Kingdoms!

The Chess Game

Version 1.0

PROGRAMMERS:

Martino Caldarella,

Hanqi Chen,

Ragui Halim,

Hoon Lee,

Jacobis Soriano,

Daniel Truong

BitBots Software Developers from UCI

Table of contents:

Glossary:

- **Artificial Intelligence (AI):** the simulation of human intelligence by computer.
- **User Interface (UI):** the means for the user to interact with the computer.
- **Data Structure:** particular method to store and organize data in a computer in order to access and modify it more efficiently.
- **Input/Output (I/O):** describes the transfer of data to/from a computer.
- **Data Type:** the classification of data (Ex: int, char, and float).
- **Function:** unique section of a program coded to do a particular task.
- **Parameter:** value passed to a function.
- **Module:** a piece of a program that contains at least one routine.
- **Bishop**: chess piece, refer to "How to Play Chess" to know its moves.
- **Castling**: a special move that may only be used once, performed when moving the King and the Rook simultaneously, refer to "How to Play Chess" to know more
- **Check**: occurs when the king is under threat. In this case, the king has to move or another piece has to be put between the king and the threatening piece.
- **Checkmate**: occurs when the king cannot escape the check. As a result the game ends and the other player wins.
- **Stalemate:** when the game ends because one team has no legal moves, and his king is not in check position.
- **Chess**: a game for two players where one player tries to capture the opponent's king.
- **En Passant**: a special move that occurs when one player moves their pawn forward two tiles forward to avoid capture by the opponent's pawn, refer to "How to Play Chess" to know more.
- **Files**: columns of the board labeled by the letters *a* to *h*.
- **Knight**: chess piece, refer to "How to Play Chess" to know its moves.
- **King**: the most important chess piece, refer to "How to Play Chess" to know its moves.
- **Pawn**: chess piece, refer to "How to Play Chess" to know its moves.
- **Piece**: any of the 32 movable objects used in the chess game.
- **Queen**: the most powerful chess piece, refer to "How to Play Chess" to know its moves.
- **Ranks**: rows of the board labeled by the numbers 1 to 8.
- **Rook**: chess piece, refer to "How to Play Chess" to know its moves

1. Software Architecture Overview

1.1. Main Data Type and Structures

Typedef enums are used to define the key elements of the game.

- **bool** defines a true and false values
- **type** defines who is making the move (human or computer)
- **color** defines the color (black or white)
- **pieceName** defines the piece (King, Queen, Bishop, Knight, Rook, Pawn), and its given value

Typedef structs are to define other important elements of the game.

- **piece -** data structure with all the information about the piece such as name, color, and whether it was captured or not
- **piece_name** - data struct that initializes integer values to different pieces of a chessboard
- **square** - data structure that describes one square of the chess board. Its elements are its coordinates, the pieces that occupies it, and the color of that piece
- **move** - data structure used to move the pieces. Its element are the initial and final coordinates of the piece
- **des** - data structure used to represents the destination square of a possible move
- **player -** data structure to represent the player of the game. Its elements are the player's type, and the color of the player's pieces.
- **Player_type** - data struct to represent the player type (human or computer)
- **movesEntry & movesList -** Double linked list used to save moves.

Arrays are used to represent the board and the maximum number of possible moves of each piece （use to determine valid move of the piece）

