Developer Manual

HeroChess Version 1.0

AVENGINEERS

Rachel Villamor, Keane Wong, Paul John Lee,
Mario Tafoya, and Irania Ruby Mazariegos
EECS 22L, University of California Irvine, Irvine, CA 92697

(Dated: April 20, 2021)

Contents

I. GLOSSARY OF CHESS TERMS 2

II. Software Architecture Overview 4

A. Main Data Types and Structures 4

B. Major Software Components 4

1. Flow of control(Order of execution) 4

2. Structs/Functions required 5

C. Module Interfaces 8

D. Overall Program Control Flow 8

III. Installation 10

A. System Requirements, Compatibility 10

B. Setup and Configuration 10

1. Setup 10

2. Configuration 10

C. Building and Compilation 11

IV. Documentation of Packages, Module, Interfaces 11

A. Detailed Description of Data Structures 11

B. Detailed Description of Functions and Parameters 12

C. Detailed description of input and output formats 13

V. Development plan and timeline 14

A. Partitioning of tasks 14

B. Team member responsibilities 16

VI. Copyright 16

VII. Index 17

I. GLOSSARY OF CHESS TERMS

Bishop: A piece within the game that can only move diagonally across the board.

Black: Classification for the player who will move second when beginning the game.

Capture: When a capturing chess piece is capable of moving into the space that is occupied by an opposing chess piece, the opposing chess piece can be removed from the game and the capturing piece moves into its previously held space. This is referred to as capture and is the primary means of advancing the game.

Castling: A special move in the game of chess where the king and rook are moved simultaneously. It is important to note that this move is only allowed if the both pieces have not made a move and the king has two spaces available towards his left or right. This move may not be performed if the king is in check or if there is another piece blocking the areas where the king and rook would perform this move through. To perform the move move the king two spaces in the desired direction(left or right) and swap the rook to the first square from the king's original position. If you start as white you could castle by moving the king from e1 to g1 and rook from h1 to f1.

Check: Designates that a chess piece is able to capture the king, forcing the targeted opponent to necessarily make a move to prevent the attack from happening. The resulting move must not leave the king open to any additional checks.

Checkmate: Designates that a check has been made, and the opponent is unable to make any moves that would protect the king from additional checks. This condition wins the game.

En passant: A special pawn capture that can only occur immediately after a pawn makes a move of two squares from its starting square, and it could have been captured by an enemy pawn had it advanced only one square. The opponent captures the just-moved pawn "as it passes" through the first square.

File: The 8 vertical columns on the board, labeled 'a' through 'h'

King: King can move in any direction in steps of one. It is important that the king stays out of check and uses its special moves with other pieces to win the game.

Knight: A knight can move in L shapes that are made up of two squares forward and one step to the side.

Pawn: Can only move forward on the board. Can choose to advance one or two spaces only on the first move. After the first move the pawn must move one space. The pawn can only capture diagonally.

Promotion: When a pawn has moved to the opposite edge of the board and cannot advance any further, it may be turned into any other game piece excluding the king.

Rank: The eight horizontal rows on the board, labelled 1 to 8.

Rook: A piece within the game that can only move vertically and horizontally across the board.

Queen: A piece within the game that can move in any direction on the board.

White: A term used to classify the player that will make the first move.

Board: A square 2D array with the size of 8x8 that consist of 64 pieces*

Main: The main function for the program

AI: Computer player which will play against humans in Human vs. AI mode

Move: A move history that indicates the source and destination xy coordinates of a piece(Chars).

Movelist: A Linked list which contains the replays of games.

Player: A player struct that contains player name and color.

.c: file extension for a c programming source file.

.h: file extension for a c header file.

Struct: a collection of variables under a single name.

Char: a character variable

Array: a collection of data items

II. SOFTWARE ARCHITECTURE OVERVIEW

A. Main Data Types and Structures

Various data structures will be used to represent aspects of the game. Most prominently, a 2D struct array will be used to represent the game board, and because a chess board is of a fixed standard size, we can more easily hard code certain functions as needed. A 2d array is chosen because its fixed size is ideal to work with for a game with universally standard dimensions, and structs will be used to contain data.

Each piece itself will be represented as a struct containing character and integer types. Because each pieces in chess do not contain piece-specific internal data (i.e behave the same no matter their position and what has happened to the piece in the past turns), we will not need to modify the piece data as we go, allowing us to use simple character and integer codes to represent pieces, and we will also not need to differentiate between duplicate pieces either. Auxiliary data structures include a queue used to track captured pieces (for display or for strategic analysis), and a doubly linked list used to keep track of the elapsed moves. This allows us to keep data in order and to undo moves as necessary. Other important data forms include text file writing for use of saving information and game states.

B. Major Software Components

1. Flow of control(Order of execution)

General Procedure:

1. Main calls game module
2. Game module reads data from the board
3. Game module checks if the move is valid
4. Game module executes or refuses move
5. Board is modified
6. Game checks for checks and checkmates
7. Next turn is executed in main

Turn execution:

1. Takes the user input for the piece they want to move(ex. E7)
2. Checks for the piece type
3. Create the list of legal moves of the selected piece(selected square)
4. Takes the user input for the destination
5. Checks for the validity
6. Execute. If the move is invalid, ask for a new input

extra:

1. Takes the user input for the piece they want to move(ex. E7)
2. Checks for the piece type
3. Create the list of legal moves of the selected piece(selected square)
4. Takes the user input for the destination
5. Checks for the validity
6. Execute. If the move is invalid, ask for a new input

2. Structs/Functions required

Piece

Piece indicates the chess pieces placed on the board. It will contain the information about the piece type and color that is required for the game to execute player moves, check the win condition each turn and display the updated board.

TABLE I: Struct Piece

Type	Name	Purpose
char	type	To identify the piece type
char	color	To identify the piece color

TABLE II: Piece

Type	Name	Purpose
Piece*	NewPiece(char name)	To Create a new piece element for the board
char	getType	To return identifier for the selected piece
char	getColor	To return the color identifier for the selected piece

Move

Move is where all the moves that players make will be stored in the form of a string. It tells which piece made what move on the board for tracking and takeback purposes. User will be able to save the moves in the form of .txt file after the game. Struct move will

TABLE III: Struct move

Type	Name	Purpose
PIECE*	piece	To identify piece moved
char*	source	Source square of the piece
char*	destination	Destination square of the piece

TABLE IV: move

Type	Name	Purpose
MOVE*	NewMove(Char *move)	To create a new move element to the move list
char*	GetMove(Move *m)	To return the source of the existing move
void	PrintMove(Move *m)	prints moves
void	DeleteMove(Move *m)	Delete a move and deallocate the memory used

Movelist

Movelist is a helper module for move.c that works as an iterator of the list of moves. A doubly linked list will be used. It allows the game to add/delete and print moves as long as the game goes. Inserting/deleting a move from the list will only be allowed statically to prevent any disruptions. Movelist has the pointers to the first and last element of the list including the length of the list, and MoveListEntry functions as an iterator which can move to the next/previous element at a time.

TABLE V: Struct Movelist

Type	Name	Purpose
int	length	Length of the list
MoveListEntry*	first	Pointer to the first element of the linked list
MoveListEntry*	last	Pointer to the last element of the linked list

TABLE VI: Struct MovelistEntry

Type	Name	Purpose
MoveList*	list	Pointer to the list
MoveListEntry*	next	Pointer to the next move
MoveListEntry*	prev	Pointer to the previous move
MOVE*	move	Move element to be stored

TABLE VII: Movelist

Type	Name	Purpose
MoveList*	NewMoveList()	Construct a new move list in a form of linked list
void	DeleteMoveList()	Destructor
void	AppendMove	Add a move after the current move
void	PrependMove	Add a move before the current move.
int	GetLength	Return the length of the list

Game

TABLE VIII: Game

Type	Name	Purpose
int	isLegal(Piece** myBoard, int colSource, int rowSource, int colDestination int rowDestination, char curTurnColor	If there is no piece at the source, it automatically fails If the selected piece is not the curTurnColor, it also fails If the Destination is out of bounds it automatically fails.
int	isLegalPawn(Piece** myBoard, int colSource, int rowSource, int colDestination, int rowDestination, char curTurnColor)	Checks if it is at initial position (either row[1] for white or row [6] for black is in the (array) and if it is, the legal moves also includes a space 2 squares forward. This piece CANNOT capture whats in front of it. Also, checks for special moves that involve the pawn like EN PASSANT.
int	isLegalRook(Piece** myBoard, int colSource, int rowSource, int colDestination, int rowDestination, char curTurnColor)	Moves consist of all spaces horizontal or vertical to it. Cannot move past pieces once it has captured a piece. Allows for castling if the king and itself both have yet to move.
int	isLegalBishop(Piece** myBoard, int colSource int rowSource, int colDestination, int rowDestination, char curTurnColor)	It's moves consist of diagonal movement across the board. can move as many squares as possible if there is no piece obstructing but must stop once it has captured.
int	isLegalQueen(Piece** myBoard, int colSource, int rowSource, int colDestination, int rowDestination, char curTurnColor)	Can move in any direction but must stop once it has captured or if it is obstructed

TABLE IX: Game

int	isLegalKing(Piece** myBoard, int colSource, int rowSource, int colDestination, int rowDestination, char curTurnColor)	Can move in any direction but only one square at a time. The exception is when it is performing castling.
int	isLegalKnight(Piece** myBoard, int colSource, int rowSource, int colDestination, int rowDestination, char curTurnColor)	Checks only if the destination is empty or has an enemy 2 squares straight, then 1 square perpendicular
int	makeMove(Piece** myBoard, int colSource, int rowSource, colDestination, int rowDestination, char TurnColor)	Uses isLegal function from TABLE V to determain if the move is okay and then uses movePiece function to execute
int	isCheckmate(Piece** myBoard, char TurnColor)	Checks for checkmates for the curTurnColor. searches board for king of that color and then determines All possible checks. Returns 0 if there is no checkmates if there is at least one check but no checkmate it returns 1. If there is a checkmate it returns 2 and the game ends
int	isChecked(Piece** myBoard, char TurnColor)	Searches for king of color TurnCOlor. Checks for checks around that king. If there is none it returns 0. If there is at least one check then it runs (and immediately returns) isCheckmate. Run this function twice every turn; Once during 'islegal' function to determine if the Once during 'islegal' function to determine if the C(which makes the move invalid) and once at the end of each turn to determine if the enemy is in check

Board

Board is the module that stores the 8x8 board where players will play the game of Chess. It is a 2D-array of struct Piece as they are the main elements of the game. Board module will enable the player moves to be an outcome on the board such as the initial setup, moving, taking pieces. The game will be able to obtain the information of a designated coordinate (whether it is occupied, type, color if occupied) that is required by the program to properly run the game.

TABLE X: Board

Type	Name	Purpose
Piece**	makeBoard()	initializes the game board in the form of a 2D array.
int	deleteBoard(Piece** myBoard)	Deletes the board, freeing the memory and deallocating the array.
Piece	getPiece(Piece** myBoard, int col, int row)	Returns the piece at position colrow
int	placePiece(Piece** myBoard, int col, int row, Piece q)	Puts the piece specified at colrow, returning 0 if successful else 1
Piece	removePiece(Piece** myBoard, int col, int row)	Removes piece from specified colrow, returning the removed piece and filling the spot with null. Returns null if there is nothing there.
int	movePiece(Piece** myBoard, int colSource, int rowSource, int colDestination, int rowDestination)	Calls removePiece on the piece at Source and stores it. It also calls removePiece on the piece at Destination, then calls placePiece at destination. if the piece was removed then it returns the removed piece. The returned piece must be saved and then freed by the program during lifetime
void	PrintBoard(Piece** myBoard)	Prints board to the console.
void	initalizeBoard()	set up the initial condition to the board.

AI

Module supports the Human vs. AI gamemode

TABLE XI: AI

Type	Name	Purpose
int	AdvancedTurn (Piece **myboard , char AIcolor)	Takes a 2d array representing game board and returns a move based on what color the AI is in the board The returned move is an integer equal to the following (col*10=row) where col and row are the column and row numbers respectively. This function returns a highly intelligent move choice based on a calculated algorithm
int	IntermediateTurn (Piece **myboard, char AIcolor)	Takes a 2d array representing the game board and returns a move based on what color the AI is in the board. The returned move is an integer equal to the following (col*10=row) where col and row are the column and row numbers respectively. This function returns a intermediately intelligent move choice based on a calculated algorithm
int	BasicTurn (Piece **myboard, char AIcolor)	Takes a 2d array representing the game board and returns a move based on what color the AI is in the board. The returned move is an integer equal to the following (col*10=row) where col and row are the column and row numbers respectively. This function returns a basic move badow on what is readily available or random.

Extra Features

- Dead Pieces(Queue)
- Replay
- ...

C. Module Interfaces

The primary functions of each module file should not require interaction with any private functions and perform their task concisely and without additional undocumented changes.

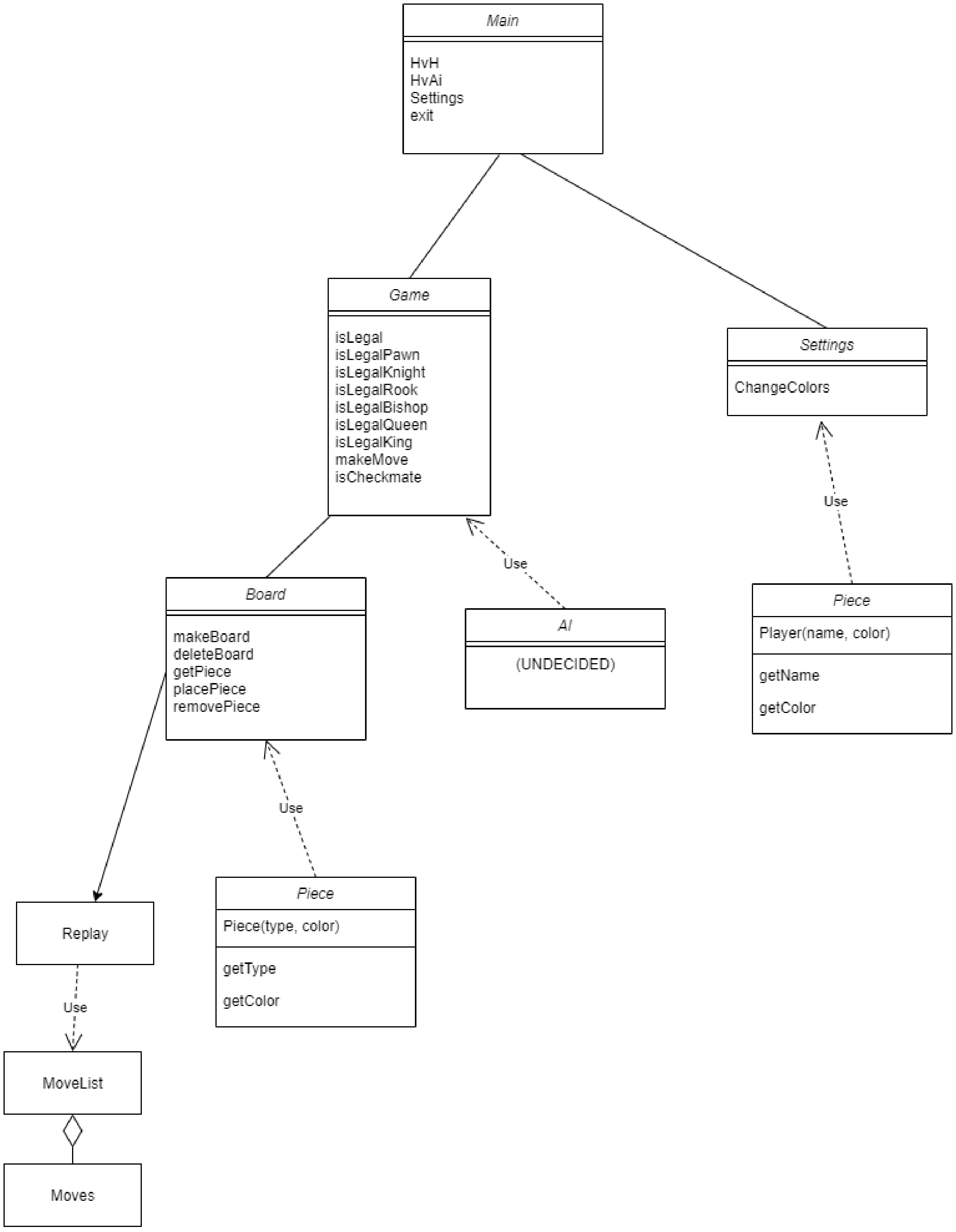
The piece files should have methods to change and get their respective parameters, and should only be changed as necessary or accessed as needed for settings and game manipulation. Similarly the board should have strict accessor and mutator functions, and should not require knowledge of the workings of the board as a data structure. These will be primarily accessed and modified by the game function, which only checks for move legality and returns appropriate messages based on errors, successes, and win conditions. The game will otherwise rely on the main function for inputs and GUI support. Otherwise, the settings files will be separated from the other files and primarily only contain mutator functions for user side settings and similarly the movelist function will not access the game nor board functions, but instead records data as the user inputs it, and is driven by the main function. This removes the modules from the rest of the program but retains its usual functionality. The **Hierarchical Diagram(alpha)** can be found in the following page.

D. Overall Program Control Flow

The HeroChess program will operate primarily from a GUI or Console command line driven by a main function to take user input and set initial parameters. This main function will control the primary operation of the game on the user end and also manage functions such as saving and replay. This is the primary function that handles turn to turn interactions, and makes use of the function outputs.

The board will be represented by a 2d array of pieces, which by themselves do not interact strongly with one another and contain only data relevant to the game rules. The board files do not check for move legality or errors except for out of bound errors. The software contained in the game.c and game.h files will primarily

FIG. 1: Hierarchical Diagram(alpha)



drive the game logic and do most of the runtime error handling and handle move legality. The game files and functions are written to handle most checks and rules automatically, and check win conditions as accurately as possible.

More separated from this hierarchy, but regularly interacting with it, are the movelist and settings files. These functions modify or read in data from the game or board without being directly tied to it or forming any of its structure. The setting files will change mostly cosmetic aspects and GUI parameters, without affecting the game itself very much.

The movelist data, represented as a linked list of structs, will be regularly called by the program to read in data and save it temporarily as the game progresses. These functions should NOT modify any game data. At the end, they can be accessed by the main function and stored into a readable plaintext file for replay later, and can alternatively be used to start the game and board from a set position.

III. INSTALLATION

A. System Requirements, Compatibility

In using any version of HeroChess users will want to access the EECS Linux Servers. Users are recommended to have the Linux version **CentOS 6.10** and perhaps even Xming if the graphical user interface is included.

The PuTTY Client does not have any special hardware or software requirements. The PuTTY client is compatible with any computer running Microsoft Windows 7, Windows 8.1, and Windows 10.

B. Setup and Configuration

1. Setup

The files will be made available throught the git repository https://github.uci.edu/EECS-22L-S-21-Team-Projects/Team19 including all the public and private files from there the company or individuals will be able to change and update any of the set function. Furthermore, all public and private files will be downloadable, usable, and clonable to personal directories.

2. Configuration

main.c-Contains main function and print main menu function.

game.c-Contains the initial setup/ game algorithms and win conditions

game.h-Declarations of functions defined in game.c

board.c-Contains the current board information

board.h-Declarations of functions defined in board.c.

piece.c-Contains the struct pertaining to the chess pieces' information (type, color) and

piece.h-Declaration of functions defined in piece.c

move.c-Contains srtuct move and relevant functions pertaining to moves

move.h-Declarations of functions defined in move.c

movelist.c-Contains srtuct MoveList and relevant functions pertaining to keeping track of the move list

movelist.h-Declaration of functions defined in movelist.c

settings.c-Contains player and game information

settings.h-Declarations of functions defined in settings.c

ai.c-Contains functions for the automated player and its various difficulties

ai.h-Declarations of functions defined in ai.c

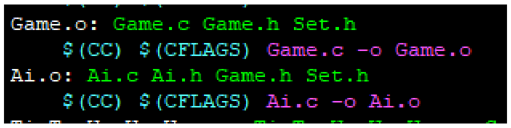
Makefile-Required. Contains the compilation command along with shared libraries

C. Building and Compilation

In building this project as described before it is absolutely necessary to build a Makefile that will collect the dependencies and compile them accurately. This is done so that the separate files can all compile at once. To begin you must open up the command line using the directions described in **B. Installation**. Once this is done you can create a Makefile using the following steps:

1. Open the relevant directory in which you git files are cloned to.
   - From the command line, run the command cd <directory path>
   - Example: cd ChessFiles
2. Create a new Makefile
   - Example vim Makefile
3. In this file list your rules and dependencies. Note: If this is done inaccurately the program might not compile. If this is the case check to see the compilation errors on the command line.

FIG. 2: Example Compilation Rules



```
Game.o: Game.c Game.h Set.h
    $(CC) $(CFLAGS) Game.c -o Game.o
Ai.o: Ai.c Ai.h Game.h Set.h
    $(CC) $(CFLAGS) Ai.c -o Ai.o
```

Once the rules have been set and the makefile has been complete you can go ahead and run the program by typing make into the command line this will prompt the makefile to run all the compilation rules and uotput a target file which will be your program file.

Furthermore, if you wanted to clean your makefile and re-run the compilation rules you can do that by

1. first clean your makefile
   - make clean
   - This will clean any .o files
2. Then re-run the make file using the make command

IV. DOCUMENTATION OF PACKAGES, MODULE, INTERFACES

A. Detailed Description of Data Structures

The majority of the important parts of the program are represented as basic structs containing basic data that pertains to the game rules.

The base piece structs are abstracted to represent only a piece on the board, and are initialized to contain the piece name and color. The rules for movement that pertain to each piece are hardcoded in the game files, and handled there independently to the pieces themselves. This data is meant to be initialized then accessed only, except in special cases.

These structs are then contained in a static 8x8 array of pieces at a fixed size to represent a board. The board is regularly modified and read to progress the game and contains purely positional data, and does not contain internal data to things such as turns elapsed, replays, or save states. This is the primary data structure to present the game and does not change, except in contents and location of contents.

Auxiliary data structures include a double linked list with a head and tail end, and contains data for elapsed moves that will then be applied to a set initial board, and saved to an external text file as appropriate. Each move in the movelist is contained in a move entry and can be accessed through the functions detailed in 1.2, and moves can be appended and deleted as necessary without the relative positioning of other moves being changed.

B. Detailed Description of Functions and Parameters

1. **NewPiece**: This function creates a new piece element for the board. The parameter is char name.
2. **getType**: This function returns the identifier for the selected piece.
3. **getColor**: This function returns the color identifier for the selected piece.
4. **NewMove**: This function creates a new move element to the move list. The parameters are char *move and PIECE *piece.
5. **GetMove**: This function returns the source/destination of the existing move. The parameter is Move *m.
6. **PrintMove**: This function prints the moves made throughout the game to a list. This list will allow the user to track their moves and take back any of their moves if they choose. The parameter is Move *m.
7. **IsLegal**: This function checks the color of the piece at the source and the color of the piece at the destination. The parameters are Piece** myBoard, int colSouce, int rowSource, int colDestination, int rowDestination, and char curTurnColor. If there is no piece at the source, the selected piece is not the current turn color, or the destination is out of bounds, the function fails. It will also fail if the selected piece and the piece at the destination are the same color. The function will proceed if the selected piece and the destination are the same color if the selected piece is a king or a rook and the destination is a rook or a king. If the destination is empty or has an opposing piece, the function checks what type of piece is at the source and calls the private isLegal function that corresponds to it. If the destination is legal, it then checks that the move does not place the friendly king in check.
8. **isLegalPawn**: This function checks if the pawn is at its initial position (row [1] for white and row [6] for black) and, if it is, the legal moves also include a space 2 squares forward. The pawn cannot capture anything that is directly in front of it. This function also checks for en passant, If the pawn moves forward and ends its turn on the opposite side of the board, a message will be printed to the console, taking user input. The user will be asked if they want to promote their pawn to bishop, knight, rook, or queen. The parameters are Piece** myBoard, int colSource, int rowSource, int colDestination, int rowDestination, chat curTurnColor.
9. **isLegalKnight**: This function only checks if the destination is empty or has an opposing piece 2 squares forward then 1 square perpendicular.
10. **makeBoard**: This function initializes the board in the form of a 2d array and fills the spaces with null.
11. **printBoard**: This function will print the board to the console with each piece arranged in their starting positions. It will also print the rank on the left-hand side of the board and the file at the bottom of the board. The parameter is Piece** myBoard which will print the initialized board and update it as the user makes moves.
12. **deleteBoard**: This function will delete the board, free the memory, and deallocate the 2d array. The parameter is Piece** myBoard.
13. **getPiece**: This function will return the piece with its position on the board. The parameters are Piece** myBoard, int col, and int row. This allows a unique return type for each piece. The return type will then be used in functions such as placePiece which needs the previous position of the piece in order to move it.

14. **placePiece**: This function will move the selected piece to the space that the user has entered. The parameters are Piece** myBoard, int col, int row, and Piece q. It will return 0 if successful and 1 if not. The return type will be used in a separate function that prints out an error message.

15. **removePiece**: This function removes a piece from a specific column and row, returning the removed piece and filling the space with null. If there is no piece to remove, the function will return null. The parameters are Piece** myBoard, int col, and int row.

16. **movePiece**: This function calls the removePiece function on the piece at Source and stores it. It also calls the removePiece function on the piece at destination, then calls the placePiece function at destination. The parameters are Piece** myBoard, int colSource, int rowSource, int colDestination, and int rowDestination.

17. **Advanced/Intermediate/basicTurnAI**: 3 different algorithms that determine, for any given board configuration, what the optimal choice is at varying levels of difficulty. The Advanced algorithm uses a series of weighted choices and heuristics to ensure a moderately useful to optimal move. The intermediate is similar but does not plan as far ahead, and the basic AI does not calculate more than 1 or two moves in advance.

C. Detailed description of input and output formats

In the Syntax/format of a move input by the user there will be two prompts printed for the user at the beginning of each turn. The first will state the following, "Please enter the location of the piece you want to move in letter-number format (e.g. a1):". The second will state the following, "Please enter the location you want the piece to move to, in letter-number format as well (e.g. a1):". Both of these prompts ask the user to enter inputs that refer to the column and row of a location in character-integer format.

In the Syntax/format of a move recorded in the log file every game's moves will be "recorded" by being added to a list. Function pertaining to this will be in the Movelist module. Implementing similar processes of printing a board each turn, each turn made throughout an entire game between both players will be printed onto a text file, which will be available in the user's directory after a game. This process of board printing will depend on all the values stored in the movelist.

The log file will contain the following:

```
# HeroChess
  - Basic Program and game information
# Settings
  - Player information
# Winner
  - Game result
# MovesList
  - All players moves.
# Replay
  - Board print out for all moves
# EOF
  - End of file notification
```

The log file will be formatted as such:

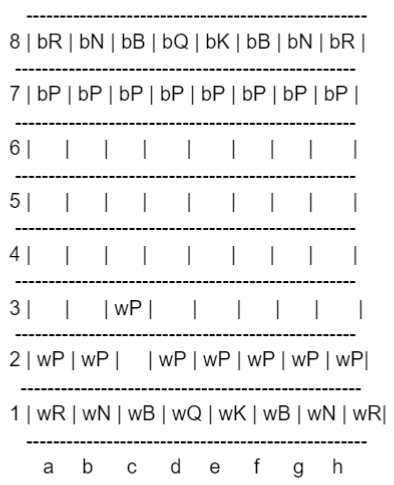
```
# HeroChess
Version: v1.0
Filename: HvAI-2021-04-20.txt
Date: 2021/04/20 11:00am

  # Settings
Player 1(Human):"White"
Player 2(AI):"Black"

  # Winner
Player 1
```

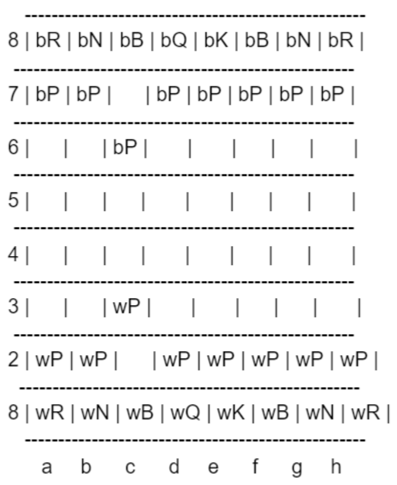
# MovesList
Player 1: C3
Player 2: C6
...

# Replay
Player 1 chooses C3



```
  ---------------------------------------------------
8 | bR | bN | bB | bQ | bK | bB | bN | bR |
 ---------------------------------------------------
7 | bP | bP | bP | bP | bP | bP | bP | bP |
 ---------------------------------------------------
6 |    |    |    |    |    |    |    |    |
 ---------------------------------------------------
5 |    |    |    |    |    |    |    |    |
 ---------------------------------------------------
4 |    |    |    |    |    |    |    |    |
 ---------------------------------------------------
3 |    |    | wP |    |    |    |    |    |
 ---------------------------------------------------
2 | wP | wP |    | wP | wP | wP | wP | wP|
 ---------------------------------------------------
1 | wR | wN | wB | wQ | wK | wB | wN | wR|
  ---------------------------------------------------
    a    b    c    d    e    f    g    h
```

Player 2 chooses C6



```
  ---------------------------------------------------
8 | bR | bN | bB | bQ | bK | bB | bN | bR |
 ---------------------------------------------------
7 | bP | bP |    | bP | bP | bP | bP | bP |
 ---------------------------------------------------
6 |    |    | bP |    |    |    |    |    |
 ---------------------------------------------------
5 |    |    |    |    |    |    |    |    |
 ---------------------------------------------------
4 |    |    |    |    |    |    |    |    |
 ---------------------------------------------------
3 |    |    | wP |    |    |    |    |    |
 ---------------------------------------------------
2 | wP | wP |    | wP | wP | wP | wP | wP |
 ---------------------------------------------------
8 | wR | wN | wB | wQ | wK | wB | wN | wR |
  ---------------------------------------------------
    a    b    c    d    e    f    g    h
```

..// all moves will be printed on the file

Player 1 wins!

# EOF

V. DEVELOPMENT PLAN AND TIMELINE

A. Partitioning of tasks

- **Timeline Overview**

1. Planning
2. Development
3. Prototyping
4. Testing
5. Prelaunch
    (a) Alpha release
    (b) Beta release
6. Launch
    (a) Master release
7. Competition

- **Planning**-Week 1
- **Development**-Week 2
    - Application Specification/User Manual
        - Whole Team Collaboration
- **Development Prototyping**-Week 3
    - Finalization's
        - Human v. Human, Human v. AI Games - Rachel Villamor
        - Implementing the Official Rules of Chess - Keane Wong
        - Advanced Moves - Irania Mazariegos
        - Moves Log - Mario Tafoya
        - Tournament Support - Paul Lee
        - LaTeX Conversion - Mario Tafoya
    - Software Implementation
        - Discussion - whole team
        - Started Code - Paul Lee
- **Development Prototyping**-Week 4
    - Software Architecture Specification
        - Manual sections
            - **Section II**(game, board, and piece functions)
              -Keane Wong

            - **Section IV**(input and output formats), **Section V**(Development plan/timeline)
              -Rachel Villamor

            - **Section II**(Overall program control flow)
              -Paul Lee

            - **Section IV**(functions and parameters)
              -Irania Mazariegos

            - **Section III**(Building and compilation), Latex Conversion
              -Mario Tafoya
- **Testing Pre-launch**-Week 5
    - Software Implementation
        - Work on module files - whole team will collaborate to edit and error-check each file
            - Main.c

        · Settings.c
        · Piece.c
        · Move.c
        · Movelist.c
        · Game.c
        · Board.c
        · Ai.c
      * Compile and run all .c and header files
      * Release Alpha Version

B. Team member responsibilities

- Members and Roles
  - Manager: Paul Lee
  - Presenter: Keane Wong
  - Recorder(s): Rachel Villamor, Mario Tafoya
  - Reflector: Irania Mazariegos

VI. COPYRIGHT

VII. INDEX