Developer Manual

SimuChess v1.0

Authors: Crystal Software (Team 19)
Jesse Corrales-Lozano
Mohammed Haque
Jose Alberto Padilla
Sean Michael Santarsiero
Ralph Hieu Cao Tran

University of California, Irvine. Henry Samueli School of Engineering.

Developer's Manual

Table of Contents

1.0 Software Architecture

This section gives a detailed overview of the software implementation of chess which includes topics such as data type, structures, software components, the API, and more.

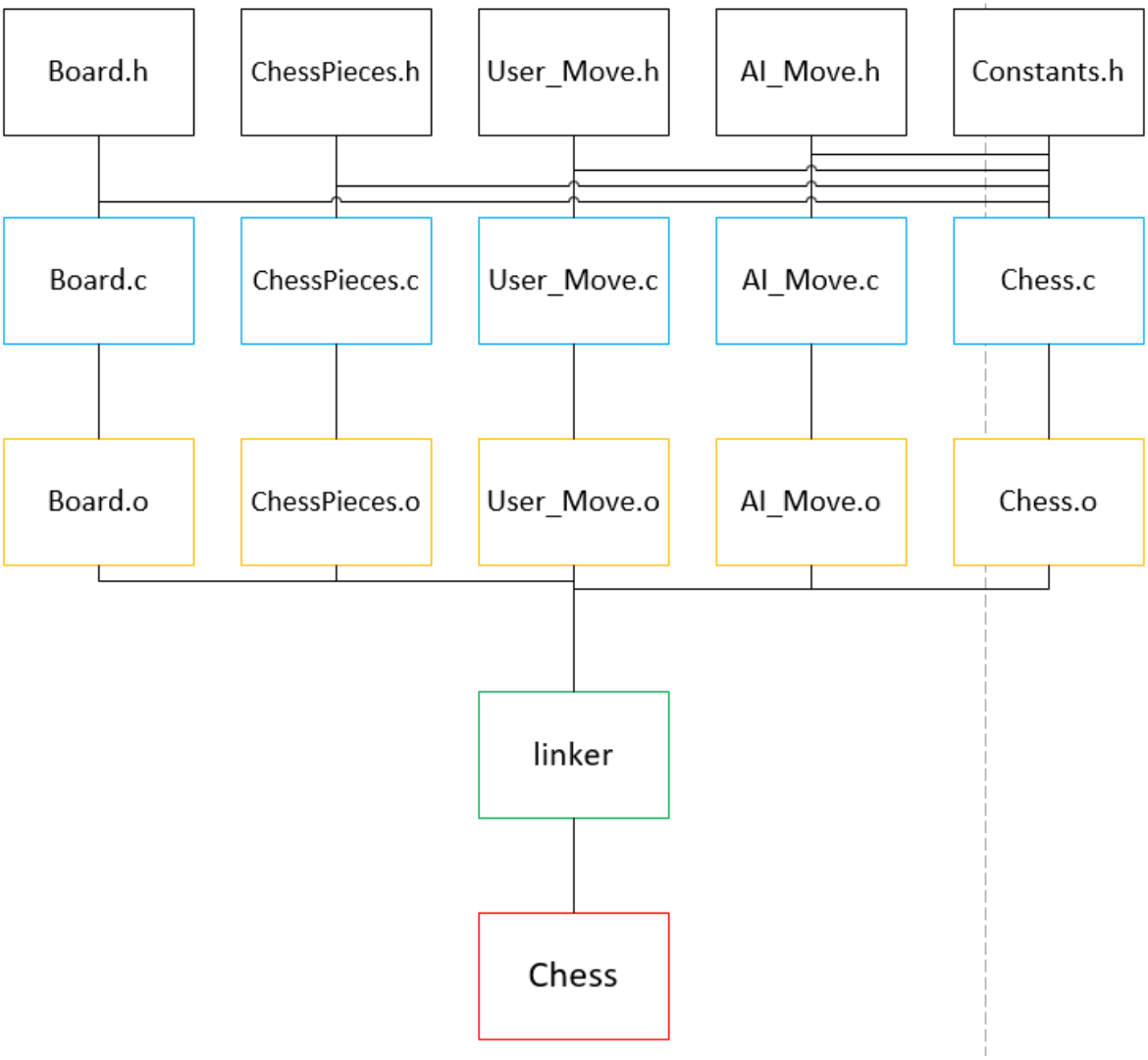
1.1 Main Data Types and Structures

- **1-Dimensional Array:** will be used to represent the chess board
- **Struct:** A struct will represent each of the chess pieces on the board and its characteristics.
- **Enum:** Serves to make code more understandable to the developer by allowing us to associate integer values with pieces which will be used for moving them around the board.
- **Int:** Primary data type that will be used to analyze and compute information about the board, its pieces, and the moves of those pieces
- **Arrays:** IntBoard[64] , Pieces[33] and Spaces[64].
- **Structs:** NODELIST, NODE, BESTMOVE, PLAYER, Piece and Space.

1.2 Major Software Components

The Board class will handle the creation of the board and fill it with chess pieces in the appropriate locations, while the Piece class will construct the actual piece type themselves ie. pawn, rook, etc. The Movement class will be responsible for calculating the valid move locations for every piece. A separate class will be dedicated to the AI which will be able to calculate its moves a few steps ahead and make the best choice. Finally, all of these classes will be included in the Chess class which will contain the main method.

1.2.1 Diagram of Module Hierarchy



1.3 Module Interfaces

The API provides us with a guideline into the capabilities of our program along with the specifications of major functions.

1.3.1 API of major module functions

- ❖ MovePawn:

- ➢ Input: Selected Pawn, Selected Move Space, Integer Board
- ➢ Output: return 0 for success and -10 for invalid
- ➢ Description: Determines what position on board a pawn can move and updates integer board. While cycling through possible specials such as Pawn Promotion and En Passant.

❖ MoveKnight:

- ➢ Input: Selected Knight, Selected Move Space, Integer Board
- ➢ Output: return 0 for success and -10 for invalid
- ➢ Description: Determines what position on board a knight can move and updates integer board.

❖ MoveKing:

- ➢ Input: Selected King, Selected Move Space, Integer Board
- ➢ Output: return 0 for success and -10 for invalid
- ➢ Description: Determines what position on board a king can move and updates integer board. Limits movement to not allow user to place their own king into check.

❖ MoveQueen:

- ➢ Input: Piece Value, Integer Board, Selected King and Selected Move Space
- ➢ Output: return 0 for success and -10 for invalid
- ➢ Description: Determines what position on board a Queen can move and updates integer board..

❖ GetQueenDirection:

- ➢ Input: Selected Queen, Selected Move Space and Enum value
- ➢ Output: return 0 for success and -10 for invalid
- ➢ Description: Determines what direction a Queen can move.

❖ MoveBishop:

- ➢ Input: Piece Value, Integer Board, Selected King and Selected Move Space
- ➢ Output: return 0 for success and -10 for invalid
- ➢ Description: Determines what position on board a Bishop can move and updates integer board..

❖ GetBishopDirection:

- ➢ Input: Selected Bishop, Selected Move Space and Enum value
- ➢ Output: return 0 for success and -10 for invalid
- ➢ Description: Determines what direction a Bishop can move.

❖ MoveRook:
- ➢ Input: Piece Value, Integer Board, Selected King and Selected Move Space
- ➢ Output: return 0 for success and -10 for invalid
- ➢ Description: Determines what position on board a Rook can move and updates integer board..

❖ GetRookDirection:
- ➢ Input: Selected Rook, Selected Move Space and Enum value
- ➢ Output: return 0 for success and -10 for invalid
- ➢ Description: Determines what direction a Rook can move.

❖ PiecePosition:
- ➢ Input: Selected Piece Value from Integer Board
- ➢ Output: return position in Pieces array of particular value
- ➢ Description: Uses enum constant value of piece and relates it back to Pieces[33] which allows access to a piece's struct members.

❖ Position:
- ➢ Input: none
- ➢ Output: returns location on board related to user input
- ➢ Description: User will enter "a6" and this will run through a enum struct Spaces array thus giving the position input is in relation to array.

❖ PlayerTurn:
- ➢ Input: Player Struct
- ➢ Output: Void function that determines which piece user is moving and calls appropriate move function.
- ➢ Description: Uses Positions() to determine a piece and space user wants and determines which move piece function to call.

❖ KingSearch:
- ➢ Input: Player Struct

- ➢ Output: Position of Player's king on the integer board.
- ➢ Description: Allows access to player's king for check and checkmate functions.

❖ Capture:
- ➢ Input: the piece that is going to be captured
- ➢ Output: none
- ➢ Description: will remove the respective piece from the board

❖ isKinginCheck:
- ➢ Input: King Value and Position
- ➢ Output: Returns 1 if in check or 0 if not in checkmate.
- ➢ Description: Determines if king is in checkmate.

❖ isKinginCheckmate:
- ➢ Input: King Value and Position
- ➢ Output: Returns 1 if in check or 0 if not in checkmate.
- ➢ Description: Determines if king is in checkmate.

❖ Lost:
- ➢ Input: boolean variable
- ➢ Output: A display message indicating which team has won
- ➢ Description: If a player has been checkmated, then a user message will be displayed indicating which team has won. Then the user will be redirected to the main menu.

❖ Castle:
- ➢ Input: Selected Chess Piece and Integer Board
- ➢ Output: returns flag that determine which of the 4 castle options are available.
- ➢ Description: Reviews board in reference to King and determines which castling movement is possible.

❖ En Passant:
- ➢ Input: Selected Pawn and Position of Pawn
- ➢ Output: increments flag and returns En Passant space
- ➢ Description: Activates flags in main that allow of this special move to be available.

❖ PawnPromotion:

- ➢ Input: Selected Pawn and Position of Pawn
- ➢ Output: void function that updates board with new piece
- ➢ Description:Depending of lplayer, allows for selection of promotions if Pawn reaches other end of board.

❖ DeactivateEnPas:

- ➢ Input: a pawn
- ➢ Output: none
- ➢ Description: will change the flag variable, thus preventing a pawn from performing En Passant if it is not performed on the current turn

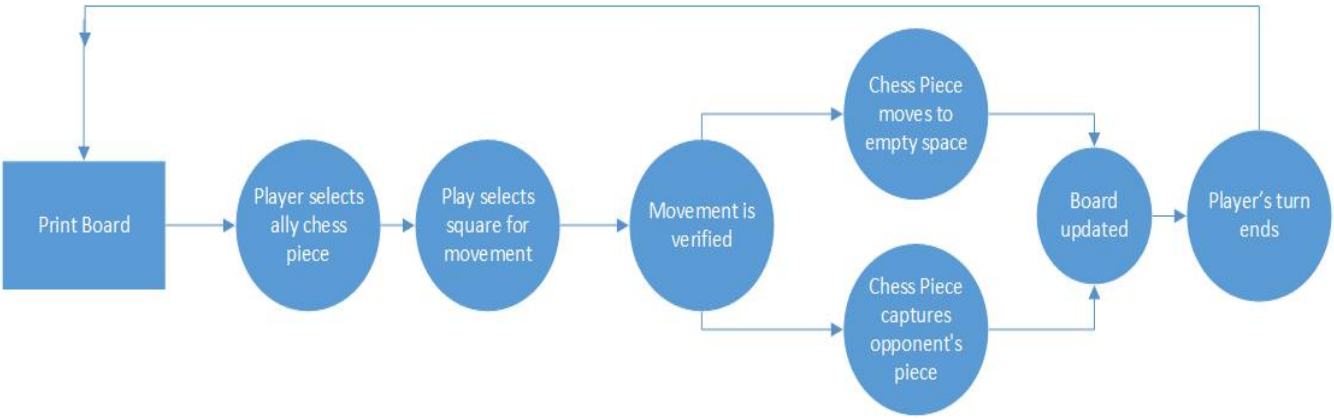
❖ PrintBoard:

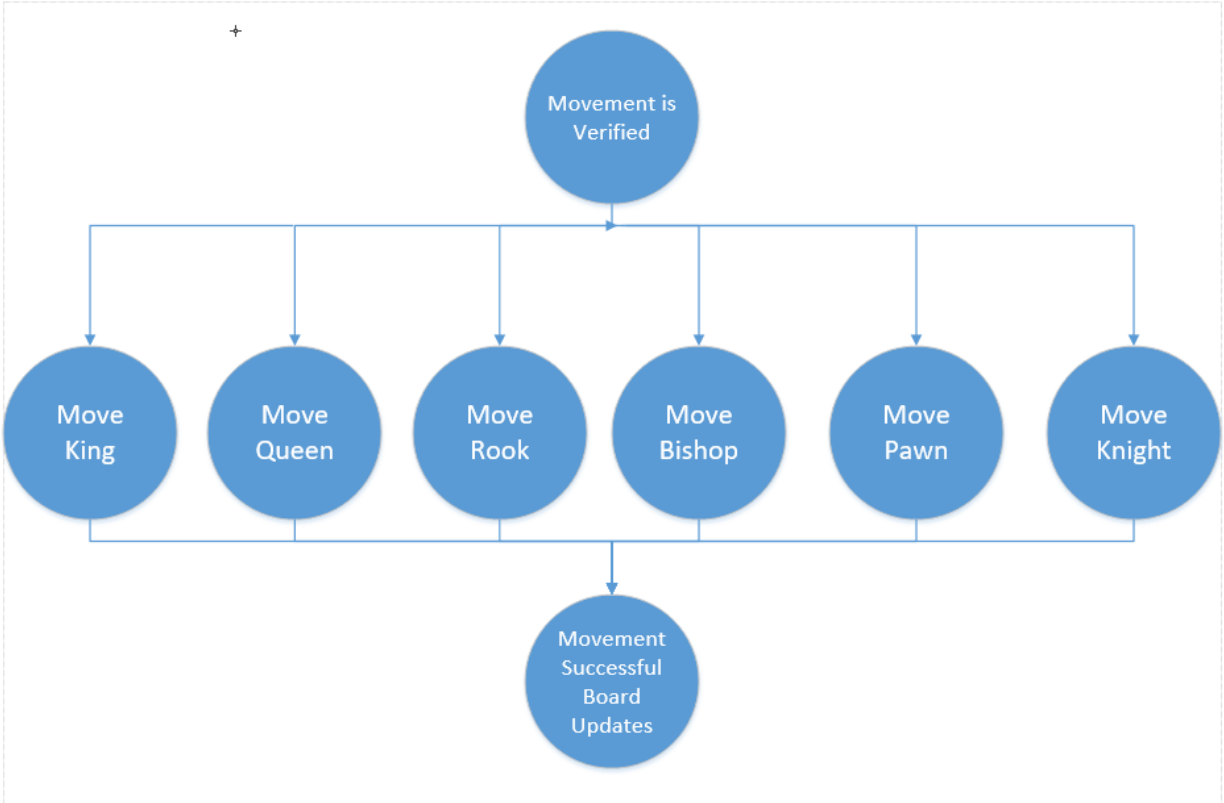
- ➢ Input: an array of integers
- ➢ Output: an ASCII board
- ➢ Description: will take the input to populate the board with the correct pieces

❖ Search:

- ➢ Input: an integer, indicating the the team
- ➢ Output: a linked list
- ➢ Description: will determine which moves are best and return a tree that will indicate the best option. Will be able to calculate moves for the AI a few steps ahead.

1.4 Overall program control flow





The program will run on a loop as long as neither king is in checkmate. The program begins by printing the board and letting the white player make a move. The intended move is then verified to be valid. If it is, the move is made. If it is not, the program lets the player select a move again. After the move is made, the board is updated and printed once again and the program now gives control to the black player. The program run in the

same loop alternating between players. Once a king is in checkmate, the program declares the winner and the program ends.

2.0 Installation

Information of the system requirements, how to setup and configure, and how to install the program.

2.1 System requirements, compatibility

CPU: Intel(R) Xeon(R) Processor E5-2660 v4 @ 2.00GHz
OS: Linux (CentOS release 6.9)
Memory: 4 GB RAM
Graphics: 512 MB Video Memory
Storage: 1MB

2.2 Setup and Configuration

How to install SimuChess:

1. Download the Chess_V1.0_src.tar.gz source code package.

2. Open the package by typing the following into the command line of a SSH client (omitting quotes): "tar -xvzf Chess_V1.0_src.tar.gz" then "cd Chess_V1.0_src" then "make".

```
tar -xvzf Chess_V1.0_src.tar.gz
```

```
cd Chess_V1.0_src
```

```
make
```

3. Type "./Chess" (omitting quotes) into the command line in folder bin to run the game and begin playing.

```
./Chess
```

2.3 Building, compilation, installation

To build and compile all files together, type in *make* within the command line while in the folder *src* to call the Makefile that generates the object files *ChessPieces.o, User_Move.o*, *board.o, AI_Move.o, and Chess.o* with the corresponding dependencies. The Makefile will also generate an executable file called *Chess* in the folder *bin* with the object files being the dependencies.

3.0 Documentation of packages, modules, interfaces

Information on data structures, functions and parameters, and input and output formats with snippets of code.

3.1 Detailed description of data structures

Struct: A struct will be used to represent the pieces. The struct will contain variables to represent what chess piece it is, i ts color, and flags that dictate whether certain special moves are possible or the check status of the king.

Array: An 1-dimensional array of 64 integers will be used to represent the board and the pieces occupying it. A 2-dimensional array will be used to store the history of the board. This will be used in the undo feature to return to previous board states.

Enum: Since integer values will be used to represent a various characteristics of a piece, enum will be used to make the code more readable and consistent.

3.1.1 Critical snippets of source code

Integer Array (IntBoard[64]) sets initial board setup in one dimensional array using enum constants to represent every piece available in game and 0's for empty spaces. This is where all actually movement and removal of pieces will happen.

Struct array (Squares[64]) is used to represent every available space on the board. It is used to be compared to user's input which is a 2-D array input (row,column) and relates it to a 1D array which is then compared to the IntBoard[].

```
int IntBoard[64]= {bQr,bQn,bQb,bQ,bK,bKb,bKn,bKr,       // This is our integer board which we will actually be rearranging and removing values to show game progres
                   bp1,bp2,bp3,bp4,bp5,bp6,bp7,bp8,
                   0,0,0,0,0,0,0,0,
                   0,0,0,0,0,0,0,0,
                   0,0,0,0,0,0,0,0,
                   0,0,0,0,0,0,0,0,
                   wp1,wp2,wp3,wp4,wp5,wp6,wp7,wp8,
                   wQr,wQn,wQb,wQ,wK,wKb,wKn,wKr};


struct Spaces Squares[64] = { {a8,"a8"},{b8,"b8"},{c8,"c8"},{d8,"d8"},{e8,"e8"},{f8,"f8"},{g8,"g8"},{h8,"h8"},
                              {a7,"a7"},{b7,"b7"},{c7,"c7"},{d7,"d7"},{e7,"e7"},{f7,"f7"},{g7,"g7"},{h7,"h7"},
                              {a6,"a6"},{b6,"b6"},{c6,"c6"},{d6,"d6"},{e6,"e6"},{f6,"f6"},{g6,"g6"},{h6,"h6"},
                              {a5,"a5"},{b5,"b5"},{c5,"c5"},{d5,"d5"},{e5,"e5"},{f5,"f5"},{g5,"g5"},{h5,"h5"},
                              {a4,"a4"},{b4,"b4"},{c4,"c4"},{d4,"d4"},{e4,"e4"},{f4,"f4"},{g4,"g4"},{h4,"h4"},
                              {a3,"a3"},{b3,"b3"},{c3,"c3"},{d3,"d3"},{e3,"e3"},{f3,"f3"},{g3,"g3"},{h3,"h3"},
                              {a2,"a2"},{b2,"b2"},{c2,"c2"},{d2,"d2"},{e2,"e2"},{f2,"f2"},{g2,"g2"},{h2,"h2"},
                              {a1,"a1"},{b1,"b1"},{c1,"c1"},{d1,"d1"},{e1,"e1"},{f1,"f1"},{g1,"g1"},{h1,"h1"} };
```

Above is the initialization of two 64 bit arrays that represent the 8x8 matrix that is our board. IntBoard is the actually board we are manipulating and it is filled with constant values defined through enum. Spaces[64] is used take user input as a string and match it up with another enum constant.

```
typedef enum {
        a8,b8,c8,d8,e8,f8,g8,h8,
        a7,b7,c7,d7,e7,f7,g7,h7,
        a6,b6,c6,d6,e6,f6,g6,h6,
        a5,b5,c5,d5,e5,f5,g5,h5,
        a4,b4,c4,d4,e4,f4,g4,h4,
        a3,b3,c3,d3,e3,f3,g3,h3,
        a2,b2,c2,d2,e2,f2,g2,h2,
        a1,b1,c1,d1,e1,f1,g1,h1
}Space;                                          // this represent the [8][8] chess board in constant values started a8=0,a7=1, and so on

enum{empty,wp1,wp2,wp3,wp4,wp5,wp6,wp7,wp8,wQr,wQn,wQb,wQ,wK,wKb,wKn,wKr,  // sets a constant value to each piece on board
        bp1,bp2,bp3,bp4,bp5,bp6,bp7,bp8,bQr,bQn,bQb,bQ,bK,bKb,bKn,bKr};
```

Above are the assigned enum values for pieces on the board and the enum values for each space on the board.

```
struct Pieces    // struct that will be used for all pieces
{

int value;       // This value is the pieces particular enum value
char color[50];  // Black or white used identify players
char name[50];   // used to generate a tag to identify piece on board
int Pawnflag;        // at the moment no use, but thinking flags can be assembled to enable other functions depending on piece
int AIvalues;

};
```

Above is the struct that is to represent every piece available to the player. This struct is used to define Pieces[33] which acts as a cabinet that holds the pieces which is shown below.

```
struct Pieces Piece[33] = {{0,"None", "Nothing",0},
                           {1,"White","wP",1},{2,"White","wP",1},{3,"White","wP",1},
                           {4,"White","wP",1},{5,"White","wP",1},{6,"White","wP",1},
                           {7,"White","wP",1},{8,"White","wP",1},
                           {9,"White","wR",1},{10,"White","wN",1},{11,"White","wB",1},
                           {12,"White","wQ",1},{13,"White","wK",1},{14,"White","wB",1},
                           {15,"White","wN",1},{16,"White","wR",1},
                           {17,"Black","bP",1},{18,"Black","bP",1},{19,"Black","bP",1},
                           {20,"Black","bP",1},{21,"Black","bP",1},{22,"Black","bP",1},
                           {23,"Black","bP",1},{24,"Black","bP",1},
                           {25,"Black","bR",1},{26,"Black","bN",1},{27,"Black","bB",1},
                           {28,"Black","bQ",1},{29,"Black","bK",1},{30,"Black","bB",1},
                           {31,"Black","bN",1},{32,"Black","bR",1}};
```

These three arrays represent the entire structure of our program is are referenced between each other to signal flags and update the board through the game.

3.2 Detailed description of functions and parameters

PlayerTurn:

This function is responsible for calculating the potential moves for a specific piece. The parameter will be the color of the current players piece. In order to do this, other functions will be provided that will check to see if the space is valid (Position), in other words, is within the bounds of the board array. After the space has been verified, we must check to see if another piece resides on it, and if not, then the move can be performed. Else, the color of the residing piece will determine whether that location can be moved to. If an ally is on a

tile, then a piece cannot move there, but if it is an enemy, then it can capture it while moving there.

PrintBoard:
Accepts an array of integers as input that will be used to populate the array board. When starting a game, a predefined array will have elements that will indicate what pieces to place and where into the actual board. Other functions will manipulate the input array, which represents pieces moving around and capturing each other, so that after every turn an updated board will be printed.

isKingInCheck, isKingInCheckMate & KingSearch:
At the end of a players turn, the piece they most recently moved will be used to calculate its potential moves from its new location. With the help of other functions (such as KingSearch), if the King of the enemy team lies on valid location that the current piece can move to, then this function will return true. When a player is put in check, their moves will be limited to ensure that their King can no longer be captured by the enemy. When a player is put in checkmate, the game ends declaring the winner

logFile:
Takes in the parameters of the start and end positions inputted by either the player or the AI. This function is called within main only for the sake of keeping track of the AI moves. It keeps track of the players moves within the function PlayerTurn which is called in main as PlayerTurn is the function that takes in and processes the players inputs.

3.2.1 Function prototypes and brief explanation

```
void PrintBoard(int arr[]);
Int Move(int loc);
```

Bool Check(Piece piece);
Refer to section 3.2 for a description of these major functions.

3.3 Detailed description of input and output formats

User enters a space on board (EX: “a7”) and this value is then compared to the struct array Squares[64]. From here, Squares[64] is related to the IntBoard[64] and Pieces[32] (struct array that contains every available piece in game). These references between the arrays will output exactly what space the user choose and confirms if it is empty or occupied. If occupied, the piece description will be given. Furthermore, user’s input will be sent through validation to confirm square is indeed present on board or error message will be returned.

3.3.1 Syntax/format of a move input by the user

User will type commands using the chess convention of rank and file. An example of such a command is >> e2 e4.
The program will read the two entries separated by a space and perform the move if it is legal.

3.3.2 Syntax/format of a move recorded in the log file

The log file will record all the moves and store them using algebraic notation. This log file will be stored in the bin folder as LogFile.txt. The log file will specify the starting location and ending location of every move made starting with the white player.

Example:
e2 -> e4
a7 -> a6
e4 -> e5

4.0 Development plan and timeline

Outlines how the group will divide the tasks and complete them in a timely fashion

4.1 Partitioning of tasks

The task of creating a chess game will be divided into 4 major components: AI, Pieces and their moves, and creating the board and rules.

4.2 Team member responsibilities

Sean Santarsiero - Creating the board and special moves

Arrays and data structures to be used due Jan 23
Functions that receives user input to move pieces due Jan 24
Error handling of invalid input due Jan 25
Link to UI due Jan 26
Knights due Jan. 25
Pawns due Jan. 23
King due Jan 27

Syam Haque - Rules of Chess, Logfile, and Main function

Create Makefile and LogFile function Jan 23.
Write a main function that tests completed code Jan 24.
Work on AI and update main function as needed due Continuous

Jose Alberto Padilla - Developing the chess pieces with the required members and functions
Rooks and Bishops due Jan. 24

Queen due Jan 26
Check function due Jan 28
Checkmate function due Jan 29

Jesse Corrales-Lozano and **Ralph Tran -** Creating an A.I. that will follow the rules of chess and be able to plan its moves in advance

AI that makes random moves due Jan. 23
AI that can capture when possible Jan 24
AI that can look 2 steps ahead using minimax algorithm due Jan 26
Alpha-Beta pruning of minimax algorithm due Jan 27

5.0 Back Matter

5.1 Copyright

5.2 References

YouTube
BlueFeverSoft
https://www.youtube.com/watch?v=bGAfaepBco4&list=PLZ1QII7yudbc-Ky058TEaOstZ

chessprogramming
https://chessprogramming.wikispaces.com/

Wikipedia
https://en.wikipedia.org/wiki/Bitboard

5.3 Index