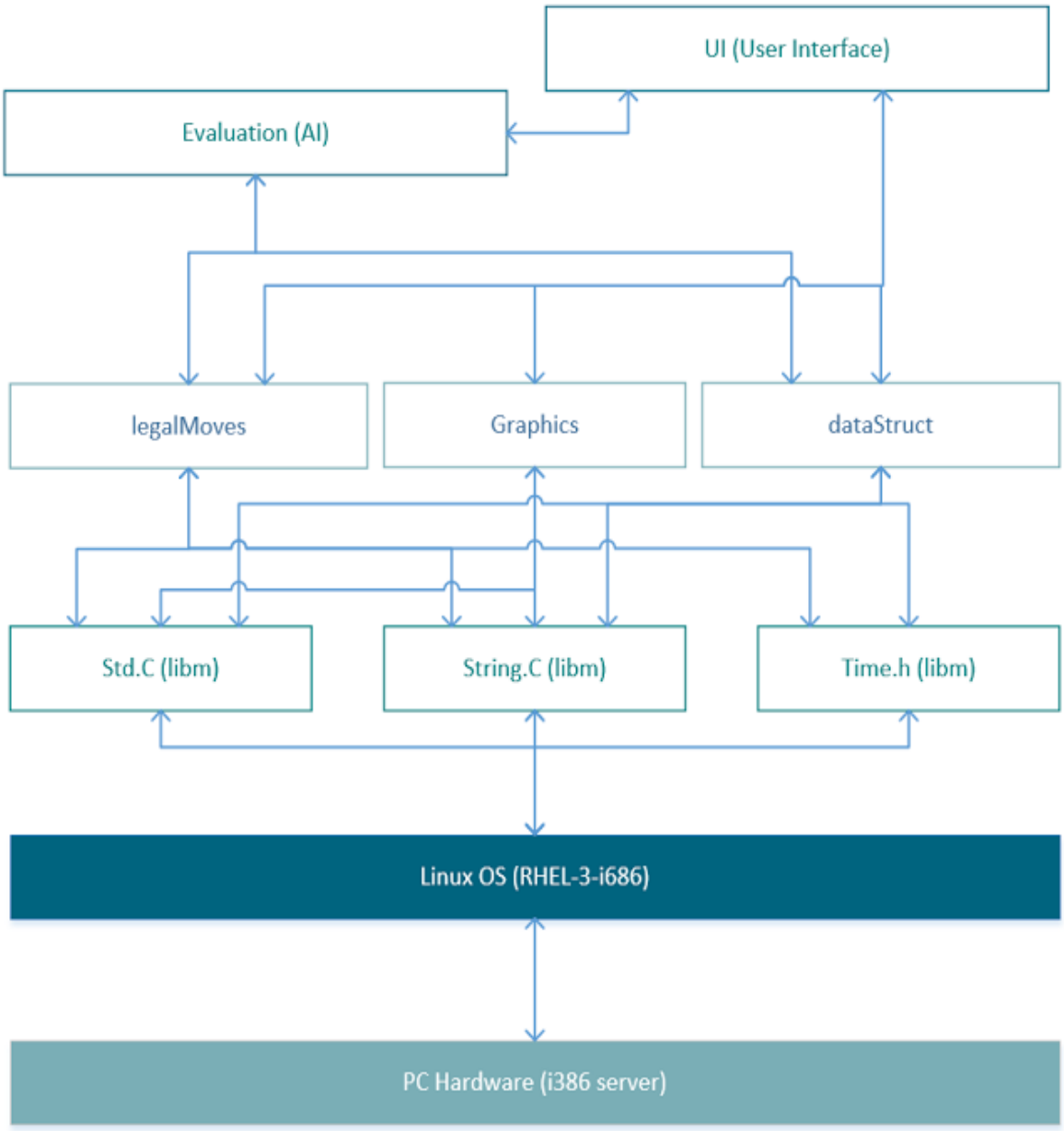
- **board** - 2d array of type *square* with size 9x9
- **king** - array of type *des* with size 8
- **queen** - array of type *des* with size 27
- **bishop** - array of type *des* with size 13
- **knight** - array of type *des* with size 8
- **rook -** array of type *des* with size 14
- **pawn** - array of type *des* with size 4
- **player1/player2** - 2 objects of type player

- **PawnValue -** 2d array of type int
- **knightValue** - 2d array of type int
- **bishopValue -** 2d array of type int
- **kingValue -** 2d array of type int
- **allMoves -** 2d array of type move with size 10x140

Data Types are used to determine if certain events have occurred (aka flags).

- **bool *Wki_Moved -** Data with type bool. Keeps in track whether white king has moved or not.
- **bool *Bki_Moved -** Data with type bool. Keeps in track whether black king has moved or not.
- **bool *Wro1_Moved -** Data with type bool. Keeps in track whether white rook has moved or not.
- **bool *Wro2_Moved -** Data with type bool. Keeps in track whether white rook has moved or not.
- **bool *Bro1_Moved -** Data with type bool. Keeps in track whether black rook has moved or not.
- **bool *Bro2_Moved -** Data with type bool. Keeps in track whether black rook has moved or not.

1.2. Major software components



software modules architecture

1.3 Module interface

- **`UI` Module**
  - **`int main()`**
    - **Input:** The function does not need any input.

        - **Output:** It returns zero at the end of the program.
        - **Description:** This function takes care of the game flow, call all the implemented functions, and organize the order of the program. The game starts and ends here.
    - **`char *printIntro()`**
        - **Input:** This function accepts no input.
        - **Output:** This function returns a string that contains the intro and instructions of the game.
        - **Description:** This function prints on screen the intro to the game and instructions on how to input moves.
    - **`int chooseMode()`**
        - **Input:** This function accepts no input.
        - **Output:** It returns 1 for HUMAN vs COMPUTER mode or 2 for HUMAN vs. HUMAN mode.
        - **Description:** This functions prompts the user for game as follows: HUMAN vs. COMPUTER or HUMAN vs. HUMAN.
    - **`void setPlayersColor()`**
        - **Description:** This function asks the user to enter the color of his pieces, and initialize a player object.
    - **`void deleteList(movesList *l)`**
        - **Input:** a list of moves
        - **Description:** the function deletes the given list (frees the allocated memory)
    - **void delay(int numSeconds)**
        - **Input:** an integer
        - **Description:** This function delays output to screen for a specified amount of seconds.
    - **void toUpper(char str[])**
        - **Input:** a string with type character
        - **Description**:
    - **bool validInput(char in[])**
        - **Input:** a string with type character
        - **Output:** a boolean type value
        - **Description:** This function determines whether User entered a valid input.
    - **void updateKings(int *wx,int *wy,int *bx, int *by,move *m)**
        - **Input:** 4 pointers that points to type int data
        - **Description:** a pointer that points to type move data
    - **void updateKingsUndo(int *wx,int *wy,int *bx, int *by,move *m)**

        - Input: 4 pointers that points to a type int data
        - Description: a pointer that points to type
    - **void krMoved(move *m)**
        - **Input:** a pointer to type move data
        - **Description:** This function determines whether King or Rook has moved.

- **`dataStruct` Module**
    - **`move stringToMove(char *s)`**
        - **Input:** a 4 character string
        - **Output:** move object
        - **Description:** the function return a move
    - **`void initializeBoard()`**
        - **Description:** The function creates 32 different pieces and put them on the board in the initial positions.
    - **movesEntry *NewMovesEntry(move *m)**
        - **Input:** pointer to a move
        - **Output:** a pointer to a movesEntry list
        - **Description:** This function creates a movesEntry list, and initialize it with move m.
    - **void DeleteMoveEntry(movesEntry *e)**
        - **Input:** pointer to a movesEntry
        - **Description:** This function deletes a given moveEntry from the movesList.
    - **movesList *CreateMovesList(void)**
        - **Output:** a pointer to a movesList
        - **Description:** This function creates a movesList.
    - **void DeleteMovesList(movesList *aList)**
        - **Description:** This function deletes movesList.
    - **void DeleteMoveEntry(movesEntry *entry)**
        - **Input:** a pointer to a move.
        - **Description:** This function frees the allocated memory for amove.
    - **void AddMoveToMoveList(movesList *list, move *aMove)**
        - **Input:** pointer to a moveList, and a pointer to a move
        - **Description:** This function add a move to moveList.
    - **void SaveMovesListToFile(const char *saveFile, moveList *gameList)**
        - **Input:** pointer to saveFile, and a pointer to gameList

        - **Description:** This function saves all the moves contained in moveList into file pointed by saveFile.
    - **void LoadGameFile(const char *lfile, movesLIst *list)**
        - **Input:** pointer to file name to be read from
        - **Description:** This functions from a txt file the name(s) of the players, adds moves to a movesList
    - **Void SaveMovesListToFile(const char *lfile, movesList *list)**
        - **Input:** file name to write to, moves list to read from and write to file
        - **Description:** This function will save a file with a the moves of an array
    - **void PrintMoveList(movesList *list)**
        - **Input:** pointer to a list
        - **Description:** This function prints all the moves contained in moveList.
    - **void PrintMove(move *s)**
        - **Input:** pointer to a move
        - **Description:** This function prints the move pointed by s.
    - **move *undo(movesList *list)**
        - **Input:** a pointer to a list
        - **Output:** a pointer to move
        - **Description**: This function undos a move.
    - **void updateBoard(move *m)**
        - **Input:** a pointer to a move
        - **Description:** This function updates the board with the new locations of the pieces.
    - **`pieceName Promotion(type t)`**
        - **Input:** player type
        - **Output:** Piece name
        - **Description:** This function asks the user to specify the piece s/he wants in case of promotion.
- **`Graphics` Module**
    - **`void printBoard()`**
        - **Description:** This functions prints out the layout of the Chessboard as well as the Chess pieces after each move.
    - **`char * printPiece(piece p)`**
        - **Input:** a piece object
        - **Output:** a string which represents each piece (i.e. bKi which means the black king)

        - **Description:** This function returns the representing string of a given piece.
- **`legalMoves` Module**
    - **`bool legalForUser(move m)`**
        - **Input:** a move object
        - **Output:** outputs a boolean value: 0 meaning illegal or 1 meaning legal.
        - **Description:** This functions checks if a move is legal given a list of moves.
    - **`void kingLegalMoves(des *arr, int *n, square s)`**
        - **Input:** a pointer to an array of des, a pointer to n (the number of legal moves found), and a square to determine the position of the piece.
        - **Description:** This functions calculates all legal and possible moves of the king and puts them in an array “arr”.
    - **`void queenLegalMoves(des *arr, int *n, square s)`**
        - **Input:** a pointer to an array of des, a pointer to n (the number of legal moves found), and a square to determine the position of the piece.
        - **Description:** This functions calculates all legal and possible moves of the queen and puts them in an array “arr”.
    - **`void rookLegalMoves(des *arr, int *n, square s)`**
        - **Input:** a pointer to an array of des, a pointer to n (the number of legal moves found), and a square to determine the position of the piece.
        - **Description:** This functions calculates all legal and possible moves of the rook and puts them in an array “arr”.
    - **`void knightLegalMoves(des *arr, int *n, square s)`**
        - **Input:** a pointer to an array of des, a pointer to n (the number of legal moves found), and a square to determine the position of the piece.
        - **Description:** This functions calculates all legal and possible moves of the knight and puts them in an array “arr”.
    - **`void bishopLegalMoves(des *arr, int *n, square s)`**
        - a pointer to an array of des, a pointer to n (the number of legal moves found), and a square to determine the position of the piece.
        - **Description:** This functions calculates all legal and possible moves of the bishop and puts them in an array “arr”.

