves to the adjacent square on the other side of the king.

En Passant: If player 2 moves their pawn forward two spaces on the pawn's first move and one of player 1's pawns happens to be to the left or right (on an adjacent square) then player 1 may capture player 2's pawn en passant (in passing) by moving to the square immediately behind player 2's pawn.

Pawn Promotion: If a pawn makes it to the eighth row of the chessboard - on the opponent's side - they may promote their pawn. That is, the player will be given the choice of replacing their pawn with either a knight, bishop, rook, or queen.

NOTATION KEY FOR THE GAME PIECES:

Rook: R or r

Knight: N or n

Bishop: B or b

Queen: Q or q

King: K or k

Pawn: P or a

Black Pieces: uppercase letters

White Pieces: lowercase letters

ASSIGNMENT OF POINT VALUES:

Pawn: 1

Knight: 3

Bishop: 3

Rook: 5

Queen: 9

A FEW EXTRA RULES:

White always has the first move.

There are no restrictions on pawn promotions; the player may choose whichever piece they prefer each time.

Players may not skip turns.

To capture an opponent's piece, you move your own piece to an appropriate square containing one of the opponent's pieces.

You then permanently remove their piece from the game board, and your piece remains on the square.

The game ends when either a player resigns, there is a draw, or there is a checkmate.

- Draw: It is a draw when neither player can win the game.

- Resignation: A player may resign from (or leave/quit) the game if they believe they will lose.

- Checkmate: A checkmate occurs when one player's king cannot move to a new square without being threatened by inevitable capture.

**Pseudo Code:**

*Show game menu.*

*Prompt user with switch statement to choose new game, load game, game description, or exit.*

*Input validation for user input option.*

*Game will call resetBoard function to place pieces on board in original positions.*

*If piece is 1 or black, uppercase letters will be at lower end of board.*

*Else if piece is 2 or white, lowercase letters will be at lower end of board.*

*If new game, then call function to have player enter name.*

*Have player choose chess piece color, black or white.*

*Ask player if they’d like to save game.*

*Enter do while loop to play chess game.*

*If piece color is white, then player gets first move.*

*Else if player is black pieces, then computer gets first move.*

*Computer move requires storing pieces in top two rows into an array called compPieces.*

*These pieces are checked to see if at least one move is available for each of the pieces.*

*If moves are available store those pieces in validMoves array.*

*Randomly select a position in the validMoves array.*

*Have that piece found and pick the first available move for the piece in the selected position.*

*Human player chooses the current position of a piece.*

*Next human player chooses the next position to move the piece to.*

*Input validation is used to determine if the current position actually contain the piece user entered.*

*Input validation is used to determine if the piece can be moved to a certain square.*

*Player is give option to exit or save while entering piece information.*

**Flow Chart:**

Switch Menu / Game Menu

Reset chess board

If black pieces, then computer gets first move.

If white pieces, then player goes first.

CompPieces array is filled with computer’s pieces in first two rows.

Player enters current position of a piece they want to move. Then next position where they want to move it.

Pieces from computer’s pieces are checked to determine if at least one valid move. Then pieces with valid move are store in valid move array.

Game continues within do while loop until the player’s king or computer’s king has no moves remaining.

**Major Variables:**

playerName: character array containing player’s name.

chessPieces[][]: character array storing pieces on board

compPieces[][]: character array storing computer’s pieces

currentAN[]: character array storing pieces position.

nextAN[]: character array for player’s next move.

**Program:**

/\*

\* File: main.cpp

\* Author: Mathew Briguglio

\* Purpose: Chess Game

\* Created on April 21, 2015, 6:21 PM

\*/

// Library #includes

#include <cstdlib>

#include <iostream>

#include <iomanip>

#include <string>

#include <cstring>

#include <cctype>

#include <fstream>

#include "compMoves.h"

#include "moves.h"

// Global Constants

const int COLS = 10;

using namespace std;

// Function Prototypes

void displayMenu(); // Complete.

char\*\* resetBoard(int); // Complete.

void newGame(); // Complete.

void saveGame(char [], int, int); // Complete.

void loadGame(); // Incomplete.

void describeGame(); // Complete.

// Some extra functions.

void pauseScreen(); // Function pauses the screen until user presses Enter.

char \*getPlayerName(char [], int, char); // Function stores player name in character array.

int getPieceColor(); // Function asks user if they want black or white chess pieces.

void showInstruct(int); // Show user instructions for moving pieces.

// Passing array of structures through validatePos function.

bool validatePos(char \*\*, Moves [], char [], int); // (Must pass 2D array to function.) Function determines if the player actually has a piece in a position.

bool validateMove(char \*\*, char [], char [], int); // Function determines whether or not the player chose a valid move.

int \*getCompMove(char \*\*, CompMoves&, int, int); // Function determines possible moves for each piece, assigns numbers to the moves, and rand selects move for comp.

bool findValidMoves(char \*\*, CompMoves&, int, int, int); // Function accepts a structure as an argument. The structure is passed by reference. It finds valid moves for comp.

int \*moveCompPiece(char \*\*, CompMoves&, int, int, int); // Function actually moves piece on board by displaying the 2D chessPieces array.

// CHESS: 64 squares, light/dark or white/black, pieces - king, queen, rook, knight, bishop, pawn

// rows (ranks) 1-8, columns (files) a-h, promotions - pawn to any other piece

// White moves first, no skipping a move, game ends with checkmate, resign, or draw, time limit exceeded (include time for final project?)

// Queen is always 4 spots from left, or d1 or d8 (depending on white or black)

// Begin Execution

int main()

{

displayMenu(); // Function call to display game menu.

return 0;

}

// Implement menu function.

void displayMenu()

{

const int PLAY = 1, LOAD = 2, DESCRIBE = 3; // Constants for switch menu.

int opt; // integer data type for user input.

do // do while loop for menu.

{

cout << "\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n";

cout << endl << endl;

cout << "---------------\n";

cout << "CSC 17a - Chess\n"

"---------------\n\n"

"1. Play Game\n"

"2. Load Game\n"

"3. Game Info.\n"

"4. Quit Game\n\n";

do // do while loop for input validation.

{

cout << "Select a menu option: ";

cin >> opt;

} while (opt < 1 || opt > 4); // Logical operators and relational operators for decision making.

cout << "\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n\n";

switch (opt)

{

case 1: newGame(); break;

case 2: loadGame(); break;

case 3: describeGame(); break;

}

} while (opt != 4);

}

// Implement function to play chess game.

void newGame()

{

const int NAME = 40;

char playerName[NAME]; // Used to store the player's name.

char \*namePtr; // Pointer for address of playerName array.

namePtr = playerName; // Point to playerName.

char ch; // char data type for user input.

int num; // int data type for user input.

const int SIZE = 3;

char currentAN[SIZE], nextAN[SIZE]; // Character arrays holding the letter and number which gives AN (algebra notation) for chess moves.

// currentAN array is the current position of the piece, and nextAN array is the position to move to.

char \*\*chessPieces; // chessPieces 2D array stores the pieces and their positions on the chess board. (e.g. R is on )

bool validPos; // Used to validate the position of a chess piece.

bool validMove; // Used to validate the move a player chooses.

int newRow, newCol; // newRow and newCol are the row and column the player moves their piece to.

int preRow, preCol; // preRow and PreCol are the row and column where the player's piece was before moving it.

int nthMove = 1; // The current move of the game.

const int COMPSIZE = 4;

int rowsCols[COMPSIZE];

int \*rowsColsPtr;

rowsColsPtr = rowsCols;

CompMoves computer; // Structure variable computer holds the computer's pieces, valid moves, and positions of pieces with valid moves.

Moves player[3]; // player is an array of structures with 3 elements. The rook, bishop, and queen will use this array.

namePtr = getPlayerName(playerName, NAME, ch); // Call function to store player name.

num = getPieceColor(); // Ask player if they want black or white chess pieces.

cout << "Would you like to save? Enter Y/N: ";

cin >> ch;

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

saveGame(playerName, num, nthMove); // Call function to save player's name and piece color.

else

cout << "Game will not be saved.\n\n";

showInstruct(num); // Display instructions for how to move pieces using keyboard.

cout << "Press enter to continue.";

cin.get();

cout << endl << endl;

// Display that it is the first move, and display whose turn it is.

if (num == 1)

cout << "Move: " << nthMove << "\t\tCurrent Move: Computer";

else if (num == 2)

cout << "Move: " << nthMove << "\t\tCurrent Move: " << playerName;

// Return the two dimensional array so I can use it in this function.

chessPieces = resetBoard(num); // resetBoard function call resets the chess board.

// do while loop so players can continue to move pieces until player wants

// to quit game or someone loses.

do

{

// If black then use this code every even number. If white then use it every odd number.

if ((num == 1 && nthMove % 2 == 0) || (num == 2 && nthMove % 2 == 1))

{

do

{

do

{

cout << "Current Position: ";

cin >> currentAN[0] >> currentAN[1] >> currentAN[2];

validPos = validatePos(chessPieces, player, currentAN, num); // Call validPos function to determine if the player entered a valid chess piece and a valid position for it.

if (validPos == false && currentAN[0] != 'e')

cout << "Invalid Position!\n";

else if (currentAN[0] == 'e')

return;

} while (validPos == false);

do

{

cout << "Move To: ";

cin >> nextAN[0] >> nextAN[1] >> nextAN[2];

if (nextAN[0] != 'u' && nextAN[0] != 'e')

{

validMove = validateMove(chessPieces, currentAN, nextAN, num);

if (validMove == false)

cout << "Invalid Move!\n";

}

else if (nextAN[0] == 'u') // If player enters a piece that cannot move anywhere, they can enter uuu to enter a new current position.

{

do

{

cout << "Current Position: ";

cin >> currentAN[0] >> currentAN[1] >> currentAN[2];

validPos = validatePos(chessPieces, player, currentAN, num); // Call validPos function to validate current position.

if (validPos == false && currentAN[0] != 'e')

cout << "Invalid Position!\n";

else if (currentAN[0] == 'e')

return;

} while (validPos == false);

cout << "Move To: "; // If they undo current position and still need to enter where to move piece.

cin >> nextAN[0] >> nextAN[1] >> nextAN[2];

validMove = validateMove(chessPieces, currentAN, nextAN, num);

if (validMove == false && nextAN[0] != 'e')

cout << "Invalid Move!\n";

else if (nextAN[0] == 'e')

return;

}

else if (nextAN[0] == 'e')

return;

} while (validMove == false);

// Ask user if they agree with the information they entered.

cout << endl;

cout << "(You can enter S to save game.)\n";

cout << "You entered: " << currentAN[0] << currentAN[1] << currentAN[2] <<

" to " << nextAN[0] << nextAN[1] << nextAN[2] << "\n";

cout << "Is this correct? Enter Y/N: ";

cin >> ch;

if (ch == 'S' || ch == 's') // Allow player to save game here.

saveGame(playerName, num, nthMove);

cout << endl;

} while (ch == 'N' || ch == 'n' || ch == 'S' || ch == 's'); // do while asking the user if they want to finalize the move.

if (nextAN[0] != 'c' && nextAN[0] != 'p') // If not a special move, move piece to new position with legal rules.

{

if ((nextAN[0] == 'P' || nextAN[0] == 'a') && nextAN[2] == '8') // Code for pawn promotion. Pawn can be promoted to a queen, rook, knight, or bishop.

{

if (num == 1) // Just have player enter a new character for the piece if a pawn makes it to row 8.

{

do

{

cout << "Pawn Promotion! Select a piece to replace your pawn: N, B, R, or Q. ";

cin >> nextAN[0];

} while (nextAN[0] != 'N' && nextAN[0] != 'B' && nextAN[0] != 'R' && nextAN[0] != 'Q');

}

else if (num == 2)

{

do

{

cout << "Pawn Promotion! Select a piece to replace your pawn: n, b, r, or q. ";

cin >> nextAN[0];

} while (nextAN[0] != 'n' && nextAN[0] != 'b' && nextAN[0] != 'r' && nextAN[0] != 'q');

}

}

newRow = 57 - static\_cast<int>(nextAN[2]); // Convert nextAN[2], the row, to the proper row number 1-8.

newCol = static\_cast<int>(nextAN[1]) - 96; // Convert nextAN[1], the column, to the proper column number 1-8.

preRow = 57 - static\_cast<int>(currentAN[2]); // Convert initial piece position to proper row and column numbers 1-8.

preCol = static\_cast<int>(currentAN[1]) - 96;

// Update 2D array.

chessPieces[newRow][newCol] = nextAN[0]; // \*\*Very important lines of code - after validating player's move, assign the new position the piece being moved.

chessPieces[preRow][preCol] = 32; // \*\*And replace the position where the piece was before with a space.

}

else if (nextAN[0] == 'c' || nextAN[0] == 'p') // If it is a special move: castle or en passant.

{

if (nextAN[0] == 'c' && currentAN[1] == 'a') // If castling with rook in column a.

{

if (num == 1) // If black pieces.

{

chessPieces[8][3] = 'K'; // King moves two spaces closer to rook in column a.

chessPieces[8][4] = 'R'; // Rook moves one square to right of king.

}

else if (num == 2) // If white pieces.

{

chessPieces[8][3] = 'k'; // King moves two spaces closer to rook in column a.

chessPieces[8][4] = 'r'; // Rook moves one square to right of king.

}

chessPieces[8][1] = 32; // Replace king and rook original positions with a space.

chessPieces[8][5] = 32;

}

else if (nextAN[0] == 'c' && currentAN[1] == 'h') // If castling with rook in column h.

{

if (num == 1) // If black pieces.

{

chessPieces[8][7] = 'K'; // King moves two spaces closer to rook in column h.

chessPieces[8][6] = 'R'; // Rook moves one square to left of king.

}

else if (num == 2) // If white pieces.

{

chessPieces[8][7] = 'k'; // King moves two spaces closer to rook in column h.

chessPieces[8][6] = 'r'; // Rook moves one square to left of king.

}

chessPieces[8][8] = 32; // Replace king and rook original positions with a space.

chessPieces[8][5] = 32;

}

else if (nextAN[2] == 'l') // Now code for en passant.

{

if (currentAN[1] == 'b') // If player's pawn is in column b and en passant to the left.

{

if (num == 1)

chessPieces[3][1] = 'P'; // Player's pawn moves to this position and it uppercase p because num = 1 (black pieces).

else if (num == 2)

chessPieces[3][1] = 'a'; // Player has white pieces so lowercase a.

chessPieces [4][2] = 32; // Player's pawn previous position.

computer.compPieces[4][1] = '0'; // Opponent's pawn position - update compPieces array.

chessPieces[4][1] = 32; // Opponent's pawn position - update chessboard.

}

else if (currentAN[1] == 'c')

{

if (num == 1)

chessPieces[3][2] = 'P';

else if (num == 2)

chessPieces[3][2] = 'a';

chessPieces [4][3] = 32;

computer.compPieces[4][2] = '0';

chessPieces[4][2] = 32;

}

else if (currentAN[1] == 'd')

{

if (num == 1)

chessPieces[3][3] = 'P';

else if (num == 2)

chessPieces[3][3] = 'a';

chessPieces [4][4] = 32;

computer.compPieces[4][3] = '0';

chessPieces[4][3] = 32;

}

else if (currentAN[1] == 'e')

{

if (num == 1)

chessPieces[3][4] = 'P';

else if (num == 2)

chessPieces[3][4] = 'a';

chessPieces [4][5] = 32;

computer.compPieces[4][4] = '0';

chessPieces[4][4] = 32;

}

else if (currentAN[1] == 'f')

{

if (num == 1)

chessPieces[3][5] = 'P';

else if (num == 2)

chessPieces[3][5] = 'a';

chessPieces [4][6] = 32;

computer.compPieces[4][5] = '0';

chessPieces[4][5] = 32;

}

else if (currentAN[1] == 'g')

{

if (num == 1)

chessPieces[3][6] = 'P';

else if (num == 2)

chessPieces[3][6] = 'a';

chessPieces [4][7] = 32;

computer.compPieces[4][6] = '0';

chessPieces[4][6] = 32;

}

else if (currentAN[1] == 'h')

{

if (num == 1)

chessPieces[3][7] = 'P';

else if (num == 2)

chessPieces[3][7] = 'a';

chessPieces [4][8] = 32;

computer.compPieces[4][7] = '0';

chessPieces[4][7] = 32;

}

}

else if (nextAN[2] == 'r')

{

if (currentAN[1] == 'a') // If player's pawn is in column a.

{

if (num == 1)

chessPieces[3][2] = 'P';

else if (num == 2)

chessPieces[3][2] = 'a';

chessPieces[4][1] = 32;

computer.compPieces[4][2] = '0';

chessPieces[4][2] = 32;

}

else if (currentAN[1] == 'b')

{

if (num == 1)

chessPieces[3][3] = 'P';

else if (num == 2)

chessPieces[3][3] = 'a';

chessPieces[4][2] = 32;

computer.compPieces[4][3] = '0';

chessPieces[4][3] = 32;

}

else if (currentAN[1] == 'c')

{

if (num == 1)

chessPieces[3][4] = 'P';

else if (num == 2)

chessPieces[3][4] = 'a';

chessPieces[4][3] = 32;

computer.compPieces[4][4] = '0';

chessPieces[4][4] = 32;

}

else if (currentAN[1] == 'd')

{

if (num == 1)

chessPieces[3][5] = 'P';

else if (num == 2)

chessPieces[3][5] = 'a';

chessPieces[4][4] = 32;

computer.compPieces[4][5] = '0';

chessPieces[4][5] = 32;

}

else if (currentAN[1] == 'e')

{

if (num == 1)

chessPieces[3][6] = 'P';

else if (num == 2)

chessPieces[3][6] = 'a';

chessPieces[4][5] = 32;

computer.compPieces[4][6] = '0';

chessPieces[4][6] = 32;

}

else if (currentAN[1] == 'f')

{

if (num == 1)

chessPieces[3][7] = 'P';

else if (num == 2)

chessPieces[3][7] = 'a';

chessPieces[4][6] = 32;

computer.compPieces[4][7] = '0';

chessPieces[4][7] = 32;

}

else if (currentAN[1] == 'g')

{

if (num == 1)

chessPieces[3][8] = 'P';

else if (num == 2)

chessPieces[3][8] = 'a';

chessPieces[4][7] = 32;

computer.compPieces[4][8] = '0';

chessPieces[4][8] = 32;

}

}

}

// This large block of code is for the human player.

}

else if ((num == 1 && nthMove % 2 == 1) || (num == 2 && nthMove % 2 == 0))

{

// Fill compPiece array on computer's first move.

if ((num == 1 && nthMove == 1) || (num == 2 && nthMove == 2))

{

for (int row = 1; row < 3; row++)

{

for (int col = 1; col < 9; col++)

{

computer.compPieces[row][col] = chessPieces[row][col]; // Store the computer's pieces and their positions (based on the positions in the 2D array for chessboard)

}

}

for (int row = 3; row < 9; row++)

{

for (int col = 1; col < 9; col++)

{

computer.compPieces[row][col] = '0'; // Fill remaining array elements with zeros to represent a square without one of the computer's pieces.

}

}

}

// Must update compPieces array right before computer's next move so

// comp knows where all its pieces are located.

// If the player captures one of the computer's pieces, update the compPiece array

// so the computer knows which pieces it still has on the chessboard.

if (nthMove > 3)

{

if (computer.compPieces[newRow][newCol] != '0') // If the position newRow newCol in compPieces array doesn't hold a 0, then it must mean one of the comp pieces

computer.compPieces[newRow][newCol] = '0'; // was there, and the player just moved their piece there to capture it. So replace this position with a 0.

}

// Must update compPieces array right before computer's next move so

// comp knows where all its pieces are located.

rowsColsPtr = getCompMove(chessPieces, computer, num, nthMove); // Call function to determine valid moves the computer can make and to select a move.

preRow = rowsColsPtr[0];

preCol = rowsColsPtr[1];

newRow = rowsColsPtr[2];

newCol = rowsColsPtr[3];

// Update compPieces array after computer moves a piece. No piece gets

// a zero, the newly filled square gets added to compPieces array.

computer.compPieces[newRow][newCol] = chessPieces[preRow][preCol];

computer.compPieces[preRow][preCol] = '0';

// Print out compPieces array so I know the pieces are being updated.

/\* cout << endl << endl;

for (int row = 1; row < 9; row++)

{

for (int col = 1; col < 9; col++)

{

cout << computer.compPieces[row][col] << " ";

}

cout << endl;

}

cout << endl << endl; \*/

// Update chessboard so it can be displayed.

chessPieces[newRow][newCol] = chessPieces[preRow][preCol];

chessPieces[preRow][preCol] = 32;

}

nthMove++; // Increment nthMove variable to keep track of whose turn it is.

// Display the total number of moves taken so far in the game.

cout << "Move: " << nthMove << "\t";

// Display whose turn it is.

// If the player chose black pieces and it is the player's turn.

if (num == 1 && nthMove % 2 == 0)

cout << "\tCurrent Move: " << playerName << endl;

// If the player chose white pieces and it is the players turn.

else if (num == 2 && nthMove % 2 == 1)

cout << "\tCurrent Move: " << playerName << endl;

// If the player chose black pieces and it is the computer's turn.

if (num == 1 && nthMove % 2 == 1)

cout << "\tCurrent Move: Computer\n";

// If the player chose white pieces and it is the computer's turn.

else if (num == 2 && nthMove % 2 == 0)

cout << "\tCurrent Move: Computer\n";

// Display updated chess board.

cout << "\n";

for (int count = 0; count < 10; count++) // Outer for loop displays 1-8 (rows/ranks).

{

for (int index = 0; index < 10; index++) // Inner for loop displays a-h (columns/files).

{

cout << chessPieces[count][index];

if ((count == 0 || count == 9) && index < 9)

cout << " ";

if (count > 0 && count < 9 && index < 9)

cout << " | ";

}

if (count > 0 && count < 9) // Inserting extra | to make vertical lines. For rows 1-8 only.

{

cout << endl;

cout << " | | | | | | | | | ";

}

cout << "\n";

if (count < 9)

cout << " \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_";

cout << "\n";

}

ch = 'Y';

} while (ch != 'N' && ch != 'n'); // do while loop to determine when game ends. Idk how I will signify the game ending just yet. Maybe, when

// Save game only at end of game play (or during game play).

//saveGame();

}

// Implement function to put pieces back to original starting positions for a new game.

// Important note: The positions/subscripts of the 2D array are represented with int data types.

// However, the elements of the array are char data types. So when displaying an element or

// comparing an element the int values of the row and column of the array are to be used.

char\*\* resetBoard(int pieceColor)

{

int ascii = 97, num = 0; // ascii is for letters a-h. num is for digits 1-8.

// Dynamically allocate memory to 2D array.

char\*\* chessBoard = new char\*[10]; // Two dimensional array storing position of chess pieces.

for (int i = 0; i < 10; i++)

chessBoard[i] = new char[10];

// First for loop assigns 1-8 to column 1 and a-h to row 10.

// a-h is 97-104 in ASCII code. 1-8 is 49-56 in ASCII code.

for (int count = 0; count < 10; count++) // count is the row number.

{

for (int index = 0; index < 10; index++) // index is the column number.

{

if (count == 9 && index > 0 && index < 9) // Row 10 and columns 1 through 8. Prints a-h.

{

chessBoard[count][index] = ascii;

ascii++;

}

else if (index == 0 && count > 0 && count < 9) // Column 1 and rows 1 through 8.

{

chessBoard[count][index] = 56 - num; // 56 is the ascii code for 8.

num++; // Taking 8 - num starts with 8 - 0 = 8, then 8 - 1 = 7, etc. The numbers are decreasing from 8 to 1.

}

else

chessBoard[count][index] = 32; // Assign a space to empty spaces. No pieces yet.

}

}

// Second for loop assigns 1-8 to column 10, a-h to row 1, and a space to all other elements.

ascii = 97;

num = 0;

for (int count = 0; count < 10; count++) // count is the row number.

{

for (int index = 0; index < 10; index++) // index is the column number.

{

if (count == 0 && index > 0 && index < 9) // Row 1 and columns 1 through 8. Prints a-h.

{

chessBoard[count][index] = ascii;

ascii++;

}

else if (index == 9 && count > 0 && count < 9) // Column 10 and rows 1 through 8

{

chessBoard[count][index] = 56 - num;

num++;

}

}

}

// Assign the corners \* because they're empty.

chessBoard[0][0] = '\*';

chessBoard[0][9] = '\*';

chessBoard[9][0] = '\*';

chessBoard[9][9] = '\*';

// I want the color the player chooses to be at the bottom of the board.

// If player chooses 2 in newGame function, they chose white, and white pieces (lowercase letters) are at bottom of board.

if (pieceColor == 2)

{

// Assign rows 2 and 3 the appropriate chess pieces. Rows 2 and 3 will be uppercase (black pieces).

chessBoard[1][1] = chessBoard[1][8] = 'R';

chessBoard[1][2] = chessBoard[1][7] = 'N';

chessBoard[1][3] = chessBoard[1][6] = 'B';

chessBoard[1][4] = 'Q';

chessBoard[1][5] = 'K';

for (int count = 1; count < 9; count++)

chessBoard[2][count] = 'P';

// Assign rows 8 and 9 the appropriate chess pieces. Rows 8 and 9 will be lowercase (white pieces).

chessBoard[8][1] = chessBoard[8][8] = 'r';

chessBoard[8][2] = chessBoard[8][7] = 'n';

chessBoard[8][3] = chessBoard[8][6] = 'b';

chessBoard[8][4] = 'q';

chessBoard[8][5] = 'k';

for (int count = 1; count < 9; count++)

chessBoard[7][count] = 'a';

// To understand why pieces are placed in certain rows just look at a chess board.

}

// If player chooses 1 in newGame function, they chose black, and black pieces (uppercase letters) are at bottom of board.

else if (pieceColor == 1)

{

// Assign rows 2 and 3 the appropriate chess pieces. Rows 2 and 3 will be uppercase (black pieces).

chessBoard[1][1] = chessBoard[1][8] = 'r';

chessBoard[1][2] = chessBoard[1][7] = 'n';

chessBoard[1][3] = chessBoard[1][6] = 'b';

chessBoard[1][4] = 'q';

chessBoard[1][5] = 'k';

for (int count = 1; count < 9; count++)

chessBoard[2][count] = 'a';

// Assign rows 8 and 9 the appropriate chess pieces. Rows 8 and 9 will be lowercase (white pieces).

chessBoard[8][1] = chessBoard[8][8] = 'R';

chessBoard[8][2] = chessBoard[8][7] = 'N';

chessBoard[8][3] = chessBoard[8][6] = 'B';

chessBoard[8][4] = 'Q';

chessBoard[8][5] = 'K';

for (int count = 1; count < 9; count++)

chessBoard[7][count] = 'P';

// To understand why pieces are placed in certain rows just look at a chess board.

}

// Display chess board with lines/squares.

cout << "\n\n\n";

for (int count = 0; count < 10; count++) // Outer for loop displays 1-8 (rows/ranks).

{

for (int index = 0; index < 10; index++) // Inner for loop displays a-h (columns/files).

{

cout << chessBoard[count][index];

if ((count == 0 || count == 9) && index < 9)

cout << " ";

if (count > 0 && count < 9 && index < 9)

cout << " | ";

}

if (count > 0 && count < 9) // Inserting extra | to make vertical lines. For rows 1-8 only.

{

cout << endl;

cout << " | | | | | | | | | ";

}

cout << "\n";

if (count < 9)

cout << " \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_";

cout << "\n";

}

return chessBoard;

// Chess board without lines/squares.

/\* cout << "\n\n\n";

for (int count = 0; count < ROWS; count++) // Outer for loop displays 1-8 (rows/ranks).

{

for (int index = 0; index < COLS; index++) // Inner for loop displays a-h (columns/files).

cout << chessBoard[count][index] << "\t";

cout << "\n\n\n";

}\*/

}

void pauseScreen()

{

cout << "Press enter to continue.";

cin.ignore();

cin.get();

cout << endl << endl;

}

char \*getPlayerName(char name[], int NAME, char ch)

{

int count = 0;

do

{

cin.ignore();

cout << "Enter player initials: ";

cin >> name;

cout << "\nYou entered: " << name << endl;

cout << "Is this correct? Enter Y/N: ";

cin >> ch;

} while (ch == 'N' || ch == 'n');

cout << endl;

return name;

}

int getPieceColor()

{

int number; // For user input options.

cout << "Color of chess pieces:\n\n";

cout << "1. Black (uppercase letters)\n"

"2. White (lowercase letters)\n\n";

do

{

cout << "Select a color: ";

cin >> number;

} while (number < 1 || number > 2);

cout << endl << endl;

return number;

}

void showInstruct(int num)

{

// If num = 1, the player chose black pieces.

if (num == 1)

{

cout << "Instructions:\n\n";

cout << "To move a piece, enter the letter representing the piece, the letter representing the\n"

"column (file) the piece is in, and the number representing the row (rank) the piece is\n"

"in. Then enter the same information for the square where you'd like to move it.\n\n"

"For example: Consider the pawn (represented by the letter P) in column c and row 2.\n"

"You would press Pc2 then ENTER for its current position. If you wanted to move it one\n"

"square forward to column c and row 3, then you would press Pc3.\n\n"

"Also, if you chose Black pieces, then you must enter an uppercase letter to represent the\n"

"piece you are moving. If you chose white pieces, you must enter a lowercase letter.\n\n";

pauseScreen();

cout << "Special Moves:\n\n"

"Castling: For \"Current Position\" enter position of the rook you want to use when castling and\n"

" for \"Move To\" enter ccc\n"

"En passant: For \"Current Position\" enter position of the pawn you want to use with En passant\n"

" move and for \"Move To\" enter ppl (en passant to left) or ppr (en passant to right)\n"

"Pawn Promotion: Happens automatically once a pawn reaches the eighth row/rank of chessboard\n\n"

"Hotkeys:\n\n"

"Exit: eee (You may enter eee to exit when the game asks for \"Current Position\" or \"Move To\")\n"

"Undo: uuu (After entering the \"Current Position\" you may enter uuu to choose a different piece to move)\n"

"Save: S (You may save game in beginning of game or when finalizing a move)\n\n";

}

// If num = 2, the player chose white pieces.

else if (num == 2)

{

cout << "Instructions:\n\n";

cout << "To move a piece, enter the letter representing the piece, the letter representing the\n"

"column (file) the piece is in, and the number representing the row (rank) the piece is\n"

"in. Then enter the same information for the square where you'd like to move it.\n\n"

"For example: Consider the pawn (represented by the letter a) in column c and row 2.\n"

"You would press ac2 then ENTER for its current position. If you wanted to move it one\n"

"square forward to column c and row 3, then you would press ac3.\n\n"

"Also, if you chose Black pieces, then you must enter an uppercase letter to represent the\n"

"piece you are moving. If you chose white pieces, you must enter a lowercase letter.\n\n";

pauseScreen();

cout << "Special Moves:\n\n"

"Castling: For \"Current Position\" enter position of the rook you want to use when castling and\n"

" for \"Move To\" enter ccc\n"

"En passant: For \"Current Position\" enter position of the pawn you want to use with En passant\n"

" move and for \"Move To\" enter ppl (en passant to left) or ppr (en passant to right)\n"

"Pawn Promotion: Happens automatically once a pawn reaches the eighth row/rank of chessboard\n\n"

"Hotkeys:\n\n"

"Exit: eee (You may enter eee to exit when the game asks for \"Current Position\" or \"Move To\")\n"

"Undo: uuu (After entering the \"Current Position\" you may enter uuu to choose a different piece to move)\n"

"Save: S (You may save game in beginning of game or when finalizing a move)\n\n";

}

}

// Implement function to save the position of pieces or to save player's score.

void saveGame(char name[], int num, int nthMove)

{

cout << "Saving game...\n";

cout << "Game saved.\n\n";

fstream file;

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

file.write(name, sizeof(name));

file.write(reinterpret\_cast<char \*>(&nthMove), sizeof(nthMove));

file.close();

}

// Implement function to continue a previous game. Pieces remain in the positions they were left in.

void loadGame()

{

const int SIZE = 20;

char chessData[SIZE];

int nthMove;

for (int count = 0; count < SIZE - 1; count++)

chessData[count] = ' ';

fstream file;

file.open("chess.txt", ios::in | ios::binary);

file.read(chessData, sizeof(chessData));

file.read(reinterpret\_cast<char \*>(&nthMove), sizeof(nthMove));

cout << "Player Name: " << chessData << endl;

cout << "Number of Moves: " << nthMove << endl << endl;

file.close();

}

// Implement function to give game description and rules.

void describeGame()

{

char ch;

cout << "GAME DESCRIPTION:\n\n";

cout << "Chess is a strategy-based board game played on a checkered game board. The board is an eight by eight grid,\n"

"thus it contains 64 squares. Chess is a two player game, and each player begins with 16 pieces: two rooks,\n"

"two knights, two bishops, one queen, one king, and eight pawns. Each game piece moves in a unique way. If player\n"

"one moves one of their game pieces and it happens to land on a square occupied by their opponent's game piece,\n"

"then player one captures player two's game piece and that captured piece is permanently removed from the board.\n"

"(Player one's game piece now occupies the square and remains on the board.) The only piece that cannot be captured\n"

"is the king. The objective of the game is to checkmate the opponent's king, meaning that any move the opponent's\n"

"king makes, it would be threatened by inevitable capture. If the opponent's king is still able to move to a square\n"

"where it cannot be captured, then the game continues! Players also earn points by capturing the opponent's game pieces.\n\n";

// Should I keep score throughout the game, and use the points?

cout << "Do you already know how the pieces move? If you're a pro, then enter Y. If you need some review, then enter N.";

cin >> ch;

if (islower(ch))

ch = toupper(ch);

if (ch == 'N')

{

cout << "\nMOVING THE PIECES:\n\n";

cout << "Rook: The rook can move any number of squares horizontally or vertically, but it cannot leap over any other pieces.\n"

"Knight: The knight can move two squares horizontally and then one square vertically OR two squares vertically and then\n"

" one square horizontally. Unlike the other pieces, the knight can leap over any other piece and land on its destination\n"

" square. The movement of the knight is often referred to as an \"L-shape\"\n"

"Bishop: The bishop can move any number of squares diagonally, but it cannot leap over any other pieces.\n"

"Queen: The queen can move any number of squares horizontally, vertically, or diagonally, but it cannot leap over any other pieces.\n"

"King: The king can move one square horizontally, vertically, or diagonally.\n"

"Pawn: The pawn can move forward one square or, if it is the pawn's first move, it can move forward two squares. The pawn can also\n"

" move diagonally forward one square only when capturing another piece. Lastly, if a pawn reaches the final row of the board\n"

" on the opponent's side, then the pawn is promoted to the player's choice of either a rook, knight, bishop, or queen.\n\n";

cout << "SPECIAL MOVES:\n\n"

"\* Castling: If one of the rooks and the king have not yet moved from their original positions, then they may castle. The king moves\n"

" two squares toward the rook the player chooses, and the rook moves to the adjacent square on the other side of the king.\n"

"\* En Passant: If player 2 moves their pawn forward two spaces on the pawn's first move and one of player 1's pawns happens to be to the\n"

" left or right (on an adjacent square) then player 1 may capture player 2's pawn \"en passant\" (in passing) by moving to\n"

" the square immediately behind player 2's pawn.\n"

"\* Pawn Promotion: If a pawn makes it to the eighth row of the chessboard - on the opponent's side - they may promote their pawn. That is,\n"

" the player will be given the choice of replacing their pawn with either a knight, bishop, rook, or queen.\n\n";

}

else

cout << "Great! You're a pro!\n\n";

pauseScreen(); // pauseScreen function.

cout << "NOTATION KEY FOR THE GAME PIECES:\n\n"

"Rook: R or r\n"

"Knight: N or n\n"

"Bishop: B or b\n"

"Queen: Q or q\n"

"King: K or k\n"

"Pawn: P or a\n\n"

"Black Pieces: uppercase letters\n"

"White Pieces: lowercase letters\n\n";

pauseScreen();

// Display the point system for when pieces are captured.

cout << "ASSIGNMENT OF POINT VALUES:\n\n"

"Pawn: 1\n"

"Knight: 3\n"

"Bishop: 3\n"

"Rook: 5\n"

"Queen: 9\n\n";

cout << "A FEW EXTRA RULES:\n\n";

cout << "\* White always has the first move.\n"

"\* There are no restrictions on pawn promotions; the player may choose whichever piece they prefer each time.\n"

"\* Players may not skip turns.\n"

"\* To capture an opponent's piece, you move your own piece to an appropriate square containing one of the opponent's pieces.\n"

" You then permanently remove their piece from the game board, and your piece remains on the square.\n"

"\* The game ends when either a player resigns, there is a draw, or there is a checkmate.\n"

" - Draw: It is a draw when neither player can win the game.\n"

" - Resignation: A player may resign from (or leave/quit) the game if they believe they will lose.\n"

" - Checkmate: A checkmate occurs when one player's king cannot move to a new square without being threatened by inevitable capture.\n\n";

pauseScreen();

}

// Implement function to determine if the player entered there own piece and entered its correct position on the board.

bool validatePos(char \*\*chessPieces, Moves p[], char cur[], int num)

{

// Moves p[] is the array of structures in which each element contains

// the number of squares certain pieces can move in any direction.

// QUESTION: Do I want to use the array of structures? It will over-complicate the code. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// cur[] is the currentAN array from newGame function.

bool validation;

int col, row;

col = static\_cast<int>(cur[1]) - 96;

row = 57 - static\_cast<int>(cur[2]);

// Must translate the Algebraic Notation (AN) entered by player to the actual row and column 0-9 of 2D array.

// To determine the position in the 2D array based on the column/file letter entered by the player

// realize that the chess board has \*, a, b, c, d, ... which corresponds with 0, 1, 2, 3, 4, ... and that

// a, b, c, etc, are 97, 98, 99, in the ASCII code. So, to get 0, 1, 2, etc you statically cast the letter

// stored in currentAN[1] as an int and then subtract 96. This gives the column in the 2D array.

// Next, to get the row in the 2D array, realize that the chess board has char variables \*, 8, 7, 6, ... which

// are changing in the opposite way of the actual subscripts of the 2D array. Also, the 8, 7, 6, ... are

// 56, 55, 54, ... in the ASCII code. So, to get 0, 1, 2, etc ... you take 57 and subtract the int value of

// the digit they entered for the row - Ex. If they entered row 7, ASCII code for 7 is 55, and 57 - 55 = 2.

// This gives the row in the 2D array.

// Player chose black pieces.

if (num == 1)

{

if (cur[0] == 'R' || cur[0] == 'N' || cur[0] == 'B' || cur[0] == 'Q' || cur[0] == 'K' || cur[0] == 'P') // First, just check for an uppercase letter for the chess piece.

{

if (chessPieces[row][col] == cur[0]) // Next, check if the piece in the position entered by player actually is the piece they entered.

validation = true;

}

else if (cur[0] == 'c' && cur[1] == 'c' && cur[2] == 'c') // Validation if player wants to castle with rook and king.

validation = true;

else

validation = false; // If the player didn't enter an uppercase letter and the position doesn't contain the piece they

// entered, then they need to change their input.

}

// Player chose white pieces.

else if (num == 2)

{

if (cur[0] == 'r' || cur[0] == 'n' || cur[0] == 'b' || cur[0] == 'q' || cur[0] == 'k' || cur[0] == 'a') // First, just check for a lowercase letter for the chess piece.

{

if (chessPieces[row][col] == cur[0]) // Next, check if the piece in the position entered by player actually is the piece they entered.

validation = true;

}

else

validation = false; // If the player didn't enter a lowercase letter and the position doesn't contain the piece they

// entered, then they need to change their input.

}

return validation;

}

bool validateMove(char \*\*chessPieces, char cur[], char nex[], int num)

{

// cur[] is currentAN array and nex[] is nextAN array from newGame function.

// This function must determine which piece the player wants to move, and whether or not the move is valid.

// Important: make sure pieces cannot move through other pieces. No pieces can be in the path of another piece.

// So, if either the piece moves off the board or encounters another piece, then the initial piece must stop.

// NOTE: I'm not ready for capturing pieces yet. For now, only moves where the player's chess piece cannot go through another piece.

// What if the player enters their current position, but there isn't a valid move for the piece?

// Won't the program just keep saying invalid move, and not let the user try to move a different piece?

bool boolianL = false, // Used in while loop for rook.

boolianR = false, // Used in while loop for rook.

boolianB = false,

boolianF = false,

validation; // Returned to new game function. If true, move is valid. If false, move is invalid.

int col, row; // Column and row positions in 2D chessPieces array.

col = static\_cast<int>(cur[1]) - 96; // Column number in array where chess piece is currently (before move).

row = 57 - static\_cast<int>(cur[2]); // Row number in array where chess piece is currently (before move).

int colNext, rowNext; // Column and row positions in 2D chessPieces array.

colNext = static\_cast<int>(nex[1]) - 96; // Column number player wants to move to.

rowNext = 57 - static\_cast<int>(nex[2]); // Row number player wants to move to.

int colL, // colL (column left) is used to determine how many spaces a piece can move to the left.

colR, // colR (column right) is used to determine how many spaces a piece can move to the right.

rowF, // rowF (row forward) is used to determine how many squares a piece can move forward.

rowB, // rowB (row Backward) is used to determine how many squares a piece can move backward.

leftMost, // Max number of squares a piece can move to the left.

rightMost, // Max number of squares a piece can move to the right.

forwardMost, // Max number of squares a piece can move forward.

backMost; // Max number of squares a piece can move backward.

// Black pieces.

if (num == 1)

{

if (cur[0] == nex[0] && (cur[1] != nex[1] || cur[2] != nex[2]) &&

colNext > 0 && colNext < 9 && rowNext > 0 && rowNext < 9)

{ // First, make sure the player entered the same piece and either the column or the row is different

// in the nextAN array from the currentAN array. Also, make sure the row or column is on the board.

if (cur[0] == 'P') // Validate moves for a pawn.

{

if (row == 7)

{

if (!isalpha(chessPieces[row - 1][col]) && nex[1] == cur[1] && nex[2] == cur[2] + 1)

validation = true; // If no pieces one square up from pawn and player wants to move one square up, then valid move.

else if (!isalpha(chessPieces[row - 1][col]) && !isalpha(chessPieces[row - 2][col]) && nex[1] == cur[1] && nex[2] == cur[2] + 2)

validation = true; // If no pieces one or two squares up from pawn and player wants to move two squares up, then valid move.

else if (islower(chessPieces[row - 1][col - 1]) && nex[1] == cur[1] - 1 && nex[2] == cur[2] + 1) // Player wants to capture up and to left.

validation = true;

else if (islower(chessPieces[row - 1][col + 1]) && nex[1] == cur[1] + 1 && nex[2] == cur[2] + 1) // Player wants to capture up and to right

validation = true;

}

else if (row != 7)

{

if (!isalpha(chessPieces[row - 1][col]) && nex[1] == cur[1] && nex[2] == cur[2] + 1)

validation = true; // If no pieces one square up from pawn and player wants to move one square up, then valid move.

else if (islower(chessPieces[row - 1][col - 1]) && nex[1] == cur[1] - 1 && nex[2] == cur[2] + 1) // Player wants to capture up and to left.

validation = true;

else if (islower(chessPieces[row - 1][col + 1]) && nex[1] == cur[1] + 1 && nex[2] == cur[2] + 1) // Player wants to capture up and to right

validation = true;

// Need an else if statement for en passant move.

}

}

else if (cur[0] == 'R')

{

// Rook has max of 4 directional options. Column will tell you how many squares left or right rook can move.

// Row will tell you how many squares forward or back rook can move.

if (nex[2] == cur[2] && nex[1] != cur[1]) // If player wants to keep rook in same row, but move to a different column.

{

colL = col; // Initialize both colL and colR to the value in col.

colR = col; // By doing this, colL can be decremented and colR can be incremented, simultaneously.

while (boolianL == false)

{

colL--;

if (isalpha(chessPieces[row][colL]) || colL == 0) // If this new column has a piece in it or if the column is 0.

{

if (isupper(chessPieces[row][colL]) || colL == 0)

leftMost = colL + 1; // The furthest to the left that the rook can move. (With another black piece next to it.)

else if (islower(chessPieces[row][colL])) // Rook can capture the piece blocking its path.

leftMost = colL;

boolianL = true; // Once leftMost value is found, boolianL is assigned the value true and loop terminates.

}

}

while (boolianR == false)

{

colR++;

if (isalpha(chessPieces[row][colR]) || colR == 9) // If this new column has a piece in it or if the column is 9.

{

if (isupper(chessPieces[row][colR]) || colR == 9)

rightMost = colR - 1; // The furthest to the right that the rook can move. (With another black piece next to it.)

else if (islower(chessPieces[row][colR])) // Rook can capture the piece blocking its path.

rightMost = colR;

boolianR = true; // Once rightMost value is found, boolianR is assigned the value true and loop terminates.

}

}

if (colNext >= leftMost && colNext <= rightMost) // Now check whether nex[1] (column player wants to move to) is b/t leftMost and rightMost.

validation = true; // If column they want is between the values, then valid move.

}

else if (nex[1] == cur[1] && nex[2] != cur[2]) // If player wants to keep rook in same column, but move to a different row.

{

// Using same variables for counting max square the rook

// can move forward and backward. Should say rowF/rowB.

rowB = row; // Initialize both colL and colR to the value in row. I'm using colL and colR as the row numbers!!!

rowF = row; // By doing this, colL can be decremented and colR can be incremented, simultaneously.

while (boolianF == false)

{

// Decrement row is going up the board - so forward.

rowF--;

if (isalpha(chessPieces[rowF][col]) || rowF == 0) // If this new row has a piece in it or if the row is 0.

{

if (isupper(chessPieces[rowF][col]) || rowF == 0)

forwardMost = rowF + 1; // The furthest that the rook can move forward. (Toward top of board - closer to row 0 - so smaller value for row.)

else if (islower(chessPieces[rowF][col])) // Rook can capture piece blocking its way.

forwardMost = rowF;

boolianF = true; // Once forwardMost value is found, boolianF is assigned the value true and loop terminates.

}

}

while (boolianB == false)

{

rowB++;

if (isalpha(chessPieces[rowB][col]) || rowB == 9) // If this new row has a piece in it or if the row is 9.

{

if (isupper(chessPieces[rowB][col]) || rowB == 9)

backMost = rowB - 1; // The furthest that the rook can move backward. (Toward bottom of board - closer to row 9 - so larger value for row.)

else if (islower(chessPieces[rowB][col])) // Rook can capture piece blocking its way.

backMost = rowB;

boolianB = true; // Once rightMost value is found, boolianR is assigned the value true and loop terminates.

}

}

if (rowNext >= forwardMost && rowNext <= backMost) // Now check whether nex[1] (column player wants to move to) is b/t backMost and forwardMost.

validation = true; // Note: forwardMost is actually smaller row number and backMost is larger row number.

// If column they want is between the values, then valid move.

}

}

else if (cur[0] == 'N')

validation = true;

else if (cur[0] == 'B')

validation = true;

else if (cur[0] == 'Q')

validation = true;

else if (cur[0] == 'K')

validation = true;

} // Next are special moves: castling and en passant. Note: A lot of options for en passant.

else if (nex[0] == 'c' && nex[1] == 'c' && nex[2] == 'c' && cur[1] == 'a' && cur[2] == '1' && chessPieces[8][5] == 'K'// Rook/king must be in original positions (and must not have been moved)

&& !isalpha(chessPieces[8][2]) && !isalpha(chessPieces[8][3]) && !isalpha(chessPieces[8][4])) // Cannot be any pieces between rook and king. (Rook on left side of board)

validation = true;

else if (nex[0] == 'c' && nex[1] == 'c' && nex[2] == 'c' && cur[1] == 'h' && cur[2] == '1' && chessPieces[8][5] == 'K'

&& !isalpha(chessPieces[8][7]) && !isalpha(chessPieces[8][6])) // Cannot be any pieces between rook and king. (Rook on right side of board)

validation = true;

else if (nex[0] == 'p' && nex[1] == 'p' && nex[2] == 'r' && cur[0] == 'P' && cur[2] == '5' && cur[1] == 'a' && chessPieces[4][2] == 'a' // Following else if statements for en passant to right.

&& !isalpha(chessPieces[3][2])) // For en passant, pawn position must be col a-h and row 5. There must be an opponent's pawn in adjacent square.

validation = true; // There must also be no piece immediately behind the opponent's pawn. Also pawn may en passant to left or right.

else if (nex[0] == 'p' && nex[1] == 'p' && nex[2] == 'r' && cur[0] == 'P' && cur[2] == '5' && cur[1] == 'b' && chessPieces[4][3] == 'a'

&& !isalpha(chessPieces[3][3]))

validation = true;

else if (nex[0] == 'p' && nex[1] == 'p' && nex[2] == 'r' && cur[0] == 'P' && cur[2] == '5' && cur[1] == 'c' && chessPieces[4][4] == 'a'

&& !isalpha(chessPieces[3][4]))

validation = true;

else if (nex[0] == 'p' && nex[1] == 'p' && nex[2] == 'r' && cur[0] == 'P' && cur[2] == '5' && cur[1] == 'd' && chessPieces[4][5] == 'a'

&& !isalpha(chessPieces[3][5]))

validation = true;

else if (nex[0] == 'p' && nex[1] == 'p' && nex[2] == 'r' && cur[0] == 'P' && cur[2] == '5' && cur[1] == 'e' && chessPieces[4][6] == 'a'

&& !isalpha(chessPieces[3][6]))

validation = true;

else if (nex[0] == 'p' && nex[1] == 'p' && nex[2] == 'r' && cur[0] == 'P' && cur[2] == '5' && cur[1] == 'f' && chessPieces[4][7] == 'a'

&& !isalpha(chessPieces[3][7]))

validation = true;

else if (nex[0] == 'p' && nex[1] == 'p' && nex[2] == 'r' && cur[0] == 'P' && cur[2] == '5' && cur[1] == 'g' && chessPieces[4][8] == 'a'

&& !isalpha(chessPieces[3][8]))

validation = true;

else if (nex[0] == 'p' && nex[1] == 'p' && nex[2] == 'l' && cur[0] == 'P' && cur[2] == '5' && cur[1] == 'b' && chessPieces[4][1] == 'a' // Following else if statements for en passant to left.

&& !isalpha(chessPieces[3][1]))

validation = true;

else if (nex[0] == 'p' && nex[1] == 'p' && nex[2] == 'l' && cur[0] == 'P' && cur[2] == '5' && cur[1] == 'c' && chessPieces[4][2] == 'a'

&& !isalpha(chessPieces[3][2]))

validation = true;

else if (nex[0] == 'p' && nex[1] == 'p' && nex[2] == 'l' && cur[0] == 'P' && cur[2] == '5' && cur[1] == 'd' && chessPieces[4][3] == 'a'

&& !isalpha(chessPieces[3][3]))

validation = true;

else if (nex[0] == 'p' && nex[1] == 'p' && nex[2] == 'l' && cur[0] == 'P' && cur[2] == '5' && cur[1] == 'e' && chessPieces[4][4] == 'a'

&& !isalpha(chessPieces[3][4]))

validation = true;

else if (nex[0] == 'p' && nex[1] == 'p' && nex[2] == 'l' && cur[0] == 'P' && cur[2] == '5' && cur[1] == 'f' && chessPieces[4][5] == 'a'

&& !isalpha(chessPieces[3][5]))

validation = true;

else if (nex[0] == 'p' && nex[1] == 'p' && nex[2] == 'l' && cur[0] == 'P' && cur[2] == '5' && cur[1] == 'g' && chessPieces[4][6] == 'a'

&& !isalpha(chessPieces[3][6]))

validation = true;

else if (nex[0] == 'p' && nex[1] == 'p' && nex[2] == 'l' && cur[0] == 'P' && cur[2] == '5' && cur[1] == 'h' && chessPieces[4][7] == 'a'

&& !isalpha(chessPieces[3][7]))

validation = true;

else

validation = false;

}

// White pieces.

else if (num == 2)

{

if (cur[0] == nex[0] && (cur[1] != nex[1] || cur[2] != nex[2]) && colNext > 0 && colNext < 9 && rowNext > 0 && rowNext < 9)

{ // First, make sure the player entered the same piece and either the column or the row is different

// in the nextAN array from the currentAN array. Also, make sure the row or column is on the board.

if (cur[0] == 'a') // Validate moves for a pawn.

{

if (row == 7)

{

if (!isalpha(chessPieces[row - 1][col]) && nex[1] == cur[1] && nex[2] == cur[2] + 1)

validation = true; // If no pieces one square up from pawn and player wants to move one square up, then valid move.

else if (!isalpha(chessPieces[row - 1][col]) && !isalpha(chessPieces[row - 2][col]) && nex[1] == cur[1] && nex[2] == cur[2] + 2)

validation = true; // If no pieces one or two squares up from pawn and player wants to move two squares up, then valid move.

else if (isupper(chessPieces[row - 1][col - 1]) && nex[1] == cur[1] - 1 && nex[2] == cur[2] + 1) // Player wants to capture up and to left.

validation = true;

else if (isupper(chessPieces[row - 1][col + 1]) && nex[1] == cur[1] + 1 && nex[2] == cur[2] + 1) // Player wants to capture up and to right

validation = true;

}

else if (row != 7)

{

if (!isalpha(chessPieces[row - 1][col]) && nex[1] == cur[1] && nex[2] == cur[2] + 1)

validation = true; // If no pieces one square up from pawn and player wants to move one square up, then valid move.

else if (isupper(chessPieces[row - 1][col - 1]) && nex[1] == cur[1] - 1 && nex[2] == cur[2] + 1) // Player wants to capture up and to left.

validation = true;

else if (isupper(chessPieces[row - 1][col + 1]) && nex[1] == cur[1] + 1 && nex[2] == cur[2] + 1) // Player wants to capture up and to right

validation = true;

// Need an else if statement for en passant move.

}

}

else if (cur[0] == 'r')

{

// Rook has max of 4 directional options. Column will tell you how many squares left or right rook can move.

// Row will tell you how many squares forward or back rook can move.

if (nex[2] == cur[2] && nex[1] != cur[1]) // If player wants to keep rook in same row, but move to a different column.

{

colL = col; // Initialize both colL and colR to the value in col.

colR = col; // By doing this, colL can be decremented and colR can be incremented, simultaneously.

while (boolianL == false)

{

colL--;

if (isalpha(chessPieces[row][colL]) || colL == 0) // If this new column has a piece in it or if the column is 0.

{

if (islower(chessPieces[row][colL]) || colL == 0)

leftMost = colL + 1; // The furthest to the left that the rook can move. (With another black piece next to it.)

else if (isupper(chessPieces[row][colL])) // Rook can capture the piece blocking its path.

leftMost = colL;

boolianL = true; // Once leftMost value is found, boolianL is assigned the value true and loop terminates.

}

}

while (boolianR == false)

{

colR++;

if (isalpha(chessPieces[row][colR]) || colR == 9) // If this new column has a piece in it or if the column is 9.

{

if (islower(chessPieces[row][colR]) || colR == 9)

rightMost = colR - 1; // The furthest to the right that the rook can move. (With another black piece next to it.)

else if (isupper(chessPieces[row][colR])) // Rook can capture the piece blocking its path.

rightMost = colR;

boolianR = true; // Once rightMost value is found, boolianR is assigned the value true and loop terminates.

}

}

if (colNext >= leftMost && colNext <= rightMost) // Now check whether nex[1] (column player wants to move to) is b/t leftMost and rightMost.

validation = true; // If column they want is between the values, then valid move.

}

else if (nex[1] == cur[1] && nex[2] != cur[2]) // If player wants to keep rook in same column, but move to a different row.

{

rowB = row;

rowF = row;

while (boolianF == false)

{

// Decrement row is going up the board - so forward.

rowF--;

if (isalpha(chessPieces[rowF][col]) || rowF == 0) // If this new row has a piece in it or if the row is 0.

{

if (isupper(chessPieces[rowF][col]))

forwardMost = rowF; // The furthest that the rook can move forward. (Toward top of board - closer to row 0 - so smaller value for row.)

else if (islower(chessPieces[rowF][col]) || rowF == 0) // Rook can capture piece blocking its way.

forwardMost = rowF + 1;

boolianF = true; // Once forwardMost value is found, boolianF is assigned the value true and loop terminates.

}

}

while (boolianB == false)

{

rowB++;

if (isalpha(chessPieces[rowB][col]) || rowB == 9) // If this new row has a piece in it or if the row is 9.

{

if (islower(chessPieces[rowB][col]) || rowB == 9)

backMost = rowB - 1; // The furthest that the rook can move backward. (Toward bottom of board - closer to row 9 - so larger value for row.)

else if (isupper(chessPieces[rowB][col])) // Rook can capture piece blocking its way.

backMost = rowB;

boolianB = true; // Once rightMost value is found, boolianR is assigned the value true and loop terminates.

}

}

if (rowNext >= forwardMost && rowNext <= backMost) // Now check whether nex[1] (column player wants to move to) is b/t backMost and forwardMost.

validation = true; // Note: forwardMost is actually smaller row number and backMost is larger row number.

// If column they want is between the values, then valid move.

}

// Need an else if statement for castle move with king. Use a special symbol for castle move. If that symbol is entered, the validation will allow it.

}

else if (cur[0] == 'n')

validation = true;

else if (cur[0] == 'b')

validation = true;

else if (cur[0] == 'q')

validation = true;

else if (cur[0] == 'k')

validation = true;

} // Next are special moves: castling and en passant.

else if (nex[0] == 'c' && nex[1] == 'c' && nex[2] == 'c' && chessPieces[8][1] == 'r' && chessPieces[8][5] == 'k' // Rook and king must be in original positions (and must not have been moved)

&& !isalpha(chessPieces[8][2]) && !isalpha(chessPieces[8][3]) && !isalpha(chessPieces[8][4])) // Cannot be any pieces between rook and king. (Rook on left side of board)

validation = true;

else if (nex[0] == 'c' && nex[1] == 'c' && nex[2] == 'c' && chessPieces[8][8] == 'r' && chessPieces[8][5] == 'k'

&& !isalpha(chessPieces[8][7]) && !isalpha(chessPieces[8][6])) // Cannot be any pieces between rook and king. (Rook on right side of board)

validation = true;

else if (nex[0] == 'p' && nex[1] == 'p' && nex[2] == 'r' && cur[0] == 'a' && cur[2] == '5' && cur[1] == 'a' && chessPieces[4][2] == 'P' // Following else if statements for en passant to right.

&& !isalpha(chessPieces[3][2])) // For en passant, pawn position must be col a-h and row 5. There must be an opponent's pawn in adjacent square.

validation = true; // There must also be no piece immediately behind the opponent's pawn. Also pawn may en passant to left or right.

else if (nex[0] == 'p' && nex[1] == 'p' && nex[2] == 'r' && cur[0] == 'a' && cur[2] == '5' && cur[1] == 'b' && chessPieces[4][3] == 'P'

&& !isalpha(chessPieces[3][3]))

validation = true;

else if (nex[0] == 'p' && nex[1] == 'p' && nex[2] == 'r' && cur[0] == 'a' && cur[2] == '5' && cur[1] == 'c' && chessPieces[4][4] == 'P'

&& !isalpha(chessPieces[3][4]))

validation = true;

else if (nex[0] == 'p' && nex[1] == 'p' && nex[2] == 'r' && cur[0] == 'a' && cur[2] == '5' && cur[1] == 'd' && chessPieces[4][5] == 'P'

&& !isalpha(chessPieces[3][5]))

validation = true;

else if (nex[0] == 'p' && nex[1] == 'p' && nex[2] == 'r' && cur[0] == 'a' && cur[2] == '5' && cur[1] == 'e' && chessPieces[4][6] == 'P'

&& !isalpha(chessPieces[3][6]))

validation = true;

else if (nex[0] == 'p' && nex[1] == 'p' && nex[2] == 'r' && cur[0] == 'a' && cur[2] == '5' && cur[1] == 'f' && chessPieces[4][7] == 'P'

&& !isalpha(chessPieces[3][7]))

validation = true;

else if (nex[0] == 'p' && nex[1] == 'p' && nex[2] == 'r' && cur[0] == 'a' && cur[2] == '5' && cur[1] == 'g' && chessPieces[4][8] == 'P'

&& !isalpha(chessPieces[3][8]))

validation = true;

else if (nex[0] == 'p' && nex[1] == 'p' && nex[2] == 'l' && cur[0] == 'a' && cur[2] == '5' && cur[1] == 'b' && chessPieces[4][1] == 'P' // Following else if statements for en passant to left.

&& !isalpha(chessPieces[3][1]))

validation = true;

else if (nex[0] == 'p' && nex[1] == 'p' && nex[2] == 'l' && cur[0] == 'a' && cur[2] == '5' && cur[1] == 'c' && chessPieces[4][2] == 'P'

&& !isalpha(chessPieces[3][2]))

validation = true;

else if (nex[0] == 'p' && nex[1] == 'p' && nex[2] == 'l' && cur[0] == 'a' && cur[2] == '5' && cur[1] == 'd' && chessPieces[4][3] == 'P'

&& !isalpha(chessPieces[3][3]))

validation = true;

else if (nex[0] == 'p' && nex[1] == 'p' && nex[2] == 'l' && cur[0] == 'a' && cur[2] == '5' && cur[1] == 'e' && chessPieces[4][4] == 'P'

&& !isalpha(chessPieces[3][4]))

validation = true;

else if (nex[0] == 'p' && nex[1] == 'p' && nex[2] == 'l' && cur[0] == 'a' && cur[2] == '5' && cur[1] == 'f' && chessPieces[4][5] == 'P'

&& !isalpha(chessPieces[3][5]))

validation = true;

else if (nex[0] == 'p' && nex[1] == 'p' && nex[2] == 'l' && cur[0] == 'a' && cur[2] == '5' && cur[1] == 'g' && chessPieces[4][6] == 'P'

&& !isalpha(chessPieces[3][6]))

validation = true;

else if (nex[0] == 'p' && nex[1] == 'p' && nex[2] == 'l' && cur[0] == 'a' && cur[2] == '5' && cur[1] == 'h' && chessPieces[4][7] == 'P'

&& !isalpha(chessPieces[3][7]))

validation = true;

else

validation = false;

}

return validation;

}

// All I have to do is update compPiece array and then bring the structure containing it

// back through this function throughout game and every time function is called validMoves array will be updated. Right?

int \*getCompMove(char \*\*piece, CompMoves &computer, int num, int nthMove)

{

const int SIZE = 4;

int rank, // The rows on a chess board are called ranks.

file, // The columns on a chess board are called files.

pieceNumber = 0, // The nth piece that has a valid move in the validMoves array.

numValid = 0, // Number of pieces that have legal moves.

choice, // The randomly selected piece that the computer will move.

positionsArray[SIZE], // Array holding the rows and columns.

\*piecePositions; // pointer storing former and new rows and columns.

bool valid = false;

for (int row = 1; row < 9; row++)

{

for (int col = 1; col < 9; col++)

computer.validMoves[row][col] = '0'; // At beginning of game store a 0 in every element of validMoves 2D array to indicate pieces that cannot be moved.

}

for (int row = 1; row < 9; row++) // All elements of validMoves array were initialized to 0. Now this for loop will find pieces that have moves and store

{ // those pieces in the validMoves array using the subscripts that denote their position.

for (int col = 1; col < 9; col++)

{

if (computer.compPieces[row][col] != '0')

{

valid = findValidMoves(piece, computer, row, col, num);

if (valid == true)

{

computer.validMoves[row][col] = computer.compPieces[row][col];

numValid++; // Increment the number of pieces with valid moves.

}

}

}

}

// Put the positions of the pieces having valid moves into another array. Have the row number stored in

// one element and the column number stored in the next element. This array would have to be dynamically

// allocated because the number of valid moves changes throughout the game. The number of pieces that have

// available moves can be found by counting the number of times the findValidMoves function returns true.

// Then that number will be doubled since the one array must hold the row and column numbers of each piece.

numValid = numValid\*2; // Reassign numValid to twice the value.

computer.validPositions = new int[numValid]; // Dynamically allocate memory to pointer internal to CompMoves structure

// The 0th subscript of validPieces array is the row number of a piece, the 1st is a column number.

// Thus, every even numbered subscript stores a row number and every odd numbered subscript stores a column number.

for (int row = 1; row < 9; row ++)

{

for (int col = 1; col < 9; col++) // Note: validMoves array stores the pieces that the computer can move, and the row and column numbers represent

{ // that specific piece's position on the chess board. I'm taking the row and column numbers of these pieces

if (isalpha(computer.validMoves[row][col])) // and storing them next to each other in a 1D array. Then one of those row or column numbers is randomly

{ // selected and that row or column number will give the other, and the piece can be determined and moved!

computer.validPositions[pieceNumber] = row; // An even subscript in validPieces array stores the row number.

computer.validPositions[pieceNumber + 1] = col; // The next subscript, an odd one, in validPieces array stores the column number.

pieceNumber = pieceNumber + 2;

} // Also, the validMoves array either contains the computer's pieces (whatever they were in the beginning)

} // or a 0. So, all that has to be checked is that it is a letter of the alphabet in if statement.

}

unsigned seed = time(0);

srand(seed);

// Suppose there are 3 pieces on the board with valid moves. Since each piece has a position

// described by a row and a column, there are 2 values associated with each of the 3 pieces.

// This means 6 elements. However, the validPieces array begins with subscript 0, so 0-5 is used

// to store the 6 row and column numbers of the pieces. pieceNumber would hold the number 6 in

// this case. Below, a number between 0 and pieceNumber - 1 is randomly generated and stored

// in the variable choice. This is perfect since choice must be between 0 and 5.

choice = rand() % pieceNumber; // choice stores the row or column number of a valid move.

if (choice % 2 == 0) // choice is even and this element in the validPieces array holds a row number.

{

rank = computer.validPositions[choice]; // The rows on a chess board are called ranks.

file = computer.validPositions[choice + 1]; // Since choice is even, the element holds a row number. The next element in validPieces array must be a column number.

}

else if (choice % 2 == 1) // choice is odd and the element holds a column number.

{

rank = computer.validPositions[choice - 1]; // Since choice odd, element holds a column number. The previous element in the validPieces array must be a row number.

file = computer.validPositions[choice]; // The columns on a chess board are called files. choice is odd, so the element is a column, or file.

}

// Note: By finding rank and file, I've only determined the position of the

// piece that will be moved, but I haven't found out where it will be moved.

piecePositions = positionsArray; // Have pointer point to positionsArray array. QUESTION: Do I have to point to an array to return values stored in it?

piecePositions = moveCompPiece(piece, computer, rank, file, num); // Call function that actually moves piece and displays updated chessboard. It also returns pointer with positions.

delete [] computer.validPositions; // Free the memory.

computer.validPositions = 0;

return piecePositions;

}

bool findValidMoves(char \*\*piece, CompMoves &c, int row, int col, int num)

{

bool validMove = false;

if (c.compPieces[row][col] == 'P' || c.compPieces[row][col] == 'a')

{

if (row == 2 && !isalpha(piece[row + 1][col]) && !isalpha(piece[row + 2][col]) && row + 2 < 9)

validMove = true;

else if (!isalpha(piece[row + 1][col]) && row + 1 < 9)

validMove = true;

else if (num == 1 && isupper(piece[row + 1][col + 1]) && row + 1 < 9 && col + 1 < 9)

validMove = true;

else if (num == 1 && isupper(piece[row + 1][col - 1]) && row + 1 < 9 && col - 1 > 0)

validMove = true;

else if (num == 2 && islower(piece[row + 1][col + 1]))

validMove = true;

else if (num == 2 && islower(piece[row + 1][col - 1]))

validMove = true;

}

else if (c.compPieces[row][col] == 'N' || c.compPieces[row][col] == 'n')

{

if (num == 1 && row - 2 > 0 && col + 1 < 9 && !islower(piece[row - 2][col + 1]))

validMove = true;

else if (num == 1 && row - 2 > 0 && col - 1 > 0 && !islower(piece[row - 2][col - 1]))

validMove = true;

else if (num == 1 && row + 2 < 9 && col - 1 > 0 && !islower(piece[row + 2][col - 1]))

validMove = true;

else if (num == 1 && row + 2 < 9 && col + 1 < 9 && !islower(piece[row + 2][col + 1]))

validMove = true;

else if (num == 1 && row - 1 > 0 && col - 2 > 0 && !islower(piece[row - 1][col - 2]))

validMove = true;

else if (num == 1 && row + 1 < 9 && col - 2 > 0 && !islower(piece[row + 1][col - 2]))

validMove = true;

else if (num == 1 && row - 1 > 0 && col + 2 > 0 && !islower(piece[row - 1][col + 2]))

validMove = true;

else if (num == 1 && row + 1 < 9 && col + 2 > 0 && !islower(piece[row + 1][col + 2]))

validMove = true;

else if (num == 2 && row - 2 > 0 && col + 1 < 9 && !isupper(piece[row - 2][col + 1]))

validMove = true;

else if (num == 2 && row - 2 > 0 && col - 1 > 0 && !isupper(piece[row - 2][col - 1]))

validMove = true;

else if (num == 2 && row + 2 < 9 && col - 1 > 0 && !isupper(piece[row + 2][col - 1]))

validMove = true;

else if (num == 2 && row + 2 < 9 && col + 1 < 9 && !isupper(piece[row + 2][col + 1]))

validMove = true;

else if (num == 2 && row - 1 > 0 && col - 2 > 0 && !isupper(piece[row - 1][col - 2]))

validMove = true;

else if (num == 2 && row + 1 < 9 && col - 2 > 0 && !isupper(piece[row + 1][col - 2]))

validMove = true;

else if (num == 2 && row - 1 > 0 && col + 2 > 0 && !isupper(piece[row - 1][col + 2]))

validMove = true;

else if (num == 2 && row + 1 < 9 && col + 2 > 0 && !isupper(piece[row + 1][col + 2]))

validMove = true;

}

else if (c.compPieces[row][col] == 'K' || c.compPieces[row][col] == 'k' || c.compPieces[row][col] == 'Q' || c.compPieces[row][col] == 'q')

{

// Giving the king and queen same moves for now, just when checking for a valid move. This

// is because if the queen can move even one square in any direction, then there is a valid move.

// The king moves are actually tricky because you'd have to check whether any

// of the opponent's pieces can capture the king in his new position.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

if (num == 1 && row - 1 > 0 && col - 1 > 0 && !islower(piece[row - 1][col - 1]))

validMove = true;

else if (num == 1 && row - 1 > 0 && !islower(piece[row - 1][col]))

validMove = true;

else if (num == 1 && row - 1 > 0 && col + 1 < 9 && !islower(piece[row - 1][col + 1]))

validMove = true;

else if (num == 1 && col - 1 > 0 && !islower(piece[row][col - 1]))

validMove = true;

else if (num == 1 && col + 1 < 9 && !islower(piece[row][col + 1]))

validMove = true;

else if (num == 1 && row + 1 < 9 && col - 1 > 0 && !islower(piece[row + 1][col - 1]))

validMove = true;

else if (num == 1 && row + 1 < 9 && !islower(piece[row + 1][col]))

validMove = true;

else if (num == 1 && row + 1 < 9 && col + 1 < 9 && !islower(piece[row + 1][col + 1]))

validMove = true;

else if (num == 2 && row - 1 > 0 && col - 1 > 0 && !isupper(piece[row - 1][col - 1]))

validMove = true;

else if (num == 2 && row - 1 > 0 && !isupper(piece[row - 1][col]))

validMove = true;

else if (num == 2 && row - 1 > 0 && col + 1 < 9 && !isupper(piece[row - 1][col + 1]))

validMove = true;

else if (num == 2 && col - 1 > 0 && !isupper(piece[row][col - 1]))

validMove = true;

else if (num == 2 && col + 1 < 9 && !isupper(piece[row][col + 1]))

validMove = true;

else if (num == 2 && row + 1 < 9 && col - 1 > 0 && !isupper(piece[row + 1][col - 1]))

validMove = true;

else if (num == 2 && row + 1 < 9 && !isupper(piece[row + 1][col]))

validMove = true;

else if (num == 2 && row + 1 < 9 && col + 1 < 9 && !isupper(piece[row + 1][col + 1]))

validMove = true;

}

else if (c.compPieces[row][col] == 'B' || c.compPieces[row][col] == 'b')

{

if (num == 1 && row - 1 > 0 && col - 1 > 0 && !islower(piece[row - 1][col - 1]))

validMove = true;

else if (num == 1 && row - 1 > 0 && col + 1 < 9 && !islower(piece[row - 1][col + 1]))

validMove = true;

else if (num == 1 && row + 1 < 9 && col + 1 < 9 && !islower(piece[row + 1][col + 1]))

validMove = true;

else if (num == 1 && row + 1 < 9 && col - 1 > 0 && !islower(piece[row + 1][col - 1]))

validMove = true;

else if (num == 2 && row - 1 > 0 && col - 1 > 0 && !isupper(piece[row - 1][col - 1]))

validMove = true;

else if (num == 2 && row - 1 > 0 && col + 1 < 9 && !isupper(piece[row - 1][col + 1]))

validMove = true;

else if (num == 2 && row + 1 < 9 && col - 1 > 0 && !isupper(piece[row + 1][col - 1]))

validMove = true;

else if (num == 2 && row + 1 < 9 && col + 1 < 9 && !isupper(piece[row + 1][col + 1]))

validMove = true;

}

else if (c.compPieces[row][col] == 'R' || c.compPieces[row][col] == 'r')

{

if (num == 1 && row - 1 > 0 && !islower(piece[row - 1][col]))

validMove = true;

else if (num == 1 && row + 1 < 9 && !islower(piece[row + 1][col]))

validMove = true;

else if (num == 1 && col + 1 < 9 && !islower(piece[row][col + 1]))

validMove = true;

else if (num == 1 && col - 1 > 0 && !islower(piece[row][col - 1]))

validMove = true;

else if (num == 2 && row + 1 < 9 && !isupper(piece[row + 1][col]))

validMove = true;

else if (num == 2 && row - 1 > 0 && !isupper(piece[row - 1][col]))

validMove = true;

else if (num == 2 && col + 1 < 9 && !isupper(piece[row][col + 1]))

validMove = true;

else if (num == 2 && col - 1 > 0 && !isupper(piece[row][col - 1]))

validMove = true;

}

// Still need rook to castle.

return validMove;

}

int \*moveCompPiece(char \*\*piece, CompMoves &c, int rank, int file, int num)

{

int moveNum, // The random number generated that chooses a move from the move choices.

newRank, // Row number of piece after move.

newFile; // Column number of piece after move.

// It will point to an address that store the row and column numbers of the original position and new

// position of the piece that was moved.

const int SIZE = 4;

int rowsCols[SIZE]; // rowsCols array stores the former row and column of the piece, and the new row and column of the piece.

int \*positions; // pointer to an int will be returned to getCompMove and will then be returned to newGame function.

positions = rowsCols; // Going to return this pointer to an array.

if (c.validMoves[rank][file] == 'P' || c.validMoves[rank][file] == 'a') // validPieces array is 1D holding rows and columns; validMoves array is 2D and holds the piece in the row/column.

{

// Not randomizing the selection yet because not all moves are valid, but

// definitely one move is valid for this piece if the rank/column were chosen.

//unsigned seed = time(0);

//srand(seed);

//moveNum = 1 + rand() % 6;

if (rank == 2 && !isalpha(piece[rank + 1][file]) && !isalpha(piece[rank + 2][file]))

{

newRank = rank + 2;

newFile = file;

}

else if (!isalpha(piece[rank + 1][file]) && rank + 1 < 9)

{

newRank = rank + 1;

newFile = file;

}

else if (num == 1 && isupper(piece[rank + 1][file + 1]) && rank + 1 < 9 && file + 1 < 9)

{

newRank = rank + 1;

newFile = file + 1;

}

else if (num == 1 && isupper(piece[rank + 1][file - 1]) && rank + 1 < 9 && file - 1 > 0)

{

newRank = rank + 1;

newFile = file - 1;

}

else if (num == 2 && islower(piece[rank + 1][file + 1]))

{

newRank = rank + 1;

newFile = file + 1;

}

else if (num == 2 && islower(piece[rank + 1][file - 1]))

{

newRank = rank + 1;

newFile = file - 1;

}

}

else if (c.validMoves[rank][file] == 'N' || c.validMoves[rank][file] == 'n')

{

if (num == 1 && rank - 2 > 0 && file + 1 < 9 && !islower(piece[rank - 2][file + 1]))

{

newRank = rank - 2;

newFile = file + 1;

}

else if (num == 1 && rank - 2 > 0 && file - 1 > 0 && !islower(piece[rank - 2][file - 1]))

{

newRank = rank - 2;

newFile = file - 1;

}

else if (num == 1 && rank + 2 < 9 && file - 1 > 0 && !islower(piece[rank + 2][file - 1]))

{

newRank = rank + 2;

newFile = file - 1;

}

else if (num == 1 && rank + 2 < 9 && file + 1 < 9 && !islower(piece[rank + 2][file + 1]))

{

newRank = rank + 2;

newFile = file + 1;

}

else if (num == 1 && rank - 1 > 0 && file - 2 > 0 && !islower(piece[rank - 1][file - 2]))

{

newRank = rank - 1;

newFile = file - 2;

}

else if (num == 1 && rank + 1 < 9 && file - 2 > 0 && !islower(piece[rank + 1][file - 2]))

{

newRank = rank + 1;

newFile = file - 2;

}

else if (num == 1 && rank - 1 > 0 && file + 2 > 0 && !islower(piece[rank - 1][file + 2]))

{

newRank = rank - 1;

newFile = file + 2;

}

else if (num == 1 && rank + 1 < 9 && file + 2 > 0 && !islower(piece[rank + 1][file + 2]))

{

newRank = rank + 1;

newFile = file + 2;

}

else if (num == 2 && rank - 2 > 0 && file + 1 < 9 && !isupper(piece[rank - 2][file + 1]))

{

newRank = rank - 2;

newFile = file + 1;

}

else if (num == 2 && rank - 2 > 0 && file - 1 > 0 && !isupper(piece[rank - 2][file - 1]))

{

newRank = rank - 2;

newFile = file - 1;

}

else if (num == 2 && rank + 2 < 9 && file - 1 > 0 && !isupper(piece[rank + 2][file - 1]))

{

newRank = rank + 2;

newFile = file - 1;

}

else if (num == 2 && rank + 2 < 9 && file + 1 < 9 && !isupper(piece[rank + 2][file + 1]))

{

newRank = rank + 2;

newFile = file + 1;

}

else if (num == 2 && rank - 1 > 0 && file - 2 > 0 && !isupper(piece[rank - 1][file - 2]))

{

newRank = rank - 1;

newFile = file - 2;

}

else if (num == 2 && rank + 1 < 9 && file - 2 > 0 && !isupper(piece[rank + 1][file - 2]))

{

newRank = rank + 1;

newFile = file - 2;

}

else if (num == 2 && rank - 1 > 0 && file + 2 > 0 && !isupper(piece[rank - 1][file + 2]))

{

newRank = rank - 1;

newFile = file + 2;

}

else if (num == 2 && rank + 1 < 9 && file + 2 > 0 && !isupper(piece[rank + 1][file + 2]))

{

newRank = rank + 1;

newFile = file + 2;

}

}

else if (c.validMoves[rank][file] == 'K' || c.validMoves[rank][file] == 'k' || c.validMoves[rank][file] == 'Q' || c.validMoves[rank][file] == 'q')

{

// Giving the king and queen same moves for now, just when checking for a valid move. This

// is because if the queen can move even one square in any direction, then there is a valid move.

// The king moves are actually tricky because you'd have to check whether any

// of the opponent's pieces can capture the king in his new position.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

if (num == 1 && rank - 1 > 0 && file - 1 > 0 && !islower(piece[rank - 1][file - 1]))

{

newRank = rank - 1;

newFile = file - 1;

}

else if (num == 1 && rank - 1 > 0 && !islower(piece[rank - 1][file]))

{

newRank = rank - 1;

newFile = file;

}

else if (num == 1 && rank - 1 > 0 && file + 1 < 9 && !islower(piece[rank - 1][file + 1]))

{

newRank = rank - 1;

newFile = file + 1;

}

else if (num == 1 && file - 1 > 0 && !islower(piece[rank][file - 1]))

{

newRank = rank;

newFile = file - 1;

}

else if (num == 1 && file + 1 < 9 && !islower(piece[rank][file + 1]))

{

newRank = rank;

newFile = file + 1;

}

else if (num == 1 && rank + 1 < 9 && file - 1 > 0 && !islower(piece[rank + 1][file - 1]))

{

newRank = rank + 1;

newFile = file - 1;

}

else if (num == 1 && rank + 1 < 9 && !islower(piece[rank + 1][file]))

{

newRank = rank + 1;

newFile = file;

}

else if (num == 1 && rank + 1 < 9 && file + 1 < 9 && !islower(piece[rank + 1][file + 1]))

{

newRank = rank + 1;

newFile = file + 1;

}

else if (num == 2 && rank - 1 > 0 && file - 1 > 0 && !isupper(piece[rank - 1][file - 1]))

{

newRank = rank - 1;

newFile = file - 1;

}

else if (num == 2 && rank - 1 > 0 && !isupper(piece[rank - 1][file]))

{

newRank = rank - 1;

newFile = file;

}

else if (num == 2 && rank - 1 > 0 && file + 1 < 9 && !isupper(piece[rank - 1][file + 1]))

{

newRank = rank - 1;

newFile = file + 1;

}

else if (num == 2 && file - 1 > 0 && !isupper(piece[rank][file - 1]))

{

newRank = rank;

newFile = file - 1;

}

else if (num == 2 && file + 1 < 9 && !isupper(piece[rank][file + 1]))

{

newRank = rank;

newFile = file + 1;

}

else if (num == 2 && rank + 1 < 9 && file - 1 > 0 && !isupper(piece[rank + 1][file - 1]))

{

newRank = rank + 1;

newFile = file - 1;

}

else if (num == 2 && rank + 1 < 9 && !isupper(piece[rank + 1][file]))

{

newRank = rank + 1;

newFile = file;

}

else if (num == 2 && rank + 1 < 9 && file + 1 < 9 && !isupper(piece[rank + 1][file + 1]))

{

newRank = rank + 1;

newFile = file + 1;

}

}

else if (c.validMoves[rank][file] == 'B' || c.validMoves[rank][file] == 'b')

{

if (num == 1 && rank - 1 > 0 && file - 1 > 0 && !islower(piece[rank - 1][file - 1]))

{

newRank = rank - 1;

newFile = file - 1;

}

else if (num == 1 && rank - 1 > 0 && file + 1 < 9 && !islower(piece[rank - 1][file + 1]))

{

newRank = rank - 1;

newFile = file + 1;

}

else if (num == 1 && rank + 1 < 9 && file + 1 < 9 && !islower(piece[rank + 1][file + 1]))

{

newRank = rank + 1;

newFile = file + 1;

}

else if (num == 1 && rank + 1 < 9 && file - 1 > 0 && !islower(piece[rank + 1][file - 1]))

{

newRank = rank + 1;

newFile = file - 1;

}

else if (num == 2 && rank - 1 > 0 && file - 1 > 0 && !isupper(piece[rank - 1][file - 1]))

{

newRank = rank - 1;

newFile = file - 1;

}

else if (num == 2 && rank - 1 > 0 && file + 1 < 9 && !isupper(piece[rank - 1][file + 1]))

{

newRank = rank - 1;

newFile = file + 1;

}

else if (num == 2 && rank + 1 < 9 && file - 1 > 0 && !isupper(piece[rank + 1][file - 1]))

{

newRank = rank + 1;

newFile = file - 1;

}

else if (num == 2 && rank + 1 < 9 && file + 1 < 9 && !isupper(piece[rank + 1][file + 1]))

{

newRank = rank + 1;

newFile = file + 1;

}

}

else if (c.validMoves[rank][file] == 'R' || c.validMoves[rank][file] == 'r')

{

if (num == 1 && rank - 1 > 0 && !islower(piece[rank - 1][file]))

{

newRank = rank - 1;

newFile = file;

}

else if (num == 1 && rank + 1 < 9 && !islower(piece[rank + 1][file]))

{

newRank = rank + 1;

newFile = file;

}

else if (num == 1 && file + 1 < 9 && !islower(piece[rank][file + 1]))

{

newRank = rank;

newFile = file + 1;

}

else if (num == 1 && file - 1 > 0 && !islower(piece[rank][file - 1]))

{

newRank = rank;

newFile = file - 1;

}

else if (num == 2 && rank + 1 < 9 && !isupper(piece[rank + 1][file]))

{

newRank = rank + 1;

newFile = file;

}

else if (num == 2 && rank - 1 > 0 && !isupper(piece[rank - 1][file]))

{

newRank = rank - 1;

newFile = file;

}

else if (num == 2 && file + 1 < 9 && !isupper(piece[rank][file + 1]))

{

newRank = rank;

newFile = file + 1;

}

else if (num == 2 && file - 1 > 0 && !isupper(piece[rank][file - 1]))

{

newRank = rank;

newFile = file - 1;

}

}

// Still need rook to castle.

\*(positions + 0) = rank; // Use pointer notation to fill array with former and new ranks and files.

\*(positions + 1) = file;

\*(positions + 2) = newRank;

\*(positions + 3) = newFile;

return positions;

}