Team 13 Developer Manual

Version: v 2.0.0

Final Release

University of California, Irvine
EECS 22L

Justin Han
Raymond Yu
Vivek Hatte
Daisuke Otagiri
Hao Ming Chiang

Table of Contents

Glossary

Application Program Interface (API) - a defined set of functions that enables data exchange between software applications. The API determines how programs should interact between the operating system or protocol.

Data Types - a list of variable types each piece, position, and color will represent.

Graphical User Interface (GUI) - an interactive user interface that includes graphical elements such as windows, icons and buttons. GUI enables programs to become user-friendly.

Installation - a method to setup the program to run later

Log file - a file that records the events that occur while the software runs.

Module Diagram - a diagram that displays the structure of the program and the connection between each module of the program

Module Interface - declares global objects (values, type definitions, exception definitions) that a module exports to make available for other modules. Modules in the program can refer to these globals using qualified identifiers.

Program Control Flow - the steps the program will execute in given user input once program opened

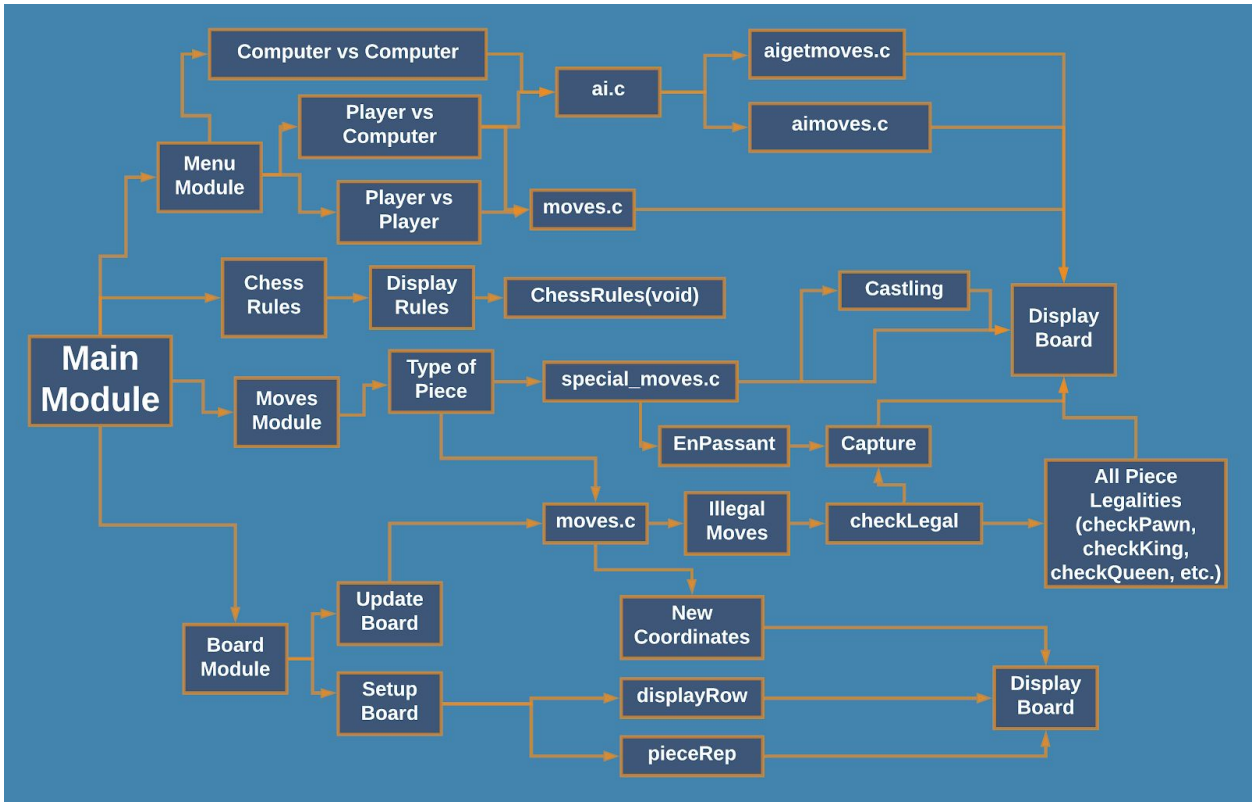
Syntax - a style an event will be formatted in that will be structured into the program

Software Architecture Overview

Main Data Types and Structures

- Main Data Types:
  - Arrays
  - Characters
  - Integers
- Board (2D integer array)
- Column letters (1D integer array)
- Chess piece names (1D integer array)

Major Software Components



(diagram created via *lucidchart.com*)

Modules:

- main
- menu/settings
- Chess Rules
- Moves
- Special moves
- board
- AI_Algorithm

Module Interfaces

```
MOVES structure
{
          int piece;
                int x_old;
                int y_old;
                int board_new[8][8];
                int x_new;
                int y_new;
                int board_value;        //the value of the board in
its current state based on our algorithm
                MOVES *next;            // thje next of the new moves
from the previous board state
                MOVES *prev;            //the next move of the new
moves for the previous board state
                MLIST *list;
}

MLIST structure
{
           MOVES *first;
           MOVES *last;
           int length;
}
```

ai.c

```
      void ai(int board[8][8], int color)
```

aigetmoves.c

```
int board_evaluate(int board[8][8])
int piecevalue(int piece)
int **BoardCopy(int og[8][8], int **new1)
MLIST getmoves(int board[8][8], MLIST *list, int color)
```

aimoves.c

```
void DeleteMove(MOVES *move)
MOVES *CreateMove(void)
void DeleteMovesList(MLIST * list)
MLIST *CreateMovesList(void)
void AppendMove(MLIST *list, MOVES *move)
```

board.c

```
void pieceRep(const int piece, char* representation);

void displayRow(int board[8][8], const int row);

void displayBoard(int board[8][8]);

void chessOutput(const int piece, const char prevChessCol, const
int prevChessRow, const char newChessCol, const int newChessRow);

void write2Log(const int piece, const char prevChesscol, const
int prevChessRow, const char newChessCol, const int newChessRow);
```

conversion.c

```
int convertChess2Array(int* arrayRow, int* arrayCol, char
colLetter, int rowNum);

int checkChessRowBounds(int rowNum);

int checkChessColBounds(char colLetter);

int colLetter2ColIndex(char colLetter);

char colIndex2ColLetter(int colIndex);

int arrayRow2ChessRow(int rowNum);
```

main.c

```
int main(void);
```

menu.c

```
int menu_returned_value (void);
```

ChessRules.c

```
void ChessRules(void);
```

moves.c

```
int move(int board[8][8], int prevRow, int prevCol, int nextRow,
int nextCol);

int checkLegal(int piece,int board[8][8], int prevRow, int
prevCol, int nextRow, int nextCol);

unsigned int newCheckPawn(int board[8][8], const int currRow,
const int currCol, const int nextRow, const int nextCol);

unsigned int checkPawnMovesForward(int piece, const int currRow,
const int currCol, const int nextRow, const int nextCol);

unsigned int checkDoubleStep(int board[8][8], const int currRow,
const int currCol, const int nextRow, const int nextCol);

unsigned int checkSingleStep(int board[8][8], const int currRow,
const int currCol, const int nextRow, const int nextCol);

int checkPawn(int board[8][8], int prevRow, int prevCol, int
nextRow, int nextCol);

int checkRook(int board[8][8], int prevRow, int prevCol, int
nextRow, int nextCol);

int checkBishop(int board[8][8], int prevRow, int prevCol, int
nextRow, int nextCol);
```

```
int checkKnight(int board[8][8], int prevRow, int prevCol, int
nextRow, int nextCol);

int checkQueen(int board[8][8], int prevRow, int prevCol, int
nextRow, int nextCol);

int checkKing(int board[8][8], int prevRow, int prevCol, int
nextRow, int nextCol);

int check4Check(int board[8][8]);

int checkmate(int board[8][8]);

int diagonalDirection(int prevRow, int prevCol, int nextRow, int
nextCol);

int checkDiagonal(int board[8][8], int prevRow, int prevCol, int
nextRow, int nextCol);

int diagonalDownRightBlock(int board[8][8], int prevRow, int
prevCol, int nextRow, int nextCol);

int diagonalDownLeftBlock(int board[8][8], int prevRow, int
prevCol, int nextRow, int nextCol);
int diagonalUpLeftBlock(int board[8][8], int prevRow, int
prevCol, int nextRow, int nextCol);

unsigned int checkAllyOnPos(int board[8][8], int currRow, int
currCol, int nextRow, int nextCol);

unsigned int checkEnemyOnPos(int board[8][8], int currRow, int
currCol, int nextRow, int nextCol);

unsigned int checkFrontalBlockage(int board[8][8], int currRow,
int currCol, int nextRow);
```

special_moves.c

```
int whiteEnPassant(int board[8][8], int prevRow, int prevCol, int
nextRow, int nextCol);

int blackEnPassant(int board[8][8], int prevRow, int prevCol, int
nextRow, int nextCol);

int castleKingSide(int board[8][8], const int currRow, const int
currCol, const int nextRow, const int nextCol);

int castleQueenSide(int board[8][8], const int currRow, const int
currCol, const int nextRow, const int nextCol);

void whitePawnPromotion(int board[8][8]);

void blackPawnPromotion(int board[8][8]);

int choosePiece(int board[8][8], const int row, const int col);

int newPiece(const char piece);
```

pvp.c

```
int playerVSplayer(void);

int conversion_x(char x_value);

int conversion_y(char y_value);
```

pvc.c

```
int playerVScomputer(int menu_returned_value);
```

Setflags.c

```
unsigned int movedWKing(const unsigned int operation);

unsigned int movedWLRook(const unsigned int operation);

unsigned int movedWRRook(const unsigned int operation);

unsigned int movedBKing(const unsigned int operation);

unsigned int movedBLRook(const unsigned int operation);
```

```
unsigned int movedBRRook(const unsigned int operation);
```

Overall Program Control Flow

- Main Menu
    - User chooses game mode (Player vs Player, Player vs AI)
        - If Player vs AI is chosen
            - User chooses what side (Black or White)
        - If Player vs Player is chosen
            - The game starts and the players choose either side that they want to play on
        - If the rules are chosen then the game displays all the rules of the game
- In game
    - Board is displayed with each piece
    - White player moves first
        - Move is chosen with the letter and numbers on the side of the board used as x y coordinates
        - Display new board
            - If Black is in checkmate, White wins
    - Black goes next
        - Move is chosen with the letter and numbers on the side of the board used as x y coordinates
        - Displays new board
            - If White is in checkmate, Black wins
    - Repeat with white moving next

Installation

System Requirements

- Any OS that allows you to SSH into linux servers
- Standard Library (C Programming)
- Math Library
- C programming compiler (e.g. gcc)

Setup and Configuration

- Login to any server on a ssh client (PuTTY.exe, etc)
- Create an empty directory to extract the files in
- Download the tar ball “Chess_V1.0_src.tar.gz”
- Type in the linux server:
  > tar -xvzf Chess_V1.0_src.tar.gz
  > make
  > ./bin/chess

Uninstalling

- Login to any server on a ssh client (PuTTY.exe, etc)
- Go to the directory with the extracted tar ball (Chess_V1.0_src.tar.gz)
- Then type:
  > make clean
  > cd
  > rm -r “your_directory_here”

Documentation of Packages, Modules, Interfaces

Detailed Description of Data Structures

- Arrays
    - int board[8][8] - stores current board state and changes every move. Integers specify the type of piece.

        - 0 - empty space
        - 1 - white rook
        - 2 - white knight
        - 3 - white bishop
        - 4 - white queen
        - 5 - white king
        - 6 - white pawn
        - -1 - black rook
        - -2 - black knight
        - -3 - black bishop
        - -4 - black queen
        - -5 - black king
        - -6 - black pawn

    - char col_letters[8] = {‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’, ‘h’}
        - Character array to symbolize board position
    - char piece_name[6][7] = {“rook”, “knight”, “bishop”, “queen”, “king”, “pawn”} characters to represent the names of the pieces

- Structures
    - We use data structures to implement the ai moves that it could possibly make with all the board states. The two structures that I used are the MOVES and the MLIST structures which are noted up above in the module interface section. The moves holds the a board state, the previous location of the piece you are moving and the final location of the piece you are moving. In addition it has a next and previous pointer to go through the list of moves. There is also a future pointer which checks all the future moves possible from that board state. In addition there is an integer that stores the board value, and this board value basically evaluates the board taking into consideration the positioning of the pieces and the number of pieces for each side to evaluate who is in the lead and who is losing. This is important for the AI so it can make the correct decisions.
    - The MLIST structure is just a list that holds all the values of the moves in a list from first to last and also keeps the length of the list, or the number of moves possible.

    - With these structures and a function to loop through the board to look for all possible moves with it, the AI can chose the correct move going down the list of possible moves with their correlated board evaluation.

Detailed Description of Functions and Parameters

Highlight = Kept but not used.

main.c

- Main function
    - Choose game mode(PVP or PVC) and starts the game
        - 1 for PVP
        - 2 for PVC
        - 3 for CVC
    - Set player color (for player vs computer)
    - Display the chess rules

ChessRules.c

- void ChessRules(void)
    - Just writes the rules out on vim

menu.c

- int menu_returned_value(void)
    - Based on the value that the user inputs, there are different game scenarios. This asks the player to input a different number to create different game scenarios, hence the name on the function. It returns an integer based on the event that the player wants, and that number executes function in order to do what the user wants.

moves.c

- int move(int board[][], int prevRow, int prevCol, int nextRow, int nextCol);
    - Moves a piece from one location on the board to the next
- int checkLegal(int piece, int board[][], int prevRow, int prevCol, int nextRow, int nextCol);
    - Checks if a player's move is legal. This function will be inside the move function
    - Returns 1 if move is legal and 0 if it isn't.
- unsigned int newCheckPawn(int board[8][8], const int currRow, const int currCol, const int nextRow, const int nextCol);
    - More organized version of the checkPawn function made separately in case we run into organization issues with the current checkPawn function.

    - **(Not currently in use by the program.)**
- unsigned int checkPawnMovesForward(int piece, const int currRow, const int currCol, const int nextRow, const int nextCol);
    - Checks that the pawn only moves forward (moves up a row for white or down a row for black).
    - **(Not currently in use by the program.)**
- unsigned int checkDoubleStep(int board[8][8], const int currRow, const int currCol, const int nextRow, const int nextCol);
    - Checks the conditions for making a double step with a pawn.
    - **(Not currently in use by the program.)**
- unsigned int checkSingleStep(int board[8][8], const int currRow, const int currCol, const int nextRow, const int nextCol);
    - Checks the conditions for making a single step with a pawn.
    - **(Not currently in use by the program.)**
- int checkPawn(int board[][], int prevRow, int prevCol, int nextRow, int nextCol);
    - Inside the legal function there will be a check on certain pieces. In this case, the function checks if the pawn is able to move to a certain position on the board given the initial coordinates and checks for pieces in between
    - Returns 1 if move is legal and 0 if it isn't.
- int checkRook(int board[][], int prevRow, int prevCol, int nextRow, int nextCol);
    - Checks if user selected a horizontal and vertical square (no diagonal)
    - Can only move up to blocked square (if ally occupies)
    - Returns 1 if move is legal and 0 if it isn't.
- int checkBishop(int board[][], int prevRow, int prevCol, int nextRow, int nextCol);
    - Checks if player moves diagonally
    - Checks if path is blocked
        - Capture if opponent piece
        - Piece can only move up until blocked space if ally piece
    - Returns 1 if move is legal and 0 if it isn't.
- int checkQueen(int board[][], int prevRow, int prevCol, int nextRow, int nextCol);
    - Checks if the queen move is legal
    - Checks if the path its going is legal by adding all the values of the array in its path to the final location
    - Returns 1 if move is legal and 0 if it isn't.
- int checkKing(int board[][], int prevRow, int prevCol, int nextRow, int nextCol);
    - Only 1 square in every direction
    - Cannot move in blocked space (unless opponent piece)
    - Returns 1 if move is legal and 0 if it isn't.
- int checkKnight(int board[][], int prevRow, int prevCol, int nextRow, int nextCol);

    - Checks if player chose an “L” shape path
    - Doesn’t check if path is blocked
    - Checks if landing square is blocked
    - Returns 1 if move is legal and 0 if it isn’t.
- int check4Check(int board[][]);
    - Checks if the King piece is in check
    - Returns 1 if king is in check and 0 if it isn’t.
- int checkmate(int board[][], int color, char piece);
    - King piece has no legal moves to make
    - No piece can protect the king (by blocking attack/capturing attacking piece)
    - Returns 1 for checkmate and 0 for no checkmate
- int diagonalDirection(int prevRow, int prevCol, int nextRow, int nextCol);
    - Changes the value of the type int direction
    - Value of direction shows legality of diagonal move
    - If direction is 0, move is not diagonal
- int checkDiagonal(int board[8][8], int prevRow, int prevCol, int nextRow, int nextCol);
    - Checks legality of diagonal move by calling other functions
        - diagonalDownRightBlock
        - diagonalDownLeftBlock
        - diagonalUpLeftBlock
        - diagonalUpRightBlock
- int diagonalDownRightBlock(int board[8][8], int prevRow, int prevCol, int nextRow, int nextCol);
    - Checks the space that is directly diagonally bottom right of the selected piece
- int diagonalDownLeftBlock(int board[8][8], int prevRow, int prevCol, int nextRow, int nextCol);
    - Checks the space that is directly diagonally bottom left of the selected piece
- int diagonalUpLeftBlock(int board[8][8], int prevRow, int prevCol, int nextRow, int nextCol);
    - Checks the space that is directly diagonally top left of the selected piece
- int diagonalUpRightBlock(intbaord[8][8], int prevRow, int prevCol, int nextRow, int nextCol);
    - Checks the space that is directly diagonally top right of the selected piece
- int checkHorizontal(int board[8][8], int prevRow, int prevCol, int nextRow, int nextCol);
    - Checks the left and right spaces of the selected piece
    - If there is a piece (ally or not) function returns 0 for the blocked square
- int checkVertical(int board[8][8], int prevRow, int prevCol, int nextRow, int nextCol);
    - Checks the top and bottom squares of the selected piece
    - If there is a piece (ally or not) function returns 0 for the blocked square

- unsigned int checkAllyOnPos(int board[8][8], int currRow, int currCol, int nextRow, int nextCol);
    - Checks if there is an ally piece on the position the user wants to move to.
- unsigned int checkEnemyOnPos(int board[8][8], int currRow, int currCol, int nextRow, int nextCol);
    - Checks if there is an enemy piece on the position the user wants to move to.
- unsigned int checkFrontalBlockage(int board[8][8], int currRow, int currCol, int nextRow);
    - Checks if there is any piece blocking in the positions vertically leading towards the final position. Does not check the final position.

special_moves.c

- int whiteEnPassant(int board[8][8], int prevRow, int prevCol, int nextRow, int nextCol);
    - Takes the value of the previous turn's black pawn's movement
    - Takes the value of the moving white pawn's row and column
    - Returns 1 for legal move if:
        - White pawn is in same row as the black pawn
        - If the diagonal space is empty
- int blackEnPassant(int board[8][8], int prevRow, int prevCol, int nextRow, int nextCol);
    - Takes the value of the previous turn's white pawn's movement
    - Takes the value of the moving black pawn's row and column
    - Returns 1 for legal move if;
        - Black pawn is in same row as the whitepawn
        - If the diagonal space is empty
- int castleKingSide(int board[8][8], const int currRow, const int currCol, const int nextRow, const int nextCol);
    - Checks the current board to see if King or Rook were moved
    - Checks if there are any pieces blocking the King and Rook for castling on the King Side
    - Returns 1 for legal castling move
- int castleQueenSide(int board[8][8], const int currRow, const int currCol, const int nextRow, const int nextCol);
    - Checks the current board to see if King or Rook were moved
    - Checks if there are any pieces blocking the King and Rook for castling on the Queen Side
    - Returns 1 for legal castling move
- void whitePawnPromotion(int board[8][8]);
    - Checks if white pawn reached the last row on the board (row 0 in terms of our array)

    - Calls choosePiece function to replace pawn with piece
- void blackPawnPromotion(int board[8][8]);
    - Checks if black pawn reached the last row on the board (row 0 in terms of our array)
    - Calls choosePiece function to replace pawn with piece
- int choosePiece(int board[8][8], const int row, const int col);
    - Ask user what piece they want to replace the pawn with
    - Return the chosen piece as an integer value (4 for Queen, 3 for Bishop, 1 for Rook, 2 for Knight)
- int newPiece(const char piece);

board.c

- void pieceRep(const int piece, char* representation);
    - Represents a piece (int) with its char representation (Ex. ‘wR’ for 1)
- void displayRow(int board[8][8], const int row);
    - Prints out an individual row. (Called by displayBoard)
- void displayBoard(int board[8][8]);
    - Prints out the entire board with formatting.
- void chessOutput(const int piece, const char prevChessCol, const int prevChessRow, const char newChessCol, const int newChessRow);
    - Prints out the previous move and calls write2Log.
- void write2Log(const int piece, const char prevChessCol, const int prevChessRow, const char newChessCol, const int newChessRow);
    - Writes the previous move to the log text file.

Conversion.c

- Int convertChess2Array(int* arrayRow, int* arrayCol, char colLetter, int rowNum);
    - Converts the array row and columns to the chess row and columns
- int checkChessRowBounds(int rowNum);
    - Checks if the row is in bounds with the user inputs
- int checkChessColBounds(char colLetter);
    - Does the same thing as checkrowbounds but for the column
- int colLetter2ColIndex(char colLetter);
    - Converts the letter version from the chessboard input to the number value
- char colIndex2ColLetter(int colIndex);
    - Does the opposite of the char to int converted for the chess board values
- int arrayRow2ChessRow(int rowNum);

    - Converts the row that is represented on the board into the array that is represented in the computer since the index starts from zero in a computer and the index on the board starts at one.

ai.c

- void ai(int board[8][8], int color)
    - Takes in the current board and the color that the ai would be playing for. If color is negative one it plays for black and if it is one it plays for white. The ai then finds all the moves that are possible through another function and creates a list of them. It is a list of structures with bored values that evaluate which side is winning. The ai then chooses the move that would give the side it is playing the highest net gain in tempo for winning.

aigetmoves.c

- int board_evaluate(int board[8][8])
    - A function in order to see the value of the to check which side has the favor. It is used in the evaluation of what moves the AI will make. It takes into account check and checkmate scenarios, piece positioning, and the number of pieces on the board for each side.
- int piecevalue(int piece)
    - Each piece is represented by a number on the board, however that number is not it's true value. For example the rook is represented by a 1 and the pawn is represented by a 6. However this is not true since the pawn is not more valuable than a rook. So this reevaluates a piece's value and puts in the algorithm to make the correct decisions.
- int **BoardCopy(int og[8][8], int **new)
    - This function just copies the board in a new location so that the function will always have a copy of the original board before it was altered

aimoves.c

- void DeleteMove(MOVES *move)
    - The move is struct with data in it so this function just frees up that space
- MOVES *CreateMove(void)
    - This allocates the memory for a move structure
- void DeleteMovesList(MLIST * list)
    - Frees up the memory allocated for a MLIST structure
- MLIST *CreateMovesList(void)
    - This allocates the memory for a MLIST structure
- void AppendMove(MLIST *list, MOVES *move)
    - Takes the move structure and puts it in a list of the moves

pvc.c

- Int playerVScomputer(int main_returned_value)
    - Initiales the player vs computer game. Sets the turns and calls the function for the ai to move depending on what color it is playing for. Same as pvp, however when it is the turn that is the color that PC is playing for, it will make the move based on the ai functions created in another module.

pvp.c

- void getAndConvertChessPos(char* description, int* arrayRow, int* arrayCol);
    - Takes in the position written in by the player with the indexes that are given in the input to make the moves and converts it to the indexes that are to move the values in the array
- int playerVSplayer(void);
    - The actual function that calls each turn to move the pieces. It takes in the x y coordinates from the board and checks all the parameters that are possible for each move. Other than that it lets the players play the game
- int conversion_x(char x_value);
- Converts the input character value to that is the board index to a integer array index for the computer to use
- int conversion_y(char y_value);
- Does the same as the conversion_x but just since the y is from 1 to 8 is just makes it go from 0 - 7

SetFlags.c:

- All the functions do the same thing and they just see if a piece has been moved

Detailed Description of Input and Output Formats

Input:

- Syntax/Format of a Move Input by the User
    - The move inputted by the user is based on typing in the start and final locations of the piece that you want to move with x and y coordinates based on the chess board x and y.
    - After each legal move initial and final location chosen, it will move the piece on the board and the other person will be asked to do the same

Output:

- Syntax/Format of a Move Output by the User
    - First prints out the board to display changes to the game state.
    - Then, program prints out which piece moved from what initial position to what final position.

- Syntax/Format of a Move Recorded in the Log File
    - The recorded log file is based on the chess coordinates at which the piece is moved to and from

      Display of the Log Example
      wQ e3 -> e4
      bK g5 -> d4

      wP a2 -> a4
      bN b8 -> c6

Development Plan and Timeline

Partitioning of Tasks

1. Structure: Creation of Board and Chess Pieces
2. Gameplay: Movement, capture, check and checkmate
3. AI: Computer generated moves
4. UI : Menu Interfaces (unfinished)

Team Member Responsibilities

Justin Han - main.c, pvc.c,
Raymond Yu - board.c, special_moves.c (En Passant, Pawn Promotion), conversion.c
Vivek Hatte - menu.c, ai.c
Daisuke Otagiri - moves.c, special_moves.c
Hao Ming Chiang - moves.c, special_moves.c

Copyright Notice

Published by Team 13 members:
Justin Han, Raymond Yu, Vivek Hatte, Daisuke Otagiri, Hao Ming Chiang

References

U.S. Chess Federation. “Let’s Play Chess. A Summary of the Moves of Chess.” 2004. PDF File

Index