Project #1: Checkers, Due: Feb. 19th, 11:59p.m.

In this project, you will build a program that allows two human players to play the game of checkers. Your program will graphically maintain the state of the game board and prompt the players for moves on the standard input/output. Your program will also know the rules of checkers, and will check the validity of moves to maintain the integrity of the game board as the game progresses. Sections 1–7 will describe the rules of checkers, and the functionality that your program must support.

To encourage you to adopt the functional decomposition programming disciplines discussed in class, you will be given much of the decomposition for this project. In particular, we will provide a header file that specifies the interfaces for several functions. **Your solution must implement these interfaces.** Section 8 will describe the required functional decomposition.

1 Rules of Checkers

Checkers is played on an 8 by 8 grid, or “checker board,” with alternating dark and light squares. The board starts with 12 red and 12 white pieces, each situated on the 12 dark squares at opposing ends of the board. (Actually, the entire game is played on the dark squares only). Players take turns making moves, with the player of the red pieces moving first. Two kinds of moves are allowed. A player can “step” a piece one square diagonally if the diagonally adjacent square is unoccupied. Alternatively, if a player’s piece is next to an opponent’s piece, and the square beyond it is free, the player can “jump” over the opponent’s piece onto the unoccupied square. The opponent’s piece is removed from the board after the jumping move.

It is possible to jump many times in a row with the same piece, capturing several of the opposing pieces on the same turn. Also, whenever a jumping move is possible, the player must make a jumping move on that turn; a stepping move can only be made on turns where jumping moves are not possible.

In the beginning, pieces can only move and jump forward. However, if a piece reaches the far end of the board, then it becomes a “king.” A king is allowed to move and jump diagonally backwards and forwards. Kings can be captured like any other piece.
2 Maintaining the Game Board

Your checkers game should maintain the state of the game board, and print it to standard output using text characters. The squares of the board should be printed using the ‘-’ and ‘|’ characters; the row and column numbers of the board squares should be printed above and to the left of the board; and squares containing pieces should be labeled “r” and “w” for normal red and white pieces, respectively, and “R” and “W” for red and white pieces that are kings. For example, at the very beginning of the game (before any moves have been made), the board should look like:

```
0 1 2 3 4 5 6 7
|---|---|---|---|---|---|---|---|
0|   | r |   | r |   | r |   | r |
|---|---|---|---|---|---|---|---|
1|r |   | r |   | r |   | r |   |
|---|---|---|---|---|---|---|---|
2|   | r |   | r |   | r |   | r |
|---|---|---|---|---|---|---|---|
3|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
4|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
5| w |   | w |   | w |   | w |   |
|---|---|---|---|---|---|---|---|
6|   | w |   | w |   | w |   | w |
|---|---|---|---|---|---|---|---|
7| w |   | w |   | w |   | w |   |
|---|---|---|---|---|---|---|---|
```

3 Moving Pieces

Your checkers program should prompt each player for a move, alternating between the red and white pieces. For example, if it’s the red pieces’ turn to move, your program should say:

RED’s move:

At the prompt, your program should expect the corresponding player to enter 4 numbers: the first 2 numbers specify the column and row, in that order (i.e., an x-y coordinate), of a square containing the piece to move, and the second 2 numbers specify the column and row of an empty square to move to. After a valid move is entered (see Section 4), your program should update the state of the game board, print the updated game board, and then prompt the opposing player for the next move.
4 Verifying Moves

After a player enters a move, but before the game board is updated, your program should verify that the move entered is a valid move. Your program should check several conditions:

1. The move should originate from and terminate to squares on the board (i.e., the move should stay on the 8x8 grid).
2. The originating square must contain a piece belonging to the player making the move, while the terminating square must be empty.
3. The move must be a legal stepping move or jumping move. A legal stepping move must move 1 square away along a diagonal. Normal pieces can only move in the “forward direction” (for red, this means towards increasing y coordinates, and for white, this means towards decreasing y coordinates); kings can move in both the forward and backward directions.
4. A legal jumping move must move 2 squares away along a diagonal with the “jumped” square containing an opponent’s piece. Similar to stepping moves, normal pieces can only jump in the forward direction; kings can jump in both the forward and backward directions.
5. On any given turn, if a player can perform a jumping move from any one of his/her pieces, only a jumping move from one of those candidate jumping pieces constitutes a valid move. During such a turn, a stepping move is not valid.

If a move meets all these conditions, it is a valid move, and your program should update the game board accordingly. (For jumping moves, this includes removing the piece that was jumped). Otherwise, your program should discard the move, print the error message “INVALID MOVE. TRY AGAIN!!”, and prompt the same player for another move. Re-prompting continues until a valid move is entered.

5 Kings

Whenever a normal piece lands on a square in the far row of the board (rows 7 and 0 for red and white pieces, respectively), the piece becomes a king. When this happens, your program should capitalize the letter used to represent the piece on the game board (i.e., change “r” to “R” and “w” to “W” for red and white pieces, respectively). As described earlier, kings can step and jump in both the forward and reverse directions.

6 Multiple Jumps

Normally, after a player enters a valid move, your program should prompt the opposing player for a move. The exception is multiple jumps by a player on the same turn. If a
player makes a jumping move that lands on a square from which another jump can be made, instead of prompting the opposing player for a move, your program should prompt the same player again for another move. In this case, the only valid move is a jumping move with the piece involved in the multi-jump sequence. After a second jumping move is entered by the player, your program should again test to see if the piece can perform another jump, and if so, prompts the same player again for another move, and so on. Hence, your program will prompt the same player N times for a multi-jump sequence involving N jumps. When the jumping piece finally lands on a square where no further jumps can be made, then your program should prompt the opposing player for a move.

7 Termination

Normally, a game of checkers ends when a player captures all of his/her opponent’s pieces, or until all of a player’s pieces are blocked so that they cannot move. However, in this project, you do not need to detect these game-ending conditions. Instead, your program should continue play until a player enters a move from square (0, 0) to square (0, 0)—in other words, the player enters 4 zeros when prompted for a move. Otherwise, your program should simply process moves, alternating between players.

8 Project Design

As discussed in class, functional decomposition is crucial for building large programs. It maximizes ease of understanding, incremental testing and debugging, as well as reuse of your code. For this project, your code is required to implement the functions in Section 8.1. The discussion in Section 8.1 not only describes what each required function does, it also specifies the arguments and return values for each required function. Your code must implement these interfaces exactly as described.

8.1 Required Functions

There are 7 required functions. These functions are described below, and are listed in the “checkers.h” header file provided on the course website (follow the hyperlink for “Project 1 Files”). In the description that follows, references are made to constants. These constants are in all uppercase letters, and are also defined in the checkers.h file.

1. void print_board()

This function prints the checkers board in the format described in Section 2, indicating the location of all the game pieces. This function has no arguments and no return values.

2. int jump_exists(int color)
This function takes a “color” as an argument, either RED or WHITE. For the specified
color, the function computes whether a jumping move is possible by the corresponding
player. If the player has at least one jumping move to make, the function returns TRUE.
Otherwise, it returns FALSE.

As discussed in Sections 1 and 4, a player must make a jumping move if a jumping move
is possible. Before prompting a player for a move and checking its validity, this function
should be called to determine whether a jumping move is expected.

3. int check_move(int color, int x_from, int y_from, x_to, y_to, int jump)

This function checks to see whether or not an entered move is valid. The function takes 6
arguments. The first argument, “color,” specifies the player making the move, either RED
or WHITE. The arguments “x_from” and “y_from” specify the square to move from, and
the arguments “x_to” and “y_to” specify the square to move to. Finally, the argument,
“jump,” specifies whether the move is a jumping move (TRUE or FALSE). The function
computes whether or not the specified move by the specified player meets the criteria for a
valid move described in Section 4. Accordingly, the function returns either VALID_MOVE
or INVALID_MOVE.

Everytime a player enters a move, your program must check the validity of the entered
move. This function provides the check.

4. void move_piece(int color, int x_from, int y_from, int x_to, int y_to,
int jump)

This function performs a specified move, modifying the checkers board to reflect the new
location of the moved piece, as well as removing an opponent's piece during a jumping
move. The arguments for the move_piece function are identical to those for the check_move
function. This function should be called after check_move has verified that a move is valid.

5. int check_step(int x_from, int y_from, x_to, y_to, int dir)

This function checks whether or not a move, specified by the four parameters x_from,
y_from, x_to, and y_to, constitutes a stepping move in either the positive or negative
direction, as specified by the dir argument. If dir is set to +1, the check is made for the
positive direction, or the “forward direction” for red pieces (i.e., the stepping move goes
towards increasing y coordinates). If dir is set to -1, the check is made for the negative
direction, or the “forward direction” for white pieces (i.e., the stepping move goes towards
decreasing y coordinates). If the move is a stepping move in the specified direction, the
function returns VALID_MOVE; otherwise, it returns INVALID_MOVE.

This function should be called (possibly multiple times) from the check_move function to
verify some of the criteria for a valid move given in Section 4.

6. int check_jump(int x_from, int y_from, x_to, y_to, int dir)
This function checks whether or not a move, specified by the four parameters \( x_{\text{from}}, y_{\text{from}}, x_{\text{to}}, \) and \( y_{\text{to}} \), constitutes a jumping move in either the positive or negative direction, as specified by the \( \text{dir} \) argument. If \( \text{dir} \) is set to +1, the check is made for the positive direction, or the “forward direction” for red pieces (\( i.e., \) the jumping move goes towards increasing \( y \) coordinates). If \( \text{dir} \) is set to -1, the check is made for the negative direction, or the “forward direction” for white pieces (\( i.e., \) the jumping move goes towards decreasing \( y \) coordinates). If the move is a jumping move in the specified direction, the function returns \text{VALID\_MOVE}; otherwise, it returns \text{INVALID\_MOVE}.

Like \text{check\_step}, \text{check\_jump} should be called (possibly multiple times) from the \text{check\_move} function to verify some of the criteria for a valid move given in Section 4.

7. \text{int is\_jumper(int x, int y)}

This function checks the square on the board at location \( x, y \). If a the square contains a piece, it checks whether the piece can make a jumping move, taking into consideration the color of the piece (\text{RED} or \text{WHITE}), and whether the piece is a normal piece or a king. Accordingly, the function returns \text{TRUE} or \text{FALSE}.

This function should be called from multiple places. In \text{check\_move}, \text{is\_jumper} can be used to enforce the fact that a jumping move is being performed when the \text{jump} argument is \text{TRUE}. \text{is\_jumper} can also be used by \text{jump\_exists} to test the jumping status of every piece on the board belonging to a player. Lastly, \text{is\_jumper} can be used to determine whether a player can continue jumping after a jumping move has been made (\( i.e., \) a multiple-jump move).

8.2 Board Data Structure

Your program should implement the checkers board using an 8x8 two-dimensional array. Each array element (square) should store an integer value that encodes whether the square is empty or whether it contains a piece. If the latter, the integer value should also encode whether the piece belongs to the red or white player, and whether it is a normal piece or a king.

For this project, you should implement the game board data structure as a global variable, and use it in all the functions that need to access the game board. As discussed in class, global variables are generally a bad idea because they break procedural abstraction. Unfortunately, more topics must be covered before we are able to pass arrays as parameters between functions. So, if we want to access the same game board data structure across different functions, we have no choice but to use a global variable.