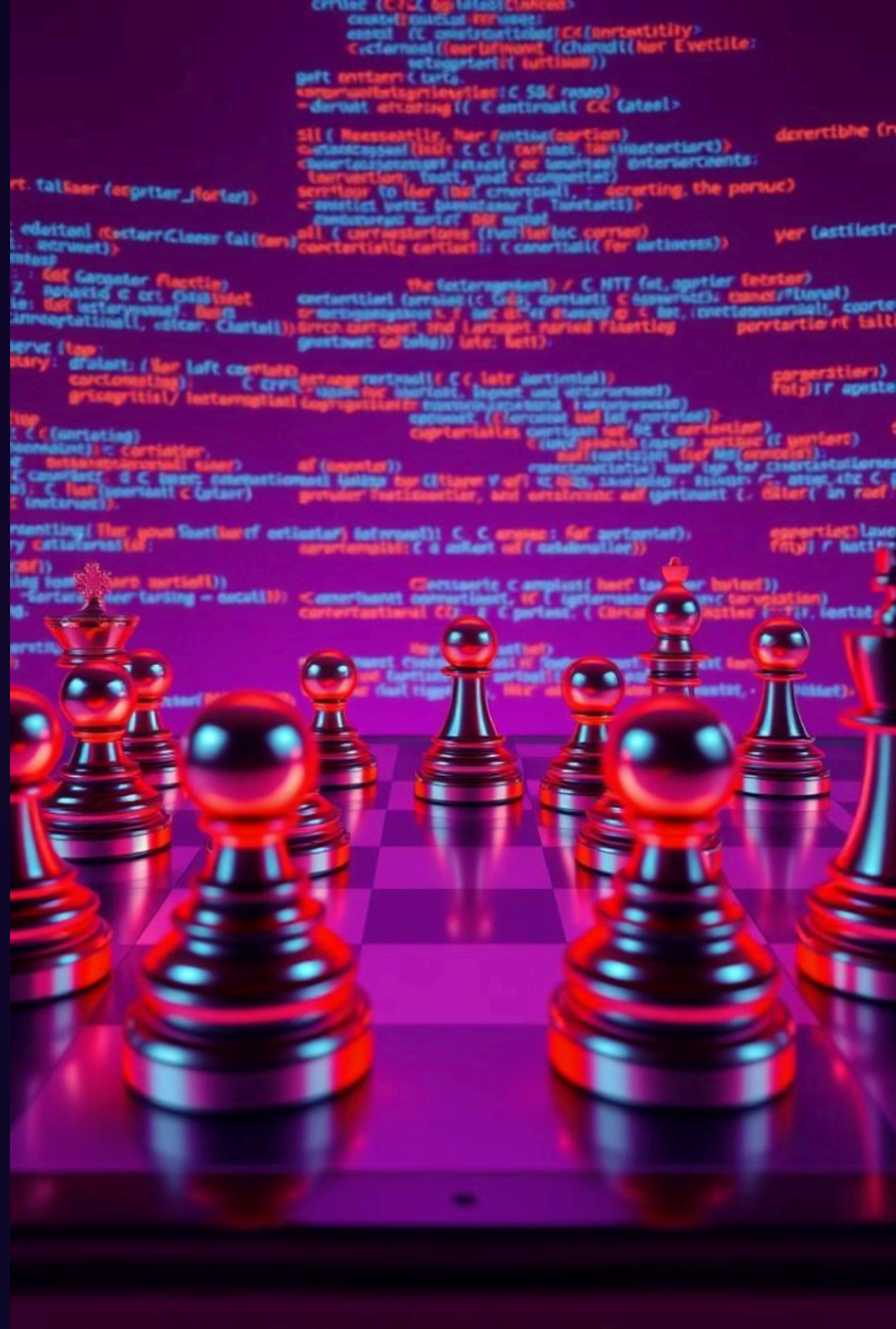
C Chess Game Implementation Overview

This C program implements a text-based chess game. It enables player-vs-bot gameplay, featuring board initialization, move validation, and checkmate detection. The code structure emphasizes modularity and follows chess rules closely.

by ChessBot Team



```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define SIZE 8

int board[SIZE][SIZE];
int lastBotMoveX = -1; // Track last moved piece's coordinates for the bot
int lastBotMoveY = -1;
int botConsecutiveMoves = 0; // Track consecutive moves of the same piece
```

```
// Function to initialize the board
void initBoard() {
    for (int i = 0; i < SIZE; i++) {
        board[1][i] = 1;   // White pawns
        board[6][i] = -1;  // Black pawns
    }
    board[0][0] = board[0][7] = 2;   // White rooks
    board[7][0] = board[7][7] = -2;  // Black rooks
    board[0][1] = board[0][6] = 3;   // White knights
    board[7][1] = board[7][6] = -3;  // Black knights
    board[0][2] = board[0][5] = 4;   // White bishops
    board[7][2] = board[7][5] = -4;  // Black bishops
    board[0][3] = 5;                  // White queen
    board[7][3] = -5;                 // Black queen
    board[0][4] = 6;                  // White king
    board[7][4] = -6;                 // Black king
    for (int i = 2; i < 6; i++) {
        for (int j = 0; j < SIZE; j++) {
            board[i][j] = 0;
        }
    }
}
```

```c
// Function to display the board
void displayBoard() {
    printf("    1   2   3   4   5   6   7   8\n");
    printf("  +---+---+---+---+---+---+---+---+\n");
    for (int i = 0; i < SIZE; i++) {
        printf("%d |", i + 1);
        for (int j = 0; j < SIZE; j++) {
            if (board[i][j] == 0) {
                printf("   |");
            } else {
                printf("%3d|", board[i][j]);
            }
        }
        printf("\n");
        printf("  +---+---+---+---+---+---+---+---+\n");
    }
    printf("\n");
    printf("Piece Legend:\n");
    printf("  1 = White Pawn, 2 = White Rook, 3 = White Knight, 4 = White Bishop, 5
= White Queen, 6 = White King\n");
    printf(" -1 = Black Pawn, -2 = Black Rook, -3 = Black Knight, -4 = Black Bishop,
-5 = Black Queen, -6 = Black King\n");
}
```

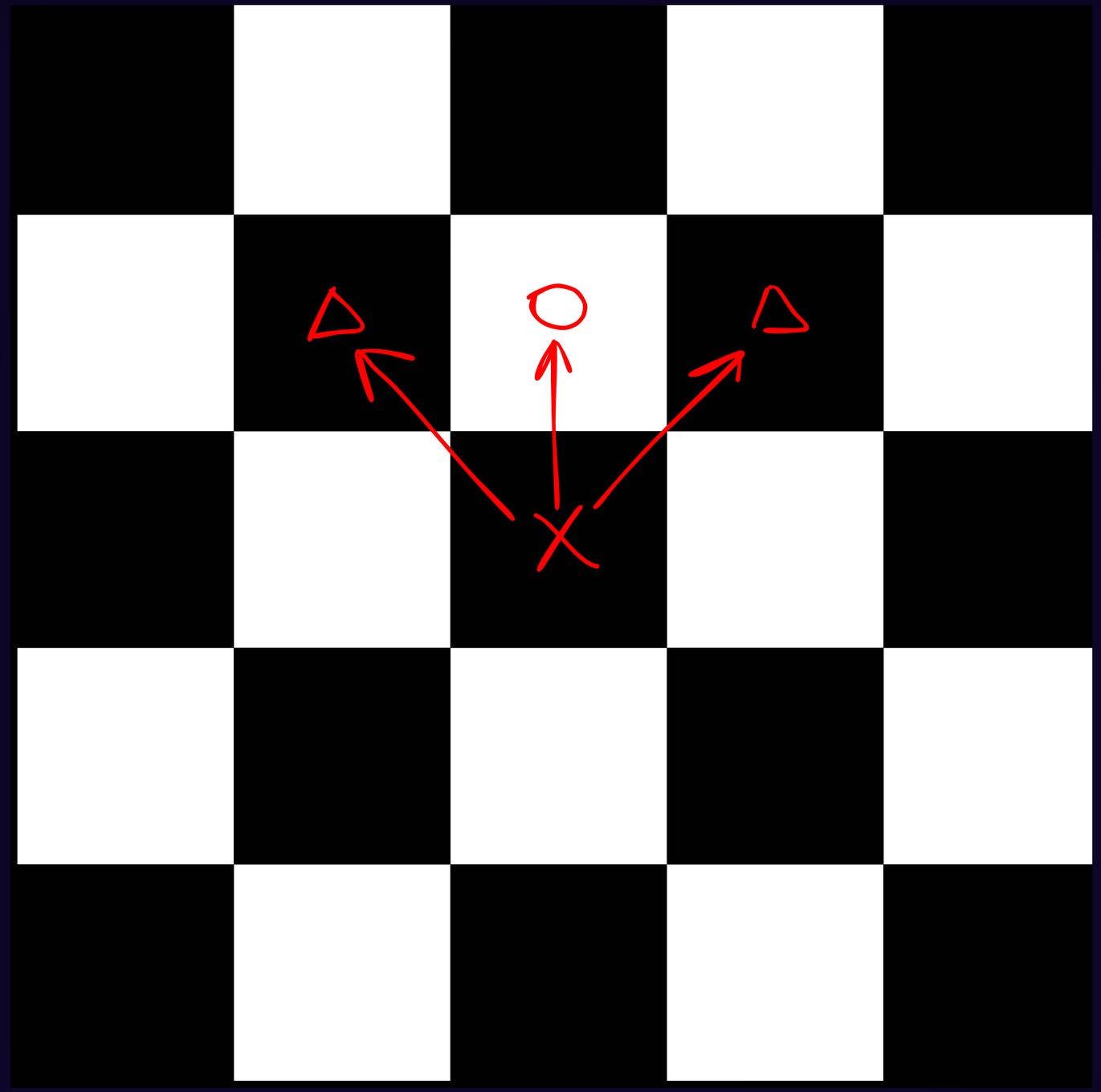
```
Initial Board:
    1   2   3   4   5   6   7   8
  +---+---+---+---+---+---+---+---+
1 |  2|  3|  4|  5|  6|  4|  3|  2|
  +---+---+---+---+---+---+---+---+
2 |  1|  1|  1|  1|  1|  1|  1|  1|
  +---+---+---+---+---+---+---+---+
3 |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
4 |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
5 |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
6 |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
7 | -1| -1| -1| -1| -1| -1| -1| -1|
  +---+---+---+---+---+---+---+---+
8 | -2| -3| -4| -5| -6| -4| -3| -2|
  +---+---+---+---+---+---+---+---+

Piece Legend:
  1 = White Pawn, 2 = White Rook, 3 = White Knight, 4 = White Bishop, 5 = White Queen, 6 = White King
 -1 = Black Pawn, -2 = Black Rook, -3 = Black Knight, -4 = Black Bishop, -5 = Black Queen, -6 = Black King
Enter move (fromX fromY toX toY) or 'end' to exit:
```

```
// Function to evaluate the value of a piece
int pieceValue(int piece) {
    switch (piece) {
        case 1: return 1;   // Pawn
        case 2: return 5;   // Rook
        case 3: return 3;   // Knight
        case 4: return 3;   // Bishop
        case 5: return 9;   // Queen
        case 6: return 100; // King (very high value)
        default: return 0;  // No value
    }
}
```

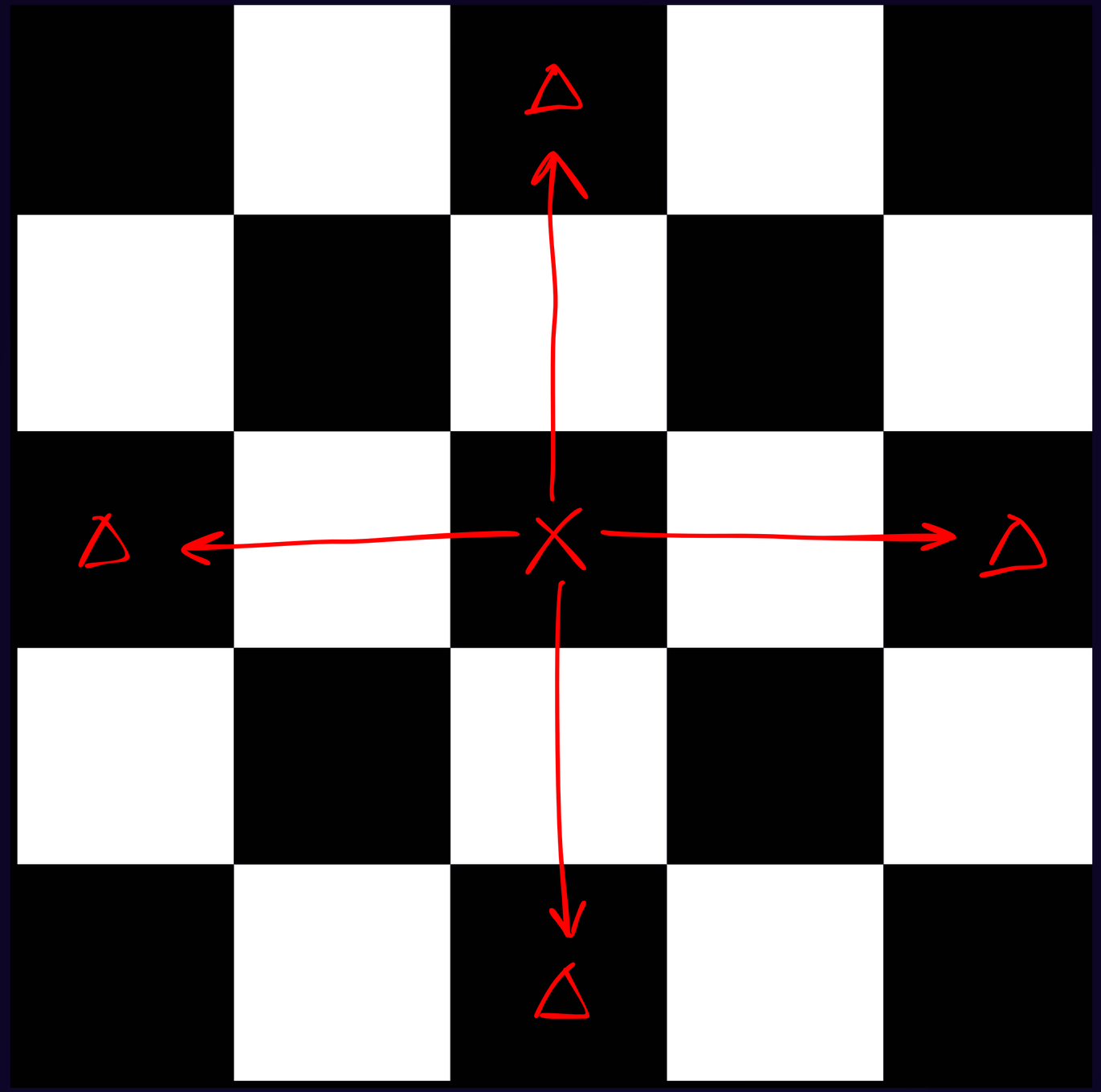
Move Validation Functions

1 Piece-Specific Validation

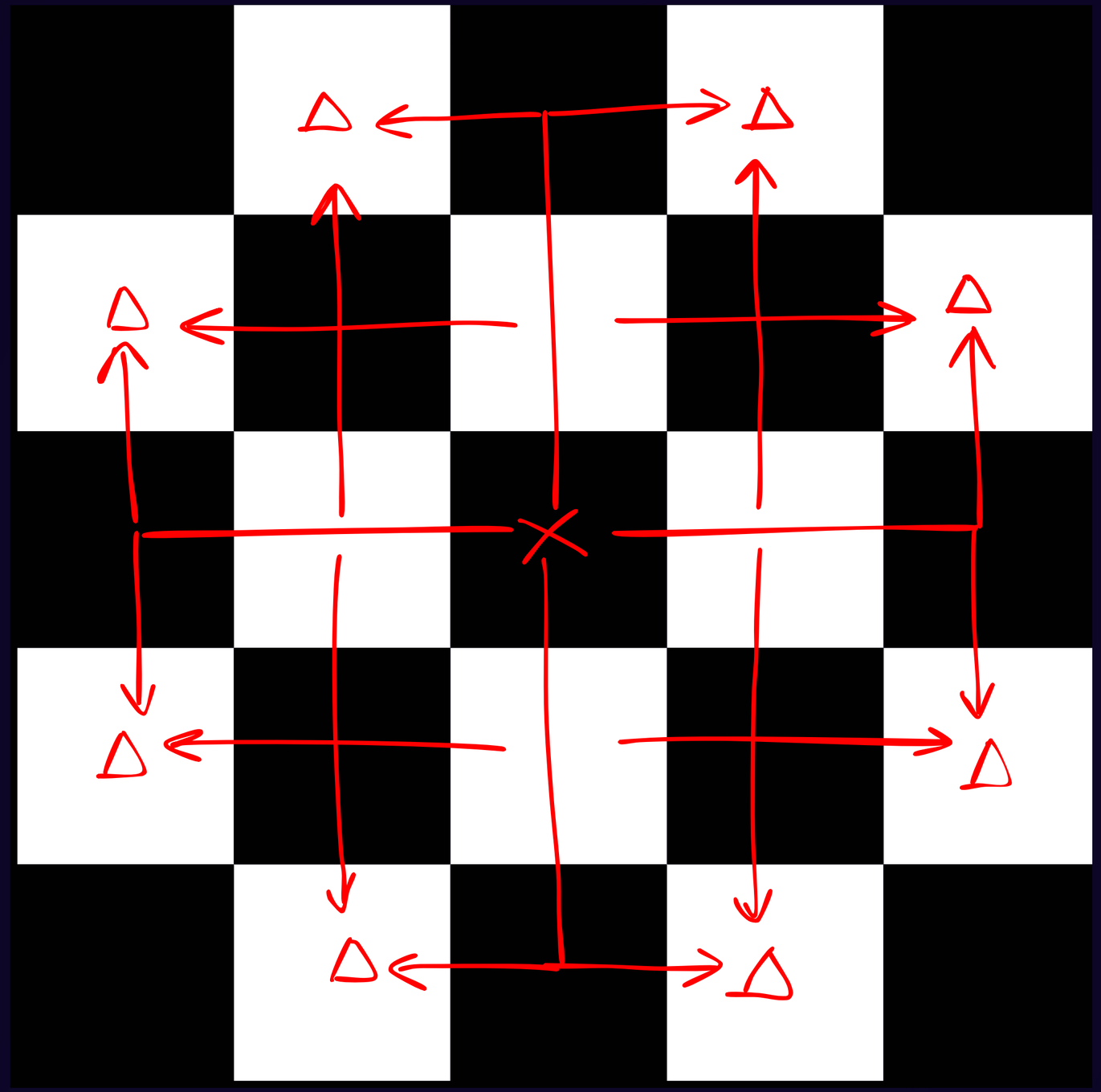
Separate functions for each piece type (e.g., isValidPawnMove, isValidRookMove). Ensures moves adhere to chess rules.



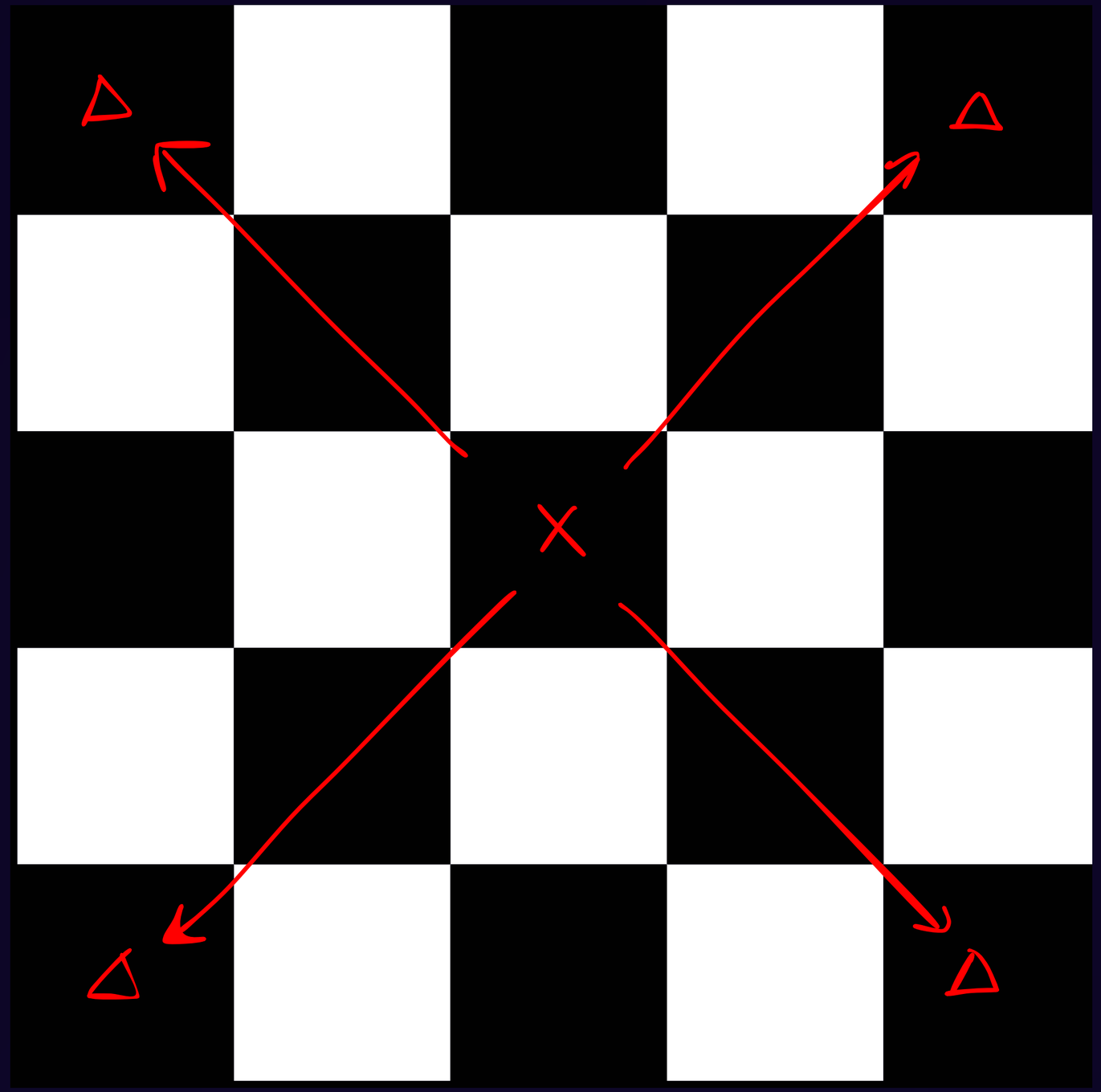
```
// Function to determine if a move is valid for a pawn
int isValidPawnMove(int fromX, int fromY, int toX, int toY) {
    if (toX == fromX + 1 && toY == fromY && board[toX][toY] == 0) {
        return 1; // Move forward
    }
    if (toX == fromX + 1 && (toY == fromY - 1 || toY == fromY + 1) && board[toX]
[toY] < 0) {
        return 1; // Capture
    }
    if (toX == fromX - 1 && toY == fromY && board[toX][toY] == 0) {
        return 1; // Move forward for black pawn
    }
    if (toX == fromX - 1 && (toY == fromY - 1 || toY == fromY + 1) && board[toX]
[toY] > 0) {
        return 1; // Capture for black pawn
    }
    return 0; // Invalid move
}
```



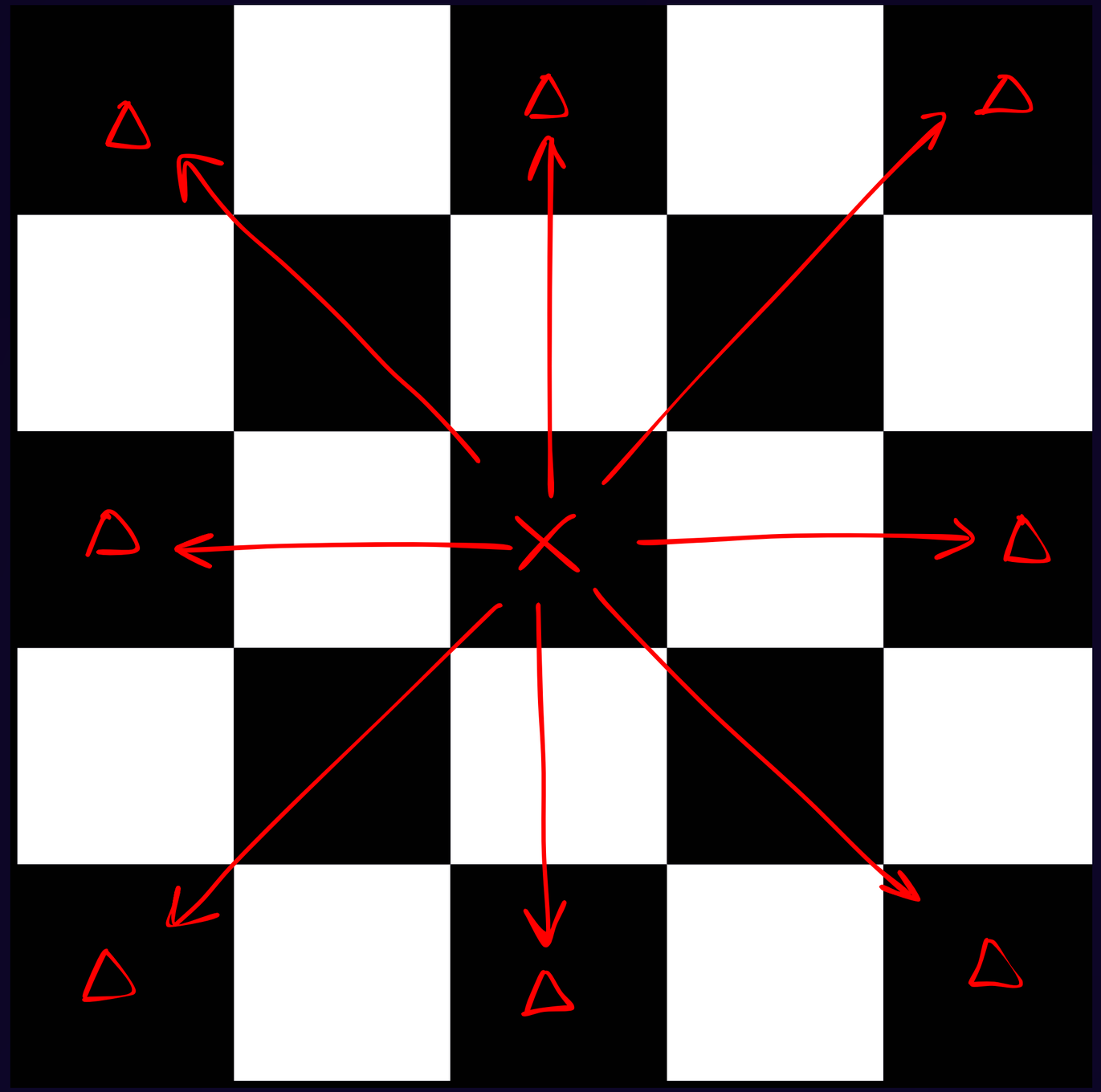
```
// Function to determine if a move is valid for a rook
int isValidRookMove(int fromX, int fromY, int toX, int toY) {
    if (fromX != toX && fromY != toY) return 0; // Rook moves in straight lines only

    int stepX = (toX - fromX) ? (toX - fromX) / abs(toX - fromX) : 0;
    int stepY = (toY - fromY) ? (toY - fromY) / abs(toY - fromY) : 0;
    for (int x = fromX + stepX, y = fromY + stepY; x != toX || y != toY; x += stepX,
y += stepY) {
        if (board[x][y] != 0) return 0; // Must be empty
    }
    return board[toX][toY] * board[fromX][fromY] <= 0; // Capture or empty square
}
```



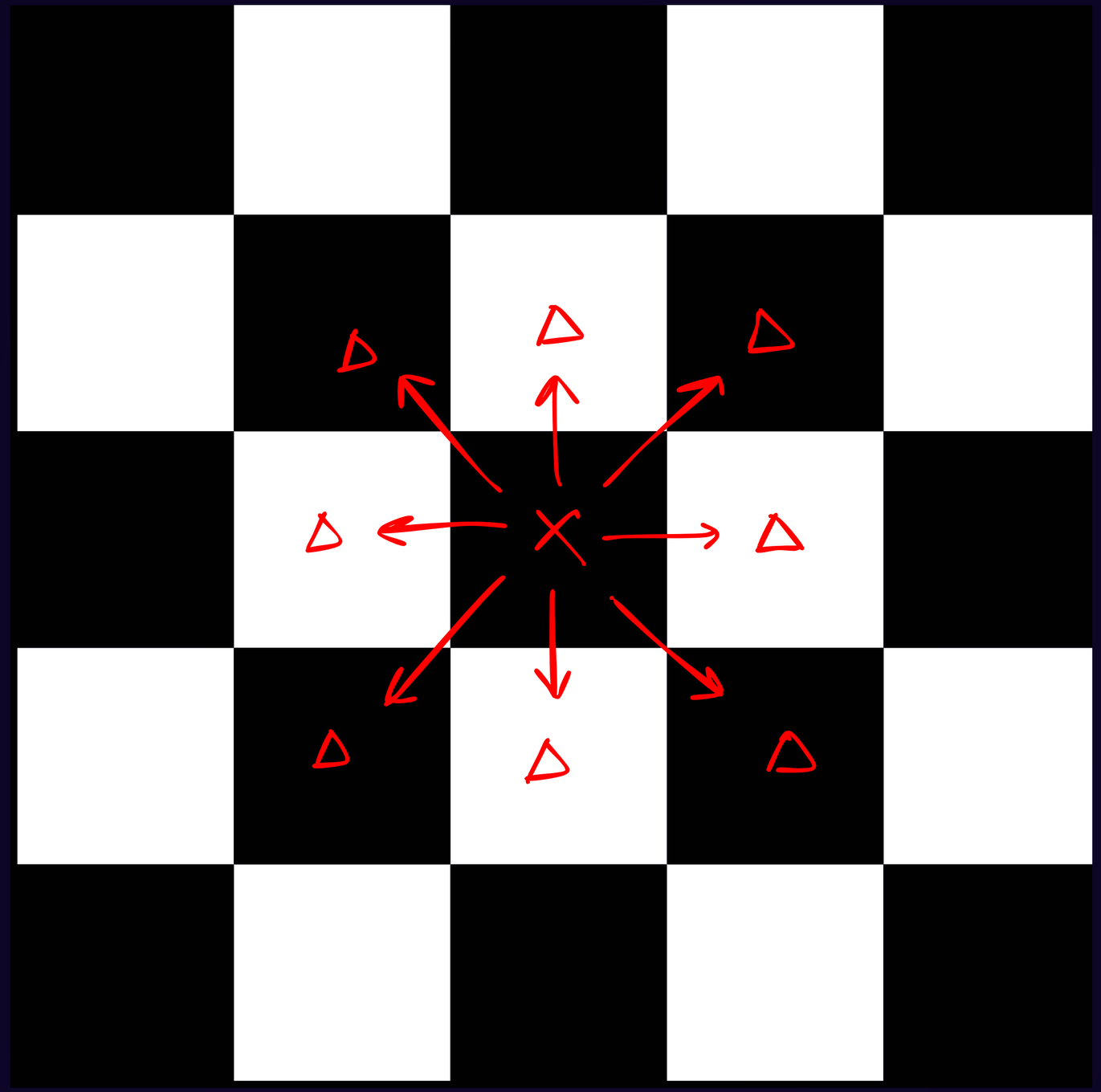
```
// Function to determine if a move is valid for a knight
int isValidKnightMove(int fromX, int fromY, int toX, int toY) {
    int dx = abs(toX - fromX);
    int dy = abs(toY - fromY);
    return (dx == 2 && dy == 1) || (dx == 1 && dy == 2) && (board[toX][toY] *
board[fromX][fromY] <= 0);
}
```



```
// Function to determine if a move is valid for a bishop
int isValidBishopMove(int fromX, int fromY, int toX, int toY) {
    if (abs(fromX - toX) != abs(fromY - toY)) return 0; // Bishop moves diagonally
    int stepX = (toX - fromX) / abs(toX - fromX);
    int stepY = (toY - fromY) / abs(toY - fromY);
    for (int x = fromX + stepX, y = fromY + stepY; x != toX || y != toY; x += stepX,
y += stepY) {
        if (board[x][y] != 0) return 0; // Must be empty
    }
    return board[toX][toY] * board[fromX][fromY] <= 0; // Capture or empty square
}
```



```
// Function to determine if a move is valid for a queen
int isValidQueenMove(int fromX, int fromY, int toX, int toY) {
    if (abs(fromX - toX) == abs(fromY - toY)) {
        int stepX = (toX - fromX) / abs(toX - fromX);
        int stepY = (toY - fromY) / abs(toY - fromY);
        for (int x = fromX + stepX, y = fromY + stepY; x != toX || y != toY; x +=
stepX, y += stepY) {
            if (board[x][y] != 0) return 0; // Must be empty
        }
    } else if (fromX != toX && fromY != toY) return 0; // Queen moves in straight
lines or diagonally
    else {
        int stepX = (toX - fromX) ? (toX - fromX) / abs(toX - fromX) : 0;
        int stepY = (toY - fromY) ? (toY - fromY) / abs(toY - fromY) : 0;
        for (int x = fromX + stepX, y = fromY + stepY; x != toX || y != toY; x +=
stepX, y += stepY) {
            if (board[x][y] != 0) return 0; // Must be empty
        }
    }
    return board[toX][toY] * board[fromX][fromY] <= 0; // Capture or empty square
}
```



```
// Function to determine if a move is valid for a king
int isValidKingMove(int fromX, int fromY, int toX, int toY) {
    int dx = abs(toX - fromX);
    int dy = abs(toY - fromY);
    return (dx <= 1 && dy <= 1) && (board[toX][toY] * board[fromX][fromY ] <= 0);
}
```

Move Validation Functions

2 canMove()

General move validation function. Calls piece-specific functions based on the moving piece's type.

```
// Function to determine if a move is valid for a piece
int canMove(int fromX, int fromY, int toX, int toY) {
    int piece = board[fromX][fromY];
    if (piece == 1) { // White pawn
        return isValidPawnMove(fromX, fromY, toX, toY);
    }
    if (piece == -1) { // Black pawn
        return isValidPawnMove(fromX, fromY, toX, toY);
    }
    if (piece == 2 || piece == -2) { // Rook
        return isValidRookMove(fromX, fromY, toX, toY);
    }
    if (piece == 3 || piece == -3) { // Knight
        return isValidKnightMove(fromX, fromY, toX, toY);
    }
    if (piece == 4 || piece == -4) { // Bishop
        return isValidBishopMove(fromX, fromY, toX, toY);
    }
    if (piece == 5 || piece == -5) { // Queen
        return isValidQueenMove(fromX, fromY, toX, toY);
    }
    if (piece == 6 || piece == -6) { // King
        return isValidKingMove(fromX, fromY, toX, toY);
    }
    return 0; // Only supporting pawn and other pieces moves right now
}
```

Bot AI Implementation

1

Piece Evaluation

Bot assesses potential moves based on piece values and strategic considerations.

2

Move Selection

Chooses the highest-scoring move. Implements a basic rule to avoid repetitive moves.

3

Execution

Bot executes the selected move, updating the board state accordingly.

```
// Function for the bot's move
void botMove() {
    int bestScore = -1;
    int bestFromX = -1, bestFromY = -1, bestToX = -1, bestToY = -1;

    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            if (board[i][j] < 0) { // Bot's piece (black)
                for (int newX = 0; newX < SIZE; newX++) {
                    for (int newY = 0; newY < SIZE; newY++) {
                        if (canMove(i, j, newX, newY)) {
                            // Check if the target square is empty or occupied by an
opponent's piece
                            if (board[newX][newY] >= 0) { // Check if it's empty or
```

```
                                int score = pieceValue(board[newX][newY]) -
pieceValue(board[i][j]);

                                // Prioritize upward movement (i.e., smaller newX
value is better)

                                if (newX < i) {
                                    score += 10; // Bonus for moving upwards
                                }
                                if (score > bestScore) {
                                    bestScore = score;
                                    bestFromX = i;
                                    bestFromY = j;
                                    bestToX = newX;
                                    bestToY = newY;
                                }
                            }
                        }
                    }
                }
            }
        }
    }
```

```
 // Execute the best move if found
    if (bestScore != -1) {
        if (lastBotMoveX == bestFromX && lastBotMoveY == bestFromY) {
            botConsecutiveMoves++;
            if (botConsecutiveMoves < 2) {
                printf("Bot moves from (%d, %d) to (%d, %d).\n", bestFromX + 1,
bestFromY + 1, bestToX + 1, bestToY + 1);
                board[bestToX][bestToY] = board[bestFromX][bestFromY];
                board[bestFromX][bestFromY] = 0;
                lastBotMoveX = bestFromX;
                lastBotMoveY = bestFromY;
            } else {
                printf("Bot cannot move the same piece more than twice in a
row.\n");
            }
        } else {
            botConsecutiveMoves = 0;
            printf("Bot moves from (%d, %d) to (%d, %d).\n", bestFromX + 1,
bestFromY + 1, bestToX + 1, bestToY + 1);
            board[bestToX][bestToY] = board[bestFromX][bestFromY];
            board[bestFromX][bestFromY] = 0;
            lastBotMoveX = bestFromX;
            lastBotMoveY = bestFromY;
        }
    } else {
        printf("Bot has no valid moves.\n");
    }
```

Move Validation Functions

3 isAttacked()

Checks if a square is under threat. Crucial for implementing check and checkmate logic.

```
// Function to check if a square is attacked by the opponent
int isAttacked(int x, int y, int opponent) {
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            if (board[i][j] == opponent) {
                if (canMove(i, j, x, y)) {
                    return 1; // The square is attacked
                }
            }
        }
    }
    return 0; // The square is not attacked
}
```

```
// Function to check for checkmate
int isCheckmate(int player) {
    int kingX, kingY;

    // Find the king's position
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            if (board[i][j] == player) {
                kingX = i;
                kingY = j;
                break;
            }
        }
    }
```

```
// Check if the king is in check
 if (!isAttacked(kingX, kingY, -player)) {
     return 0; // The king is not in check
 }
```

```
// Check if there are any valid moves for the player
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            if (board[i][j] == player) { // The player's piece
                for (int newX = 0; newX < SIZE; newX++) {
                    for (int newY = 0; newY < SIZE; newY++) {
                        if (canMove(i, j, newX, newY)) {
                            // Simulate the move
                            int temp = board[newX][newY];
                            board[newX][newY] = player; // Move the piece
                            board[i][j] = 0; // Empty the original square

                            // Check if the king is still in check after the move
                            if (!isAttacked(kingX, kingY, -player)) {
                                // Undo the move
                                board[i][j] = player;
                                board[newX][newY] = temp;
                                return 0; // There is a valid move to escape check
                            }

                            // Undo the move
                            board[i][j] = player;
                            board[newX][newY] = temp;
                        }
                    }
                }
            }
        }
    }

    return 1; // The king is in check and has no valid moves (checkmate)
}
```

Game Flow and Win Conditions

1

Player Move

User inputs move. Program validates and executes if legal.

2

Bot Move

AI selects and executes move. Board updates after each turn.

3

Checkmate Check

Program checks for checkmate after each move. Game ends if detected.

```c
// Main function
int main() {
    initBoard();
    printf("Initial Board:\n");
    displayBoard();

    while (1) {
        // Player input
        char input[10];
        printf("Enter move (fromX fromY toX toY) or 'end' to exit: ");
        fgets(input, sizeof(input), stdin);

        // Check if the input is "end"
        if (strcmp(input, "end\n") == 0) {
            break;
        }

        int fromX, fromY, toX, toY;
        sscanf(input, "%d %d %d %d", &fromX, &fromY, &toX, &toY);

        // Adjust for 0-indexed array
        fromX -= 1;
        fromY -= 1;
        toX -= 1;
        toY -= 1;
```

```
        // Validate and execute the player's move
        if (canMove(fromX, fromY, toX, toY)) {
            board[toX][toY] = board[fromX][fromY];
            board[fromX][fromY] = 0;
            printf("Player moves from (%d, %d) to (%d, %d).\n", fromX + 1, fromY +
1, toX + 1, toY + 1);
        } else {
            printf("Invalid move.\n");
            continue;
        }

        // Check for checkmate after the player's move
        if (isCheckmate(6)) {
            printf("Checkmate! Black wins!\n");
            break;
        }

        // Bot move
        botMove();

        // Check for checkmate after the bot's move
        if (isCheckmate(-6)) {
            printf("Checkmate! White wins!\n");
            break;
        }

        displayBoard();
    }

    return 0;
}
```

Potential Enhancements

Advanced AI

Implement more sophisticated decision-making algorithms for the bot player.

Move History

Add functionality to undo moves and review game history.

Networking

Implement multiplayer functionality for online chess matches.

QR Code to GitHub: