22CSE 220: Systems Fundamentals I

Stony Brook University
Homework Assignment #4
Spring 2024
Due: Sunday, April 14th, 2024 by 9:00 pm EDT

Updates to this Document

- Some of the test cases in the `parse_move` suite are named incorrectly. However, the return values of these test cases are still correct.

Learning Outcomes

1. An ability to use the C sockets API to exchange messages between two processes.
2. An ability to parse and interpret strings that convey structured information.

Assignment Overview

For this assignment you will implement a rudimentary, networked Chess-playing program. We imagine that a Chess player starts a server on his/her computer, and another player connects as the client. They play until one player forfeits. You will not need to implement all the rules of Chess. A brief summary of the rules of Chess that you must implement are provided later in the document. For those with some Chess knowledge, complex rulings such as en passant, castling, checking, and checkmating are excluded.

A copy of the game's state is maintained separately by the client and server. Messages exchanged by the two parties help to synchronize the copies. Much of the assignment entails writing functions to process these messages and update the game state. A typical message indicates how a piece should move on the chessboard. In essence, you will have to parse input in this format:

```
[piece_old_location][piece_new_location]
```

After implementing the chess move engine, you will implement the logic to process FEN strings, which provide a means of describing the state of a Chess game.

Any output your program generates to the screen will be ignored, so feel free to print debugging and informational messages to standard output (stdout) and standard error (stderr). The test cases are based entirely on checking the correctness of the game state, structs, return values, and so forth.

Chess Rules and Notation

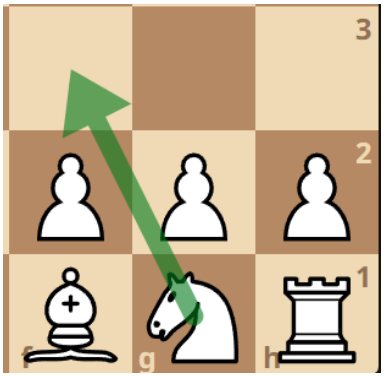
If you are not familiar with the rules of Chess, we have prepared a brief guide for you to check out. Feel free to check out https://lichess.org/editor to play around with Chess pieces.

Part 1: Initialize the Game State

```
void initialize_game(ChessGame *game)
```

A `ChessGame` struct represents the state of an ongoing Chess game. A `ChessMove` struct represents a single move of the game.

```
typedef struct {
     ChessMove moves[MAX_MOVES];
     char capturedPieces[MAX_CAPTURED_PIECES];
     int moveCount;
     int capturedCount;
     int currentPlayer;
     char chessboard[8][8]; // [row #][col #]  0-based indexing
} ChessGame;

typedef struct {
    char startSquare[3];
    char endSquare[4];
} ChessMove;

// Example ChessMove
ChessMove move = {"g1", "f3"};
```



In your `chessboard` array within the `ChessGame` struct, you should have a representation of the chess board. Black's pieces are given in lowercase:

```
r: rook
n: knight
b: bishop
q: queen
k: king
p: pawn
```

Uppercase letters represent the same pieces, but for White. Below is what your expected `chessboard` field should look like within the struct after `initialize_game` returns. The bolded **`r`** is at `chessboard[0][0]`. A period indicates an empty space. Use the period character itself in the array.

```
rnbqkbnr
pppppppp
........
........
........
........
PPPPPPPP
RNBQKBNR
```

Set `moveCount` and `capturedCount` to 0. The `currentPlayer` field indicates whose turn it is: the constants `WHITE_PLAYER` (0) or `BLACK_PLAYER` (1). Set `currentPlayer` to `WHITE_PLAYER`. In the final client/server application, the client is the White player, and the server is Black. Individual player moves are stored in the `moves` array. Pieces that are captured are stored inside of the `capturedPieces` array. This array holds only single characters (r, P, n, Q, etc.)

Note that the `endSquare` field of the ChessMove struct has an extra character index. This is used to represent pawn promotions. More on this later.

Note

You have been provided the following helper function to print the state of a game board:

```
void display_chessboard(ChessGame *game) {
    printf("\nChessboard:\n");
    printf("  a b c d e f g h\n");
    for (int i = 0; i < 8; i++) {
        printf("%d ", 8 - i);
        for (int j = 0; j < 8; j++) {
            printf("%c ", game->chessboard[i][j]);
```

```
        }
        printf("%d\n", 8 - i);
    }
    printf("  a b c d e f g h\n");
}
```

The output of this function upon call after initialization and upon game start should be:

```
Chessboard:
  a b c d e f g h
8 r n b q k b n r 8
7 p p p p p p p p 7
6 . . . . . . . . 6
5 . . . . . . . . 5
4 . . . . . . . . 4
3 . . . . . . . . 3
2 P P P P P P P P 2
1 R N B Q K B N R 1
  a b c d e f g h
```

Remember that the bolded **`r`** at **a8** is stored in `chessboard[0][0]`.

Using `display_chessboard` to debug the state of your board while you develop. One key point to note is that the row #1 on a chess board is **on the bottom of the chess board**, and we store it on the `chessboard[7]`.

Part 2: Validating Piece Movement

In this part you will write a set of functions to test whether a given movement is valid for the piece located at the given row and column. Row and column numbers are 0-based. We assume that the piece at the given source position is of the expected type. For example, for `is_valid_rook_move` we assume that there is a rook at the provided position. If it is valid for the piece to move to the destination row and column, the function returns `true`. A “valid” move is defined below for each piece for the sake of the related function. Note that these functions do not need to consider who is the current player. The only exception to this is the pawn because its direction of movement is important. For instance, if a pawn is moving diagonally towards row #7 in `chessboard` (i.e., capturing a piece while moving towards the White player’s side of the board), the piece at the destination must be an uppercase letter.

If the movement is invalid, the function returns `false`.

```
bool is_valid_pawn_move(char piece, int src_row, int src_col,
                        int dest_row, ChessGame *game)

bool is_valid_rook_move(int src_row, int src_col, int dest_row,
                        int dest_col, ChessGame *game)

bool is_valid_knight_move(int src_row, int src_col, int dest_row,
                          int dest_col)

bool nt is_valid_bishop_move(int src_row, int src_col, int dest_row,
                             int dest_col, ChessGame *game)

bool is_valid_queen_move(int src_row, int src_col, int dest_row,
                         int dest_col, ChessGame *game)

bool is_valid_king_move(int src_row, int src_col, int dest_row,
                        int dest_col)

bool is_valid_move(char piece, int src_row, int src_col,
                   int dest_row, int dest_col, ChessGame *game)
```

The `is_valid_move` function will be the primary function that will call the appropriate helper function to do the error checking.

The functions that you will implement in addition to the `is_valid_move`:

`is_valid_pawn_move`

- Verifies that the pawn is moving forward by one or two squares, and in the correct direction based on the color.
- Checks if the pawn is capturing correctly.

`is_valid_rook_move`

- Verifies that the rook is moving horizontally or vertically.
- Verifies that there are no pieces blocking the path between the source and destination positions.

`is_valid_knight_move`

- Verifies that the knight is moving in an L-shape (two squares vertically and one square horizontally, or two squares horizontally and one square vertically).

`is_valid_bishop_move`

- Verifies that the bishop is moving diagonally.
- Verifies that there are no pieces blocking the path between the source and destination positions.

`is_valid_queen_move`

- Verifies that the queen is moving like a rook (horizontally or vertically) or like a bishop (diagonally).

`is_valid_king_move`

- Verifies that the king is moving one square in any direction (horizontally, vertically, or diagonally).

Return `true` if the move is valid or `false` if it is invalid.

`is_valid_move` should be quite straightforward: call the appropriate `is_valid` function from above, depending on the piece at `src_row` and `src_col`, and return the return value of that particular `is_valid` function. Return `false` if there is no piece at the given source location.

Part 3: Parsing Movement Strings

```
int parse_move(const char *str, ChessMove *move);
```

The `parse_move` parses the given `str` string and modifies the `ChessMove` structure. An example of a move could be `b3b5`, in the case of a rook. This means that the rook should be from `b3` to `b5` on the board. You also need to account for promotion of a pawn when it advances to the furthest row chess board. An example of a pawn promotion for White would be `e7e8q` . In this example, a pawn on `e7` moves to `e8` and is promoted to a Queen. Note that the letter used to indicate the chosen piece ('q', 'r', 'b', 'n'), is always indicated in lowercase, regardless of the player.

Your function must handle the error situations given below. The words in `ALL_CAPS` are macros for the relevant error code that your function must return. They are provided in `hw4.h`. In the event multiple errors occur, return the first one in the order that the constants appear below. Return 0 for a successful parsing of the `move` string.

`PARSE_MOVE_INVALID_FORMAT`

- Return this error code if the length of the `move` string is not equal to 4 or 5 characters.
- Return this error code if the row letter is not in the range `'a'` through `'h'`.
- Examples of invalid inputs that correspond to this error code: `e2f`, `k3k8`

`PARSE_OUT_OF_BOUNDS`:

- Return this error code if there is an attempt to move a piece outside of the 8x8 grid, or if the move reference a position outside the grid.
- Example of invalid inputs that correspond to this error code: `e9e7`, `a0a2`

`PARSE_MOVE_INVALID_DESTINATION`

- This error case applies only to strings of length 5, which correspond with pawn promotions.
- Return this error if the destination square is not on the appropriate row. Promotions for White must happen on row 8, and row 1 for Black.
- Example: `e2e4q` is invalid because you are not moving to the 8th or 1st row while attempting to promote a pawn. In contrast, `e7e8q` is acceptable, assuming a White pawn is at position `e7`.

`PARSE_MOVE_INVALID_PROMOTION`

- This error case applies only to strings of length 5, which correspond with pawn promotions.
- Return this error code if the `move` string has a length of 5 characters, but the promotion piece is not one of the valid options ('q', 'r', 'b', 'n').
- Examples of invalid inputs that correspond to this error code: `c7c8k`, `e7e8x` Note that `x` is not a valid piece.

Valid Example #1:

```
ChessGame game;
ChessMove move;
parse_move("e2e4", &move)
```

Assuming this is a valid move, `move.startSquare = "e2"` and `move.endSquare = "e4"`. These fields should be populated after a call to `parse_move()`.

Valid Example #2:

```
ChessGame game;
ChessMove move;
// this represents a promotion to Black bishop
parse_move("d7d8b", &move)
```

Assuming this is a valid move, `move.startSquare = "d7"` and `move.endSquare = "d8b"`.

Note that `parse_move` is in charge of all string-related errors (and not violations of game rules). Hence, you do not need a `ChessGame` struct.

Part 4: Moving Pieces

```
int make_move(ChessGame *game, ChessMove *move, bool is_client,
              bool validate_move);
```

In the previous part, you implemented parsing for a string representing a Chess move. Now you will implement the function that makes the move on the board and checks the game state for discrepancies. The assumption is that `parse_move` had been called to modify a `ChessMove` struct, and now `make_move` will use the struct to move a piece from one square to another. Refer to the other document as needed for the rules of movement in Chess.

The `is_client` argument tells us if this function is being called on behalf of the client (White) player (`is_client = true`), or the server (Black) player (`is_client = false`).

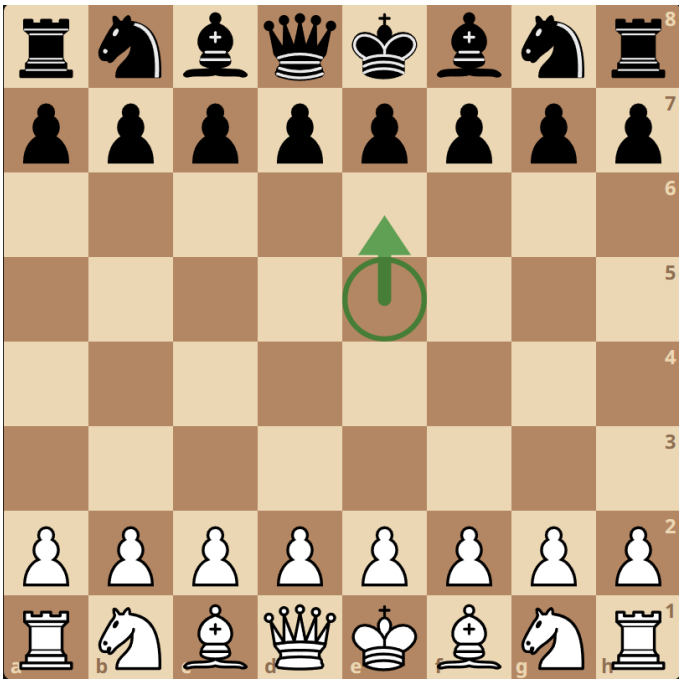
The `validate_move` argument tells us whether this call to `make_move` needs to validate the `move` struct itself or just assume that the contents of `move` are correct. This flag is useful during network communication. Later in the document we will tell you when `validate_move` should be `true` or `false`. Your code structure for `make_move` should look like this:

```
// any required variable declarations go here

if (validate_move) {

     // error checks go here

}
// update game->moves, game->chessboard, etc.
```

Below are the error cases you should be concerned with regarding this part. Check them in the order provided. Every other Chess rule is irrelevant and will not be tested; just implement what's given below. Return the constant corresponding to that error. Use the `is_valid_move` function to help you complete this part. Return 0 if there is no error with the provided move.

`MOVE_OUT_OF_TURN`: Moving out of turn is not allowed. If the client tries to move when it's the server's turn, or vice versa, return this error code.

`MOVE_NOTHING`: Attempting to move from an empty square is not allowed.



`MOVE_WRONG_COLOR`: Attempting to move the opponent's piece is not allowed. If the current player tries to move a piece of the opposite color, return this error code.

`MOVE_SUS`: Capturing your own pieces is not allowed.

`MOVE_NOT_A_PAWN`: The move string has a length of 5 characters (indicating a pawn promotion), but the piece at the start square is not a pawn ('P' or 'p'). You are not allowed to promote pieces that are not pawns. Example: `endSquare = e7e8q`, but there is no pawn on `e7`.

`MOVE_MISSING_PROMOTION`: The move string has a length of 4 characters, but the destination square is on the row for pawn promotion and the piece being moved is a pawn. Example: `endSquare = e8`. To be valid, `endSquare` would need to be something like `endSquare = e8q`.

`MOVE_WRONG`: Return this error code if `is_valid_move()` returned `false`.

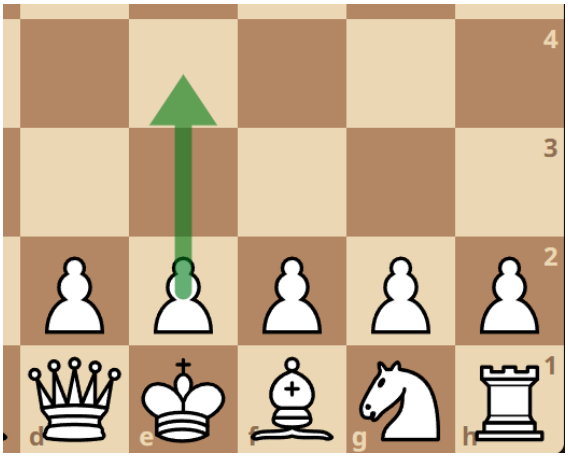
While moving the piece, you will have to promote this piece in `make_move` if the piece moves onto the last row for either side (White or Black) with a pawn.

Note that you still have to handle the color of the piece correctly. For example, a White pawn promoted to a queen should be a 'Q'. Likewise, a Black pawn promoted to a queen should be a ‘q’. These changes should be reflected in your `ChessGame` struct.

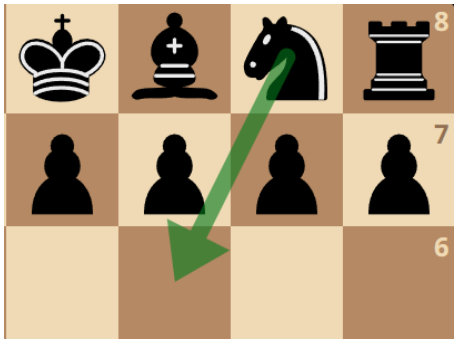
Now, assuming there was no error with the provided `ChessMove`, or if `validate_move = false`, update the state of the `ChessGame` struct. You will have to update the `ChessGame` structure after every move to reflect the current state of the game which includes the positions of pieces; the number of moves; captured pieces; pawn promotions, and which player's turn it is. More specifically, add the move to the end of the `moves` field of the `ChessGame` struct and increment `moveCount`. Likewise, store each capture piece in the `capturedPieces` field of the `ChessGame` struct. Increment the `capturedCount` field as needed.

Below are some examples of what this function should do.

```
ChessMove move = {"e2", "e4"};
```



```
ChessMove move = {"g8", "f6"};
```



We have defined `MAX_MOVES` as 512 and `MAX_CAPTURED_PIECES` as 16. The former is a hard limit on what you should expect for how many moves you can expect us to test. The latter is the ceiling for how many pieces you can theoretically capture in chess – the king piece is included in this, but more on that later.

For reference, an average chess game usually lasts around 40-60 moves. As for the latter, well, there are only 32 pieces on the board. Assume every piece can be captured by an opponent piece for sake of simplicity. This likely makes zero sense if you know the rules of chess, as the king is uncapturable normally, but it's for your benefit.

Finally, update the currentPlayer field in the `ChessGame` struct to the other player (`WHITE_PLAYER` or `BLACK_PLAYER`).

Part 5: FEN String Generation

```
void chessboard_to_fen(char fen[], ChessGame *game)
```

In this part, your task is to develop the logic to create a FEN (Forsyth-Edwards Notation) string, which gives a representation of a chess board's state. To simplify matters somewhat, we are using a slightly different version of true FEN string notation.

Here is a mapping of the letters we will be using in our FEN string:

```
r: rook
n: knight
b: bishop
q: queen
k: king
p: pawn
```

Uppercase letters represent the same pieces but for White. At the very end of the string is a space and a single character that indicates whose turn is next:

`b`: Black's turn to move
`w`: White's turn to move

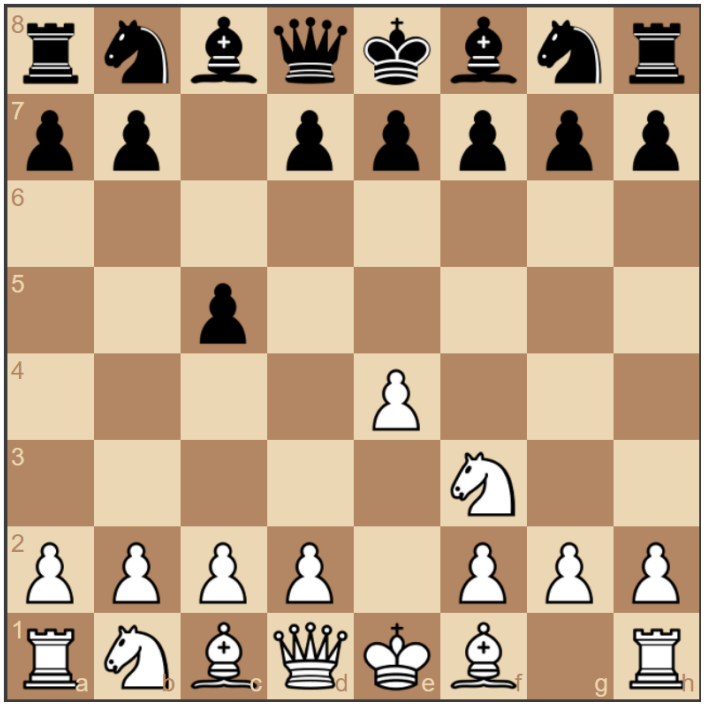
Example: The sequence below describes the piece placement field of the starting position of a game of chess. Each "/" separates a row from the next – starting with the top row.

```
rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w
```

The 8's represent empty tiles. You can verify that here with this board to see how the FEN string above describes the starting board.

There are usually special rules in Chess that need to be included in the FEN strings, but we will be excluding them.

Let us examine an example of generating a FEN string by hand:



In this example, we can see that a few moves have already been made in this chess game. FEN strings always start with the top row on the chess board. So in this case, the first piece we want to put in the FEN string based on the above board is the leftmost Black rook.

In this row, we have the rook, knight, and bishop mirrored on both sides of the queen and king. We now know then that our FEN string for this row is: `rnbqkbnr`

In this row, we have seven pawns. Two pawns are positioned on the far left, followed by an empty square, and then the remaining five pawns are adjacent to each other.

In FEN, empty squares are denoted by numbers corresponding to the count of consecutive empty squares. Therefore, we write the two pawns as 'pp', the single empty square as '1', and the subsequent five pawns as 'ppppp'. Combined, the FEN representation for this row is `pp1ppppp`.

`8` is the FEN string because there are 8 spaces.

`2p5` is the FEN string because there is a pawn separating 2 and 5 consecutive squares.

`4P3` is the FEN string because there is a White pawn separating 4 and 3 consecutive squares.

`5N2` is the FEN string because there is a White knight separating 5 and 2 consecutive squares.

`PPPP1PPP` is the FEN string because there is 1 space separating pawns.

`RNBQKB1R` is the FEN string

Lastly, we want to denote whose turn it is which in this case is Black so we append a space a `b` to the string. Finally, we have:

```
rnbqkbnr/pp1ppppp/8/2p5/4P3/5N2/PPPP1PPP/RNBQKB1R b
```

Part 6: FEN String Parsing

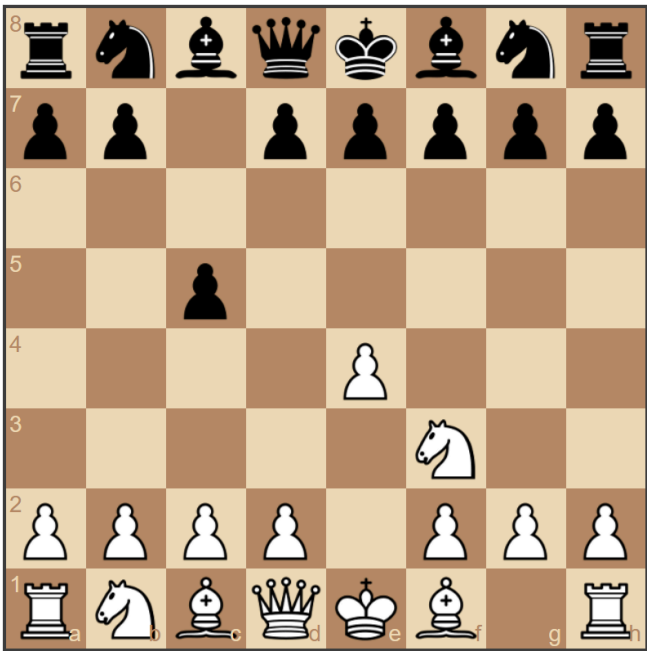
```
void fen_to_chessboard(const char *fen, ChessGame *game)
```

Now that you have implemented the functionality to create FEN strings, the next step is to add functionality to parse a given FEN string and reconstruct the board state it represents. We assume that `fen` contains a valid FEN string.

As an example of how `fen_to_chessboard` can be used, if the server uses the ***`/import FEN_string_here`*** command, then they must give a FEN string input. For the following example, we will input this FEN string:

`rnbqkbnr/pp1ppppp/8/2p5/4P3/5N2/PPPP1PPP/RNBQKB1R b`

which would then represent this respective board in a `ChessGame.chessboard` array:



```
rnbqkbnr/
pp1ppppp/
8/
2p5/
4P3/
5N2/
PPPP1PPP/
RNBQKB1R
```

Additionally, the `b` at the end of the FEN string indicates to us whose turn it is. So in this scenario, upon loading the game, the person prompted for a move will be Black. For ease of reading, below I have broken the FEN string into their respective rows so that you can verify for yourself that the image and the string do, indeed, correspond to each other.

Part 7: Save & Load Games

In this section, you will implement two functions that allow users to save and load the game progress using a simple file-based database. The `save_game` function saves the current game state along with the username, while the `load_game` function instead loads a previously saved game state from the username.

```
int save_game(ChessGame *game, const char *username,
              const char *db_filename);
```

The function arguments are:

- `game`: A pointer to a `ChessGame` structure representing the current state of the chess game.
- `username`: A string representing the username associated with the game state. The username may not contain spaces and cannot be empty.
- `db_filename`: The filename of the database file from which the game state will be loaded.

The algorithm to implement for the `save_game` function is as follows:

1. Open the database file.
2. Generate the FEN string of the current game state using the `chessboard_to_fen` function.
3. Write the username and FEN string to the file in the format `"username:fen\n"`. Return 0 to indicate a successful save operation. Return -1 on error.
4. In the event that the client has already saved before, write another entry to the end of the file. Do not delete/replace the previous save file.

The saved game states are stored in a simple text file format, with each line containing the username and FEN string separated by a colon (":"). Here's an example:

```
testuser:rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w
testuser:rnbqkbnr/pppp1ppp/8/4p3/4P3/8/PPPP1PPP/RNBQKBNR w
```

```
int load_game(ChessGame *game, const char *username,
              char *db_filename, int save_number);
```

The function arguments are:

- `game`: A pointer to the `ChessGame` structure where the loaded game state will be loaded into.
- `username`: A string representing the username associated with the game state to be loaded.
- `db_filename`: The filename of the database file from which the game state will be loaded.
- `save_number`: The save file to retrieve. Cannot be zero. Return -1 if the specified save file is not formatted as described above in the `load_game` specification.

The algorithm to implement for the `load_game` function is as follows:

1. Open the database file.
2. Read the file line by line, searching for a line for the given player. The argument `save_number` tells you which one of the saved games we want. For example, suppose `username = "Daniel"` and `save_number = 3`. This means we want to find the third game in the database for the user `"Daniel"`. Start counting at 1, not 0.
3. Search for matching game state. If no matching game state is found for the given username, return -1 to indicate that the game was not found.

Example of load/save in action during a live play. Assume the chess board state is as follows:

```
Chessboard:
  a b c d e f g h
8 r n b q k b n r 8
7 p p p p p p p p 7
6 . . . . . . . . 6
5 . . . . . . . . 5
4 . . . . P . . . 4
3 . . . . . . . . 3
2 P P P P . P P P 2
1 R N B Q K B N R 1
  a b c d e f g h
```

The following messages are entered on the client:

```
Enter a message for the server: /save Zhi Bin
Usernames cannot contain spaces.

Enter a message for the server: /save ZhiBin
Game saved successfully.
```

The client quits and comes back to play chess after some time and connects again:

```
Enter a message for the server: /chessboard

Chessboard:
  a b c d e f g h
8 r n b q k b n r 8
7 p p p p p p p p 7
6 . . . . . . . . 6
5 . . . . . . . . 5
4 . . . . . . . . 4
3 . . . . . . . . 3
2 P P P P P P P P 2
```

```
1 R N B Q K B N R 1
  a b c d e f g h
```

Notice that the initial state upon joining the server is a blank slate with no pieces moved.

```
Enter a message for the server: /load ZhiBin 0
Save file not found for the given username and save number.

Enter a message for the server: /load ZhiBin 1
Game loaded successfully.
```

Chess board state for both server and client after this exchange:

```
Enter a message for the server: /chessboard

Chessboard:
  a b c d e f g h
8 r n b q k b n r 8
7 p p p p p p p p 7
6 . . . . . . . . 6
5 . . . . . . . . 5
4 . . . . P . . . 4
3 . . . . . . . . 3
2 P P P P . P P P 2
1 R N B Q K B N R 1
  a b c d e f g h
```

Now both the server and client are updated with the board played previously.

Part 8: Send and Receive Messages

```
int send_command(ChessGame *game, const char *message,
                 int socketfd, bool is_client);
int receive_command(ChessGame *game, const char *message,
                 int socketfd, bool is_client);
```

In this part, you will implement the two key functions indicated above. These two functions will handle some of the communication between the client and server, allowing them to exchange various commands related to the chess game.

The `send_command` function is responsible for sending commands from either the client or server to the other party by using the `send()` system call. `send_command` takes four parameters:

- `game` is a pointer to the `ChessGame` structure representing the current state of the chess game.
- `message` is the message to be processed by the function
- `socketfd` is the socket file descriptor needed for communication
- `is_client`: a boolean that indicates if the function was called on behalf of the client (`true`) or server (`false`).

For this assignment, the commands that `send_command` must handle are:

`/move` (takes a string representing a move as an argument)
`/forfeit` (takes no arguments)
`/chessboard` (takes no arguments)
`/import` (takes a FEN string as argument)
`/load` (takes a username and game number as arguments)
`/save` (takes a username to save the file under)

The possible return values for `send_command` are:

`COMMAND_MOVE`: Indicates a valid `/move` command.

`COMMAND_FORFEIT`: Indicates a `/forfeit` command.

`COMMAND_IMPORT`: Indicates an `/import` command.

`COMMAND_UNKNOWN`: Indicates an unknown or invalid command. Return this if no command can be matched.

`COMMAND_ERROR`: Indicates an error occurred during command execution. Return this if a command is attempted but errors occur. This is explained further in the algorithm pseudocode below.

Examples of these commands are given below:

```
/move e2e4
/forfeit
/chessboard
/import rnbqkbnr/pp1ppppp/8/2p5/4P3/5N2/PPPP1PPP/RNBQKB1R b
/load Daniel 12
/save
```

The algorithm to implement for `send_command` is as follows:

1. If the command is `/move`, call `parse_move` to verify that the move is valid. If so, call `make_move` (with `validate_move = true`) and ,assuming that the call to `make_move` returned without error, call `send` to send `message` over the socket; return `COMMAND_MOVE`. Otherwise, for any errors detected by `parse_move` or `make_move`, return `COMMAND_ERROR`.
2. If the command is `/forfeit`, call `send` to send `message` over the socket. Return `COMMAND_FORFEIT`.
3. If the command is `/chessboard`, call `display_chessboard` and return `COMMAND_DISPLAY`.
4. If the command is `/import` and this function was called on behalf of the server (`is_client = false`), call `fen_to_chessboard` and then call `send` to send a `message` over the socket. Return `COMMAND_IMPORT`.
5. If the command is `/load` and its arguments are valid, call `load_game` to update the game state and then call `send` to send `message` over the socket. Use `game_database.txt` as the name of the database file. Return `COMMAND_LOAD`. If the arguments to the `/load` command are invalid or no game could be loaded with the provided username and game number, return `COMMAND_ERROR`.
6. If the command is `/save`, save the game in `game_database.txt` with the client's username by calling `save_game`. Return `COMMAND_SAVE`. If the argument to the `/save` command is invalid or the game could not be saved, return `COMMAND_ERROR`.

If none of the above commands can be matched, return `COMMAND_UNKNOWN`.

The `receive_command` function is similar to `send_command`, but does have some subtle differences. The primary purpose of `receive_command` is to sift through the inputs from the `send_command` function. The algorithm to implement for `receive_command` is as follows:

1. If the command is `/move`, call `parse_move` to verify that the move is valid. Return `COMMAND_ERROR` for any errors with the call to `parse_move`. Otherwise, call `make_move` (with `validate_move = false`) and then return `COMMAND_MOVE`.
2. If the command is `/forfeit`, close the socket. Return `COMMAND_FORFEIT`.
3. If the command is `/import` and this function was called on behalf of the server (`is_client = true`), call `fen_to_chessboard` and return `COMMAND_IMPORT`.
4. If the command is `/load`, call `load_game` to update the game state. Return `COMMAND_LOAD`. If the arguments to the `/load` command are invalid or no game could be loaded with the provided username and game number, return `COMMAND_ERROR`.

If none of the above commands can be matched, return -1.

Part 9: Client/Server

`hw4_server.c` - server application
`hw4_client.c` - client application

For the final part, you will complete the starter code provided in the client and server C files indicated above. Base your code on the simple chat client/server program provided on the course website. Once the server is started, you should then be able to run the client program in order to connect to the server. After building your code, the server should be run using `./build/hw4_server` and client connect through `./build/hw4_client`. You might find it helpful to run these in programs in separate tabs in VS Code, or in a split bash Terminal. All communication takes place over port `8080`, as given in the macro `PORT`.

Immediately after the client has connected to the server, call the `initialize_game` function in both the client and server. This initialization must happen before the game loops start executing. Optionally, call `display_chessboard`.

The algorithm for the server's main while loop is given below. This code is executed after connection from the client has been accepted.

1. Call `read` to accept the client's input over the socket.
2. Call `receive_command` to process the client's input. If the client request is to forfeit, close the socket and terminate the server.
3. Prompt the server player for input.
4. Call `send_command` to attempt sending the command to the client. If `send_command` returns `COMMAND_ERROR` or `COMMAND_UNKNOWN`, prompt the server again for a different command. If the server's command is to forfeit, close the socket and terminate the server.

Client's main while loop:

1. Prompt the client player for input.
2. Call `send_command` to send the command to the server. If `send_command` returns `COMMAND_ERROR` or `COMMAND_UNKNOWN` or `COMMAND_SAVE`, prompt the client again for a different command. If the client's command is to forfeit, close the socket and terminate the client.
3. Call `read` to accept the server's input over the socket.
4. Call `receive_command` to process the server's input. If the server's command is to forfeit, close the socket and terminate the client.

Template Repository

Starter code is available on CodeGrade and GitHub, as with previous assignments.

In addition, we have provided you two executables named `hw4_client_soln` and `hw4_server_soln` which you can use to understand the **SOME** of expected behavior of the entire program. The executables are not 100% robust, and are not intended to be. They provide a sample of some features and do **NOT** necessarily implement the entirety of the project. Type `./hw4_server_soln` at the command line in one terminal window, and `./hw4_client_soln` in a different window. The client initiates the "slash" commands, so start typing input in the client's window first.

Building and Running Your Code at the Command Line

1. Configure the code to build. This needs to be done only once:
   ```
   cmake -S . -B build
   ```
2. Build the code after changing the code:
   ```
   cmake --build build
   ```
3. Run the test cases.
   ```
   ./build/run_tests
   ```

Testing & Grading Notes

Note: any evidence of hard-coding test case input or output will automatically result in a score of zero for the assignment. This kind of behavior is borderline academic dishonesty.

Generative AI

Generative AI may not be used to complete this assignment.

Academic Honesty Policy

Academic honesty is taken very seriously in this course. By submitting your work for grading you indicate your understanding of, and agreement with, the following Academic Honesty Statement:

1. I understand that representing another person's work as my own is academically dishonest.
2. I understand that copying, even with modifications, a solution from another source (such as the web or another person) as a part of my answer constitutes plagiarism.

3. I understand that sharing parts of my homework solutions (text write-up, schematics, code, electronic or hard-copy) is academic dishonesty and helps others plagiarize my work.
4. I understand that protecting my work from possible plagiarism is my responsibility. I understand the importance of saving my work such that it is visible only to me.
5. I understand that passing information that is relevant to a homework/exam to others in the course (either lecture or even in the future!) for their private use constitutes academic dishonesty. I will only discuss material that I am willing to openly post on the discussion board.
6. I understand that academic dishonesty is treated very seriously in this course. I understand that the instructor will report any incident of academic dishonesty to the University's Academic Judiciary.
7. I understand that the penalty for academic dishonesty might not be immediately administered. For instance, cheating on a homework assignment may be discovered and penalized after the grades for that homework have been recorded.
8. I understand that buying or paying another entity for any code, partial or in its entirety, and submitting it as my own work is considered academic dishonesty.
9. I understand that there are no extenuating circumstances for academic dishonesty.