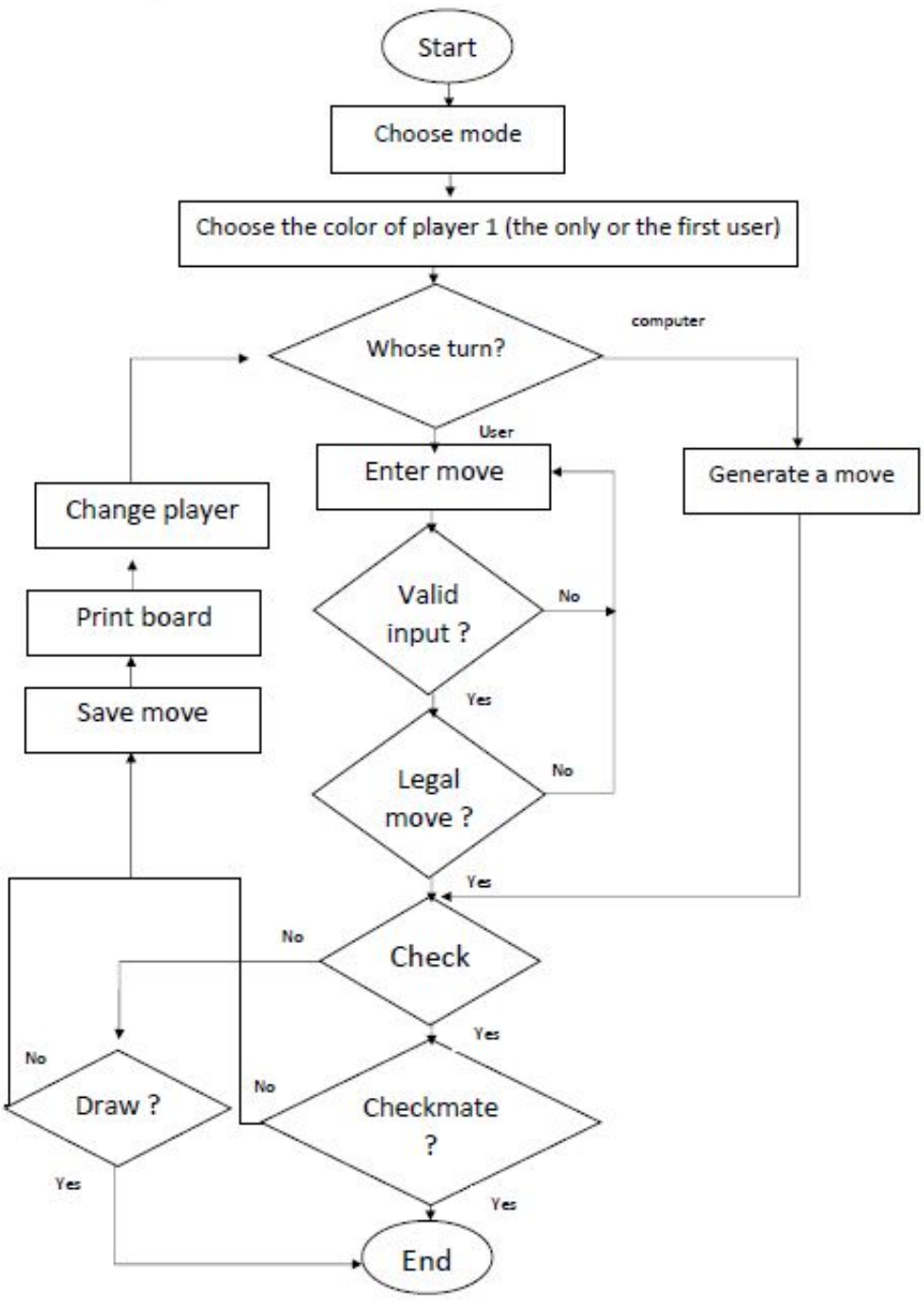
- **`void pawnLegalMoves(des *arr, int *n, square s)`**
  - **Input:** a pointer to an array of des, a pointer to n (the number of legal moves found), and a square to determine the position of the piece.
  - **Description:** This functions calculates all legal and possible moves of the pawn and puts them in an array “arr”.
- **`void EnPassant(piece p, piece p2)`**
  - **Input:** two pieces of type piece
  - **Description:** This function checks to see if a piece a chess piece is in EnPassant with another piece
  - **Return:** returns true if EnPassant
- **`bool kingInCheck(square s)`**
  - **Input:** the position of the king
  - **Output:** 0(False) or 1(True)
  - **Description:** This function checks whether white/black king is under threat of capture on an opponent’s next turn.
- **`bool Checkmate(square s)`**
  - **Input:** the position of the king
  - **Output:** 0(False) or 1(True)
  - **Description:** If white/black king is in check and has no legal move to be made, this function returns 1.
- **`bool draw()`**
  - **Output:** 0(False) or 1(True)
  - **Description:** If the game reaches to meet any of the draw conditions, returns 1.
- **`bool insuffmaterial()`**
  - **Output:** 0(False) or 1(True)
  - **Description:** If there is no sufficient pieces to finish the game, this function returns true. It returns false otherwise.
- **`bool threeRepetition(movesList *s)`**
  - **Output:** 0(False) or 1(True)
  - **Description:** This function returns true if a move is repeated three times. It returns false otherwise. In this case, the user will be asked to draw or not.
- **`bool stalemate()`**
  - **Output:** 0(False) or 1(True)
  - **Description:** This function returns true if a king is not in the check but a players has no legal moves to be made by any

pieces. The game will be called to draw if stalemate returns true.

