**Project 1**

**<Chess>**

**CSC-17a**

**Name: Mathew Briguglio**

**Date: 06/12/15**

**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 2000 lines

The number of variables: about 60

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 given 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.

**Places with Newer Concepts from Textbook**

Classes: BoardGame.h, FileWriter.h, GamePlay.h, GameSetUp.h, GameWriter.h, CompMoves.h (All Header Files)

Constructor: In .cpp files with same names as header files

Destructor: BoardGame.cpp file line beginning on line 31, deleting major variables.

This Pointer: Used in all cpp files.

Inheritance: GameWriter.h inherits data members from FileWriter class.

Virtual Function: Used in GamePlay.h header file, line 11. GameSetUp.h, line 11

Public Members: In all header files.

Private Members: In all header files.

Protected Members: FileWriter.h, line 13.

**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 "GameSetUp.h"

#include "BoardGame.h"

using namespace std;

// 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 displayMenu(BoardGame\*);

int main()

{

BoardGame\* game = new BoardGame();

if (displayMenu(game) == 0)

return 0;

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

// game->getCheesePieces() 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;

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

cin >> ch;

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

game->getGameWriter()->saveGame(game->getGameSetUp()->getName(), game->getGameSetUp()->getColor(), nthMove); // Call function to save player's name and piece color.

else

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

game->getGameSetUp()->showInstruct(game->getGameSetUp()->getColor()); // 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 (game->getGameSetUp()->getColor() == 1)

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

else if (game->getGameSetUp()->getColor() == 2)

cout << "Move: " << nthMove << "\t\tCurrent Move: " << game->getGameSetUp()->getName();

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

game->setCheesePieces(game->resetBoard(game->getGameSetUp()->getColor())); // 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 ((game->getGameSetUp()->getColor() == 1 && nthMove % 2 == 0) || (game->getGameSetUp()->getColor() == 2 && nthMove % 2 == 1))

{

do

{

do

{

cout << "Current Position: ";

cin >> game->getCurrentAN()[0] >> game->getCurrentAN()[1] >> game->getCurrentAN()[2];

validPos = game->getGamePlay()->validatePos(game->getCheesePieces(), game->getCurrentAN(), game->getGameSetUp()->getColor()); // Call validPos function to determine if the player entered a valid chess piece and a valid position for it.

if (validPos == false && game->getCurrentAN()[0] != 'e')

cout << "Invalid Position!\n";

else if (game->getCurrentAN()[0] == 'e')

return 0;

} while (validPos == false);

do

{

cout << "Move To: ";

cin >> game->getNextAN()[0] >> game->getNextAN()[1] >> game->getNextAN()[2];

if (game->getNextAN()[0] != 'u' && game->getNextAN()[0] != 'e')

{

validMove = game->getGamePlay()->validateMove(game->getCheesePieces(), game->getCurrentAN(), game->getNextAN(), game->getGameSetUp()->getColor());

if (validMove == false)

cout << "Invalid Move!\n";

}

else if (game->getNextAN()[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 >> game->getCurrentAN()[0] >> game->getCurrentAN()[1] >> game->getCurrentAN()[2];

validPos = game->getGamePlay()->validatePos(game->getCheesePieces(),game->getCurrentAN(), game->getGameSetUp()->getColor()); // Call validPos function to validate current position.

if (validPos == false && game->getCurrentAN()[0] != 'e')

cout << "Invalid Position!\n";

else if (game->getCurrentAN()[0] == 'e')

return 0;

} while (validPos == false);

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

cin >> game->getNextAN()[0] >> game->getNextAN()[1] >> game->getNextAN()[2];

validMove = game->getGamePlay()->validateMove(game->getCheesePieces(), game->getCurrentAN(), game->getNextAN(), game->getGameSetUp()->getColor());

if (validMove == false && game->getNextAN()[0] != 'e')

cout << "Invalid Move!\n";

else if (game->getNextAN()[0] == 'e')

return 0;

}

else if (game->getNextAN()[0] == 'e')

return 0;

} 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: " << game->getCurrentAN()[0] << game->getCurrentAN()[1] << game->getCurrentAN()[2] <<

" to " << game->getNextAN()[0] << game->getNextAN()[1] << game->getNextAN()[2] << "\n";

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

cin >> ch;

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

game->getGameWriter()->saveGame(game->getGameSetUp()->getName(), game->getGameSetUp()->getColor(), 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 (game->getNextAN()[0] != 'c' && game->getNextAN()[0] != 'p') // If not a special move, move piece to new position with legal rules.

{

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

{

if (game->getGameSetUp()->getColor() == 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 >> game->getNextAN()[0];

} while (game->getNextAN()[0] != 'N' && game->getNextAN()[0] != 'B' && game->getNextAN()[0] != 'R' && game->getNextAN()[0] != 'Q');

}

else if (game->getGameSetUp()->getColor() == 2)

{

do

{

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

cin >> game->getNextAN()[0];

} while (game->getNextAN()[0] != 'n' && game->getNextAN()[0] != 'b' && game->getNextAN()[0] != 'r' && game->getNextAN()[0] != 'q');

}

}

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

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

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

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

// Update 2D array.

game->getCheesePieces()[newRow][newCol] = game->getNextAN()[0]; // \*\*Very important lines of code - after validating player's move, assign the new position the piece being moved.

game->getCheesePieces()[preRow][preCol] = 32; // \*\*And replace the position where the piece was before with a space.

}

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

{

if (game->getNextAN()[0] == 'c' && game->getCurrentAN()[1] == 'a') // If castling with rook in column a.

{

if (game->getGameSetUp()->getColor() == 1) // If black pieces.

{

game->getCheesePieces()[8][3] = 'K'; // King moves two spaces closer to rook in column a.

game->getCheesePieces()[8][4] = 'R'; // Rook moves one square to right of king.

}

else if (game->getGameSetUp()->getColor() == 2) // If white pieces.

{

game->getCheesePieces()[8][3] = 'k'; // King moves two spaces closer to rook in column a.

game->getCheesePieces()[8][4] = 'r'; // Rook moves one square to right of king.

}

game->getCheesePieces()[8][1] = 32; // Replace king and rook original positions with a space.

game->getCheesePieces()[8][5] = 32;

}

else if (game->getNextAN()[0] == 'c' && game->getCurrentAN()[1] == 'h') // If castling with rook in column h.

{

if (game->getGameSetUp()->getColor() == 1) // If black pieces.

{

game->getCheesePieces()[8][7] = 'K'; // King moves two spaces closer to rook in column h.

game->getCheesePieces()[8][6] = 'R'; // Rook moves one square to left of king.

}

else if (game->getGameSetUp()->getColor() == 2) // If white pieces.

{

game->getCheesePieces()[8][7] = 'k'; // King moves two spaces closer to rook in column h.

game->getCheesePieces()[8][6] = 'r'; // Rook moves one square to left of king.

}

game->getCheesePieces()[8][8] = 32; // Replace king and rook original positions with a space.

game->getCheesePieces()[8][5] = 32;

}

else if (game->getNextAN()[2] == 'l') // Now code for en passant.

{

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

{

if (game->getGameSetUp()->getColor() == 1)

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

else if (game->getGameSetUp()->getColor() == 2)

game->getCheesePieces()[3][1] = 'a'; // Player has white pieces so lowercase a.

game->getCheesePieces() [4][2] = 32; // Player's pawn previous position.

game->getCompMoves().compPieces[4][1] = '0'; // Opponent's pawn position - update compPieces array.

game->getCheesePieces()[4][1] = 32; // Opponent's pawn position - update chessboard.

}

else if (game->getCurrentAN()[1] == 'c')

{

if (game->getGameSetUp()->getColor() == 1)

game->getCheesePieces()[3][2] = 'P';

else if (game->getGameSetUp()->getColor() == 2)

game->getCheesePieces()[3][2] = 'a';

game->getCheesePieces() [4][3] = 32;

game->getCompMoves().compPieces[4][2] = '0';

game->getCheesePieces()[4][2] = 32;

}

else if (game->getCurrentAN()[1] == 'd')

{

if (game->getGameSetUp()->getColor() == 1)

game->getCheesePieces()[3][3] = 'P';

else if (game->getGameSetUp()->getColor() == 2)

game->getCheesePieces()[3][3] = 'a';

game->getCheesePieces() [4][4] = 32;

game->getCompMoves().compPieces[4][3] = '0';

game->getCheesePieces()[4][3] = 32;

}

else if (game->getCurrentAN()[1] == 'e')

{

if (game->getGameSetUp()->getColor() == 1)

game->getCheesePieces()[3][4] = 'P';

else if (game->getGameSetUp()->getColor() == 2)

game->getCheesePieces()[3][4] = 'a';

game->getCheesePieces() [4][5] = 32;

game->getCompMoves().compPieces[4][4] = '0';

game->getCheesePieces()[4][4] = 32;

}

else if (game->getCurrentAN()[1] == 'f')

{

if (game->getGameSetUp()->getColor() == 1)

game->getCheesePieces()[3][5] = 'P';

else if (game->getGameSetUp()->getColor() == 2)

game->getCheesePieces()[3][5] = 'a';

game->getCheesePieces() [4][6] = 32;

game->getCompMoves().compPieces[4][5] = '0';

game->getCheesePieces()[4][5] = 32;

}

else if (game->getCurrentAN()[1] == 'g')

{

if (game->getGameSetUp()->getColor() == 1)

game->getCheesePieces()[3][6] = 'P';

else if (game->getGameSetUp()->getColor() == 2)

game->getCheesePieces()[3][6] = 'a';

game->getCheesePieces() [4][7] = 32;

game->getCompMoves().compPieces[4][6] = '0';

game->getCheesePieces()[4][6] = 32;

}

else if (game->getCurrentAN()[1] == 'h')

{

if (game->getGameSetUp()->getColor() == 1)

game->getCheesePieces()[3][7] = 'P';

else if (game->getGameSetUp()->getColor() == 2)

game->getCheesePieces()[3][7] = 'a';

game->getCheesePieces() [4][8] = 32;

game->getCompMoves().compPieces[4][7] = '0';

game->getCheesePieces()[4][7] = 32;

}

}

else if (game->getNextAN()[2] == 'r')

{

if (game->getCurrentAN()[1] == 'a') // If player's pawn is in column a.

{

if (game->getGameSetUp()->getColor() == 1)

game->getCheesePieces()[3][2] = 'P';

else if (game->getGameSetUp()->getColor() == 2)

game->getCheesePieces()[3][2] = 'a';

game->getCheesePieces()[4][1] = 32;

game->getCompMoves().compPieces[4][2] = '0';

game->getCheesePieces()[4][2] = 32;

}

else if (game->getCurrentAN()[1] == 'b')

{

if (game->getGameSetUp()->getColor() == 1)

game->getCheesePieces()[3][3] = 'P';

else if (game->getGameSetUp()->getColor() == 2)

game->getCheesePieces()[3][3] = 'a';

game->getCheesePieces()[4][2] = 32;

game->getCompMoves().compPieces[4][3] = '0';

game->getCheesePieces()[4][3] = 32;

}

else if (game->getCurrentAN()[1] == 'c')

{

if (game->getGameSetUp()->getColor() == 1)

game->getCheesePieces()[3][4] = 'P';

else if (game->getGameSetUp()->getColor() == 2)

game->getCheesePieces()[3][4] = 'a';

game->getCheesePieces()[4][3] = 32;

game->getCompMoves().compPieces[4][4] = '0';

game->getCheesePieces()[4][4] = 32;

}

else if (game->getCurrentAN()[1] == 'd')

{

if (game->getGameSetUp()->getColor() == 1)

game->getCheesePieces()[3][5] = 'P';

else if (game->getGameSetUp()->getColor() == 2)

game->getCheesePieces()[3][5] = 'a';

game->getCheesePieces()[4][4] = 32;

game->getCompMoves().compPieces[4][5] = '0';

game->getCheesePieces()[4][5] = 32;

}

else if (game->getCurrentAN()[1] == 'e')

{

if (game->getGameSetUp()->getColor() == 1)

game->getCheesePieces()[3][6] = 'P';

else if (game->getGameSetUp()->getColor() == 2)

game->getCheesePieces()[3][6] = 'a';

game->getCheesePieces()[4][5] = 32;

game->getCompMoves().compPieces[4][6] = '0';

game->getCheesePieces()[4][6] = 32;

}

else if (game->getCurrentAN()[1] == 'f')

{

if (game->getGameSetUp()->getColor() == 1)

game->getCheesePieces()[3][7] = 'P';

else if (game->getGameSetUp()->getColor() == 2)

game->getCheesePieces()[3][7] = 'a';

game->getCheesePieces()[4][6] = 32;

game->getCompMoves().compPieces[4][7] = '0';

game->getCheesePieces()[4][7] = 32;

}

else if (game->getCurrentAN()[1] == 'g')

{

if (game->getGameSetUp()->getColor() == 1)

game->getCheesePieces()[3][8] = 'P';

else if (game->getGameSetUp()->getColor() == 2)

game->getCheesePieces()[3][8] = 'a';

game->getCheesePieces()[4][7] = 32;

game->getCompMoves().compPieces[4][8] = '0';

game->getCheesePieces()[4][8] = 32;

}

}

}

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

}

else if ((game->getGameSetUp()->getColor() == 1 && nthMove % 2 == 1) || (game->getGameSetUp()->getColor() == 2 && nthMove % 2 == 0))

{

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

if ((game->getGameSetUp()->getColor() == 1 && nthMove == 1) || (game->getGameSetUp()->getColor() == 2 && nthMove == 2))

{

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

{

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

{

game->getCompMoves().compPieces[row][col] = game->getCheesePieces()[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++)

{

game->getCompMoves().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 (game->getCompMoves().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

game->getCompMoves().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 = game->getGamePlay()->getCompMove(game->getCheesePieces(), game->getCompMoves(), game->getGameSetUp()->getColor(), nthMove); // Call function to determine valid moves the computer can make and to select a move.

//rowsColsPtr = play.getCompMove(chessPieces, computer, setup.getColor(), nthMove);

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.

game->getCompMoves().compPieces[newRow][newCol] = game->getCheesePieces()[preRow][preCol];

game->getCompMoves().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 << game->getCompMoves().compPieces[row][col] << " ";

}

cout << endl;

}

\*/

cout << endl << endl;

// Update chessboard so it can be displayed.

game->getCheesePieces()[newRow][newCol] = game->getCheesePieces()[preRow][preCol];

game->getCheesePieces()[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 (game->getGameSetUp()->getColor() == 1 && nthMove % 2 == 0)

cout << "\tCurrent Move: " << game->getGameSetUp()->getName() << endl;

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

else if (game->getGameSetUp()->getColor() == 2 && nthMove % 2 == 1)

cout << "\tCurrent Move: " << game->getGameSetUp()->getName() << endl;

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

if (game->getGameSetUp()->getColor() == 1 && nthMove % 2 == 1)

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

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

else if (game->getGameSetUp()->getColor() == 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 << game->getCheesePieces()[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();

// delete game->getGameWriter();

delete game;

return 0;

}

// Implement menu function.

int displayMenu(BoardGame\* game)

{

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

int opt, ans; // 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. Load Game\n"

"2. Game Info.\n"

"3. Play Game.\n"

"4. Quit Program\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: game->getGameWriter()->loadGame(); break;

case 2: game->describeGame(); break;

case 3: break;

case 4: ans = 0; break;

}

} while (opt != 3 && opt != 4);

return ans;

}