- **int generateMoves(color r, square ws, square bs, movesList *, int depth)**
  - **Input:** r of type color
  - **Output:** an integer
  - **Description:** This function generates all moves for AI.
- **int alphaBeta(color r, int alpha, int beta, int depth, int i, movesList *l, square ws, square bs)**
  - **Input:** color, the current position in integers and two squares
  - **Output:** an integer corresponding to the best value move
  - **Description:** This function calculates the best move for AI and returns an integer corresponding to best value move.

- **Evaluation Module**
  - **int evaluatePiece(square s)**
    - **Input:** a square struct containing a piece
    - **Output:** a value of the piece
    - **Description:** This function calculates the value of the piece contained in square struct and returns that value.
  - **int evaluateBoard(color r)**
    - **Input:** a color of a piece
    - **Output:** returns a score
    - **Description:** This function calculates the total score given the layout of the chessboard.

1.4 Overall program control flow



software flowchart

2. Installation

2.1. System requirements, Compatibility

- Linux OS 2.6.32-696.18.7.el6.x86_64
- PC(x86_64 Server)

2.2. Setup and Configuration

- No setup or configuration is needed.

2.3 Building, Compilation, and Installation

- The Following steps are required for proper installation:
  1. On the terminal window, extract file using the following command:
     - "`gtar xvzf Chess_Alpha_src.tar.gz`"
  2. Compile the game by entering the following command:
     - type "`make`" to compile the game.
  3. Once compilation is done, enter:
     - "`./bin/chess`" to start the game.

3. Documentation of packages, modules, interfaces

3.1. Data Structures.

- **`piece`** - struct with the following elements
  - **`color r`** - variable that stores the color of the piece
  - **`pieceName name`** - variable that stores the type (King, Queen...)
  - Code:

```
typedef struct Piece{
            color r;
            pieceName name;

}piece;
```

- **`square`** - struct with the following elements
  - **`char x`** - variables that stores the File of the square
  - **`int y`** - variable that stores the Rank of the square
  - **`color r`** - variable that stores the color of the piece on that position
  - **`piece p`** - variable that stores the piece at that position
  - Code:

```
typedef struct Square{
          char x;
          int y;
          color r;
          piece p;
}square;
```

- **move** - struct with the following elements
  - **Int x1, x2** - initial and final File position
  - **int y1, y2** - initial and final Rank position
  - Code:

```
typedef struct Move{
       int x1, x2;
       int y1, y2;
}move;
```

- **des** - struct with the following elements
  - **int x** - File position
  - **int y** - Rank position
  - Code:

```
typedef struct destination{
          int x;
          int y;
}des;
```

- **player** - struct with the following elements
  - **type t** - The type of the player (computer or human)
  - **color r** - The color of the player's pieces
  - Code:

```
typedef struct Player{
     type t;
     color r;
}player;
```

- **movesEntry & movesList -** Double linked list used to save moves
  - Code:

```
typedef struct mEntry movesEntry;

typedef struct mList{
     int length;
     movesEntry *first;
     movesEntry *last;
```

```
}movesList;

struct mEntry{
     move *m;
     movesList list;
     movesEntry *next;
     movesEntry *prev;
};
```

- **square board[9][9] -** two dimensional array of type *square* used to represent the board
- **des king[10] -** array of type *des* used to store the possible moves of the king
- **des queen[27] -** array of type *des* used to store the possible moves of the queen
- **des bishop[13] -** array of type *des* used to store the possible moves of the bishop
- **des knight[8] -** array of type *des* used to store the possible moves of the knight
- **des rook[14] -** array of type *des* used to store the possible moves of the rookt
- **des pawn[4]-** array of type *des* used to store the possible moves of the pawn
- **player *player1 -** a player object used to describe the first player
- **player *player2** - a player object used to describe the second player
- **move allMoves[10][140] -** array of type move used to store all moves for AI.

3.2. Function and Parameters

- **move stringToMove(char *s)** - this function takes the input from the user and translates to two different coordinates that represents the initial position of the piece and the eventual final position. It requires a pointer to a string as parameter and returns a type *move*.
- **void initializeBoard()** - The function creates 32 different pieces and put them on the board in the initial positions.
- **bool legalForUser(move *m)-** this function checks if the move entered is legal allowed or not. It requires move as parameter and returns a type *bool* (true/false)
- **move bestMoveForAI(color r, square ws, square bs)** - this function returns the best move for the computer player.
- **void kingLegalMoves(des *arr, int *n, square s)** - This function generates legal moves for a king chess piece.

- **`void queenLegalMoves(des *arr, int *n, square s)`** - This function generates legal moves for a queen.
- **`void rookLegalMoves(des *arr, int *n, square s)`** - This function generates legal moves for a rook.
- **`void knightLegalMoves(des *arr, int *n, square s)`** - This function generates legal moves for a Knight.
- **`void pawnLegalMoves(des *arr, int *n, square s)`** - This function generates legal moves for a pawn.
- **`void bishopLegalMoves(des *arr, int *n. square s)`** - these functions calculate the number of legal moves based on the number of possible moves of each piece. They follow the chess rules and every piece has its own function. The functions know the position of the piece given the square “s”. The functions return the number of possible moves found “n”, and save them in the array “arr”. The en passant and promotion moves are included in the pawn’s function. The castaling is included in the king’s function.
- **`bool kingInCheck(square s)`** - the function checks if the King is in check. It returns a type *bool* (true/false)
- **`bool Checkmate()`** - the function checks if the King is in checkmate and therefore end the game. It returns a type *bool* (true/false)
- **`bool draw()`** - This function is used to check whether the game has met any conditions for the draw.Conditions for the draw as follows:
    - Insufficient material to finish the game. Ex)Bishop or Knight vs. King, King vs King
    - Threefold Repetition: If entire position of the board repeated three times consecutively, the game is called to draw.
    - Stalemate: When a player is not in check but has no valid moves.
- **`char *printIntro()`** - This function prints on screen the intro to the game and instructions on how to input moves.
- **`int chooseMode()`** - This functions asks the user what game mode s/he would like to play: HUMAN vs. COMPUTER or HUMAN vs. HUMAN.
- **`void setPlayersColor()`** - This function asks the user to enter the color of his pieces, and initialize a player object.
- **`void AddMovetoMoveList(movesList *list, move *aMove)`** - the function saves the given move in a double linked list.
- **`void DeleteMovesList(movesList *aList)`** - the function deletes the move list (frees the allocated memory)
- **`movesEntry *NewMovesEntry(move *m)`** - This function creates a movesEntry list, and initialize it with move m.
- **`void DeleteMoveEntry(movesEntry *e)`** - This function deletes a given movesEntry from the movesList.
- **`movesList *CreateMovesList(void)`** - This function creates a new movesList.

- **`void AddMoveToMoveList(movesList *list, move *aMove)`** - This function add a given move to a given moveList.
- **`void PrintMovesList(movesList *list)`** - This function prints out all the moves saved in movesList.
- **`Void PrintMove(move *s)`** - This function prints out the contents of s or the move.
- **`void SaveMovesListToFile(const char *saveFile, movesList *gameList)`** - This function saves all the moves into the file specified by saveFile.
- **`Void updateBoard(move *m)`** - This function updates the board along with the new piece position.
- **`void printBoard()`** - This functions prints out the layout of the Chessboard as well as the Chess pieces after each move.
- **`char *printPiece(piece p)`** - This function returns the representing string of a given piece.
- **`bool insuffmaterial()`** - This function returns true if a players has no material to checkmate the king. Ex) knight vs king, bishop vs king, king vs king.
- **`bool threeRepetition(moveList *s)`** - This function returns true if both players made three consecutive repeated moves. Players will be asked to end the game as draw.
- **`bool stalemate(square s)`** - This function returns true if a player has no legal move and a king is not in check.
- **`pieceName promotion()`** - This function promotes a pawn to either rook, bishop, queen, or a knight based on a player's choice.
- **`bool EnPassant(piece p, piece p2)`** - This function determines whether pawn can do en passant or not.
- **`int chooseMode()`** - This function allows the User to choose which gamemode s/he would to play in.
- **`updateKings(int *wx,int *wy,int *bx,int *by,move *m)`**
- **`void krMoved(move *m)`** - This function determines either the King or Rook has moved yet.
- **`void toUpper(char str[])`**
- **`void delay(int numSeconds)`** - This function gives some delay when printing the intro of the game.
- **`int main()`** - This function is where the control flow of the game takes place.

3.3. Inputs and Outputs format

- To make a move chess with chess piece, user must input the position of the piece and then followed by the destination of subject piece. For

example, if user wants to move a chess piece located at a2 to destination a4, user must input "`a2 a4`" or "`A2 A4`".

- Log file will be recorded in the following format:
  - *[(Old Position) (New Position)]*
  - For example, if White Pawn moved from 2a to 4a and captured a Black Pawn at Turn 3, it will be recorded as "`2a 4a`" on log file.
- To quit the game, user may enter "quit" when making a move.
- To undo a move, user may enter "undo" when making a move,
- In case of errors the following messages may be displayed:
  - **"**Invalid input. Please enter a valid move, 'save' or 'quit' "
    - User attempted to move piece outside the chessboard. User must input a valid square (A1, A2, A3....H8). User is also given the option to save or quit the game.
  - "You've selected an empty square."
    - User attempted to move a piece that does not exist. User must move a piece that does exist.
  - "Not a legal move."
    - User attempted to perform an illegal move. Perform only legal moves specified by the official chess rules to avoid this from happening again.
  - "Invalid input. Please enter 'black' or 'white'."
    - User chose a color that is neither white nor black. User must choose either white or black.
  - "King will be in check. Select a different move."
    - Attempted move will put King in check. User must choose a different move.
  - "Please select white/black pieces only."
    - User attempted to move a chess piece that is not their own. User can only move their own color piece.
  - "Invalid input. Inputted %c but expected an input of Q, N, B, or R. Please try again."
    - User entered an invalid input. User must input either Q, N, B, or R as inputs.
  - "Could not open file. Aborting..."
    - File either could not be found or opened. Contact BitBots Software Developer for support.
  - "Out of memory! Aborting game..."
    - Some unforeseen error has occurred. Please contact BitBots Software Developer for support.

4. Development Plan and Timeline

4.1. Timeline

By End of Week 3 (Alpha version):

- Implement moves for each Chess piece along with their special moves.
- Implement User Interface and log file
- Implement Check and Checkmate functions
- Implement Draw function

By End of Week 4 (Version 1.0):

- Test the program
- Implement AI
- Implement human vs human mode
- Implement undo.

4.2. Partitioning of Tasks and team members responsibilities:

- UI & log file: Jacob
- Pawn, En passant, Promotion & King: Daniel
- Queen, Bishop, Rook, Castling, Knight: Martino
- Check: Hanqi, Ragui
- CheckMate: Ragui, Hanqi
- Draw: Hoon, Ragui

4.3. Team Responsibilities (all the team):

- Testing the game
- Implementing the game flow
- AI
- Undo
- Human vs Human mode

4.4. What is done (version 1.0)

- Text-based user interface (with colored pieces)
- All chess rules (piece movements, check, checkmate, draw)
- 3 modes (human vs computer, human vs human, computer vs computer)
- Option to undo moves
- Option to choose the color of pieces
- A log file of the played game is saved in bin directory.

Backmatter

Copyright:

References:

- **AlphaBeta algorithm:** https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/
- **Chess rules:** https://en.wikipedia.org/wiki/Chess

Index: