Ryan Moskovciak
CS 3503 – W04
10/13/2024 Fall Semester

Bit Board Checkers Game Documentation

Introduction

Description: Bit Board Checkers is a full-functioning classic checkers game. Its implementation allows candidates to compete vigorously on a digital representation of checkers utilizing the standard rules. The game supports both regular and jump moves, including sequences of jumps. Additionally, there is an implementation for Kinging of pieces, allowing players to move forward and backward. This version of checkers calculates each available move and presents it to the user, except when jumps are available. Jumps are considered mandatory to take if they occur, so each user will only have jump options presented to them if they are given. The score of captured pieces is kept and displayed to the players; once a user collects all their opponent's pieces or places their opponent in a position where they cannot move but continue to have pieces on the board, they will automatically forfeit the game.

Fundamental Rules/Features:

a. **Support for single and multiple jumps.**
b. **King functionality for backward and forward movement.**
c. **Ability to king during a sequence of jumps.**
d. **When a king is captured, they will revert to a standard piece.**
e. **Win by collecting opponents' pieces and preventing them from moving.**
f. **Mandatory jumps move if available.**

Purpose: I intend to develop this application to demonstrate and enhance my understanding of bit manipulation in a real-world context. Creating this application increased my comprehension of leveraging bit manipulation to improve the efficiency of my applications and presented my ability to problem-solve and work through a problem from start to finish. Luckily, little information was available to create this specific implementation of a bit of a board game. Requiring me to discover the solution on my own.

Target Audience: Anyone who enjoys a game of checkers.

System Requirements / Tools

Operating System: Cross-Platform

Source Code Language: C++

Integrated Development Environment: VS Code

Compiler: Microsoft Visual C++ (MSVC)

Repository: Github

Installation and Setup

Clone the Repository: git clone https://github.com/ryanMHub/BitboardCheckers.git

Compile the Game: Navigate to the project folder and compile using a C++17 compatible compiler.

Windows:

a. cd BitboardCheckers
b. g++ -o checkers.exe main.cpp Game.cpp Board.cpp -std=c++17

Linux/macOS:

a. cd BitboardCheckers
b. g++ -o checkers main.cpp Game.cpp Board.cpp -std=c++17

Run the Game: After compiling run the game

Windows: checkers.exe

Linux/macOS: ./checkers

Code Structure

Harnessing the object-oriented aspects of C++, this application was broken up into core responsibilities. In addition, classes, structs, constants, and enums were constructed to facilitate easy access for all application parts, utilizing them as libraries. Additionally, all bit manipulation activities have their class called BitUtilities.

Breakdown:

- **Checkers.cpp -** Contains the main function and the overall game controller.
- **MoveManager.cpp/.h** – Is a singleton that handles all the decision-making for determining, presenting, acquiring move, and executing moves
- **View.cpp/.h** – A static class that handles user displays.
- **Player.h** – Is an object used to represent each active player in the game and store player resources.
- **Types.h** – Contains necessary structs, constants, and Enums used throughout the application.
- **BitUtilities.cpp/.h** – handles all bit manipulation requirements

Game Management

This section of the application handles the core functionality of the checkers game, including initialization, the game flow, and the calling of each subsequent process.

Key Components:

- **GameController -**
    - Manages the game loop, handling player turns, updating the board, and processing moves with MoveManager. This loop will process until one player is out of pieces or moves.

```cpp
//This function is used to drive the control of the game
void gameController(unsigned int* board, unsigned int* kings){
    //Initialize players using their player codes
    Player playerOne = Player(PlayerCode::PLAYERONE);
    Player playerTwo = Player(PlayerCode::PLAYERTWO);

    //Intialize the arrays that will store each players main board of pieces and their kings,
    board[playerOne.getCode()] = playerOne.getMain();
    board[playerTwo.getCode()] = playerTwo.getMain();
    kings[playerOne.getCode()] = playerOne.getKing();
    kings[playerTwo.getCode()] = playerTwo.getKing();

    //Initialize the moveManager singleton which will be used to process all of the required m
    MoveManager* moveManager = MoveManager::getInstance();

    initializeBoard(board); //initialize the starting positions for each players main boards
    PlayerCode current = playerOne.getCode(); //initialize the starting player
    bool running = true; //initialize the state of the game
    int scores[] = {0, 0}; //start each player at a score of 0

    //TODO: Determine how I'm going to declare winner based on blocked or less pieces
    //loop while running is true, changes when a player collects all pieces, or player has no
    while(running){
        View::displayCurrentBoard(current, board, kings, scores);
        View::pause();
        View::clear();
        running = moveManager->moveController(current, board, kings, scores);
        //TODO: needs to check both king and main boards
        std::tie(scores[0], scores[1]) = checkScore(board, kings);
        running &= (scores[0] == TOTAL_PIECES || scores[1] == TOTAL_PIECES)?false:true;
        current = static_cast<PlayerCode>(BitUtilities::flipNumber(current));
        View::clear();
    }

    gameOver(scores, current);
}
```

Controls Each Iteration of Game

- **Board Initialization -**
    - Set up the initial state of each player's piece at the start of the game.

```cpp
//initialize the game boards of each player with the given array
void initializeBoard(unsigned int* board){
    board[0] = 0x00000FFF; //initialize player 1
    board[1] = 0xFFF00000; //initialize player 2
}
```

Initialize Standard Boards to Starting State

- **Check Score -**
    - Counts the remaining kings and standard pieces utilizing the bit utility class. Returning a tuple of the remaining pieces is deducted from the total pieces. Each value returned represents the number of pieces each player has captured.

Player One	Player Two
8	8

Players Score Display

```cpp
//return each players score by counting
std::tuple<int, int> checkScore(unsigned int* board, unsigned int* kings){
    int playerOne = BitUtilities::countBits(0, board);
    playerOne += BitUtilities::countBits(0, kings);
    int playerTwo = BitUtilities::countBits(1, board);
    playerTwo += BitUtilities::countBits(1, kings);

    return std::make_tuple((TOTAL_PIECES-playerTwo), (TOTAL_PIECES-playerOne));
}
```

Check Score Implementation

- **Game Over -**
    - Determines who and how won the game, builds the required strings, and passes to the View class to display the results to the user.

```
//displays the winner of the game
void gameOver(int scores[], PlayerCode current){
    string player = "";
    string message = "";

    if(scores[0] < TOTAL_PIECES && scores[1] < TOTAL_PIECES){
        player = (current == PLAYERONE)?"One":"Two";
        string loser = ((current == PLAYERONE)?"Two":"One");
        message = "                    Player " + loser +" Forfeits";
    } else {
        player = (scores[0] == TOTAL_PIECES)?"One":"Two";
        string left = (scores[0] == TOTAL_PIECES)?std::to_string(TOTAL_PIECES-scores[1]):std::to_string(TOTAL_PIECES-scores[0]);
        message = "               Player " + player + " Has " + left + " Pieces Left";
    }
    View::displayWinner(player, message);
}
```

Determines Who Won Game and How

```
******************** Player Two Wins ************************
                   Player One Forfeits
Press Any Key..................
```

One of the Game Over Displays

Move Management

The Move Manager is a singleton class responsible for move-related functionality. It determines all available moves, presents them to the user, acquires the user's selection, and executes that move.

Key Components:

- **Move Controller -** Manages each step through the process of each player's turn. As well as the root of all required collections for the process.

```cpp
//execute all of the required functionality for the move process
bool MoveManager::moveController(PlayerCode current, unsigned int* board, unsigned int* kings, int score[]){
    //initialize int moves, map index to 'Char' increment using above idea, map 'Char' to set of struct move
    unsigned int movesBoard = 0;
    std::map<int, char> indexToChar;
    std::map<char, std::vector<Move>> charToPiece;

    //call function to get moves passing all above as reference.
    buildMovesMap(current, board, kings, movesBoard, indexToChar, charToPiece);

    //if there are no available moves player loses return false for unsuccessful move
    if(movesBoard == 0) return false;

    //display moves with modified displayBoard
    View::displayPlayerMoves(current, board, kings, movesBoard, indexToChar, score);

    char key;
    int index;

    //prompt user for selection
    std::tie(key, index) = getUserSelection(charToPiece);

    //make move on the board
    executeMove(current, board, kings, charToPiece[key].at(index));

    //move was a success
    return true;
}
```

Move Controller Drives Each Step of the Moving Process

- **Building Moves Mapping -** This function steps through each index of a number's bits and determines if there is a move available for it. Then, it routes it in the proper direction. After building a vector of all potential moves. Those moves are mapped to their location on the board using uppercase letters as keys.

```cpp
void MoveManager::buildMovesMap(PlayerCode current, unsigned int* board, unsigned int* kings, unsigned int& movesBoard, std::map<int, char>& indexToChar, std::map<char, vector<Move>>& charToPiece) {
    char moveMarkerCounter = 'A';
    std::vector<Move> moves;
    for(int i = 0 ; i < 32 ; i++){
        if(BitUtilities::checkBit(board[current], i) == 0 && BitUtilities::checkBit(kings[current], i) == 0) continue; //check if there is a piece at the position, if not continue to the next locati

        Border border = checkBorder(i); //check the position of the piece relative to the borders and store its state
        bool isKing = BitUtilities::checkBit(kings[current], i); //check if the current piece is a king
```

Initializing Important values

```
//Determine all directions that the current piece can move based on its location
if(border == TOP || border == RIGHT_TOP) {
    if(border == TOP){
        checkMove(current, isKing, border, moves, board, kings, i, (i + steps[0][0][0]));
        checkMove(current, isKing, border, moves, board, kings, i, (i + steps[0][0][1]));
    } else {
        checkMove(current, isKing, border, moves, board, kings, i, (4+i));
    }
} else if((border == BOTTOM || border == LEFT_BOTTOM)){
    if(border == BOTTOM){
        checkMove(current, isKing, border, moves, board, kings, i, (i + steps[1][1][0]));
        checkMove(current, isKing, border, moves, board, kings, i, (i + steps[1][1][1]));
    } else {
        checkMove(current, isKing, border, moves, board, kings, i, (-4+i));
    }
} else if(border == LEFT  || border == RIGHT){
    checkMove(current, isKing, border, moves, board, kings, i, ((current == PLAYERONE)?(4+i):(-4+i)));
    if(isKing) checkMove(current, isKing, border, moves, board, kings, i, ((current == PLAYERTWO)?(4+i):(-4+i)));
} else {
    checkMove(current, isKing, border, moves, board, kings, i, (i + steps[current][((i/4)%2)][0]));
    checkMove(current, isKing, border, moves, board, kings, i, (i + steps[current][((i/4)%2)][1]));
    if(isKing) {
        checkMove(current, isKing, border, moves, board, kings, i, (i + steps[BitUtilities::flipNumber(current)][((i/4)%2)][0]));
        checkMove(current, isKing, border, moves, board, kings, i, (i + steps[BitUtilities::flipNumber(current)][((i/4)%2)][1]));
    }
}
```

Determining where to send each index

- **Check Move Validity -** Checks if the move is valid, a single move, or a potential-jump because there is an enemy within distance.

```
void MoveManager::checkMove(PlayerCode current, bool isKing, Border border, std::vector<Move>& moves, unsigned int* board, unsigned int* kings, int i, int movePosition){
    //if there is already the currents player's piece at the destination return
    if(BitUtilities::checkBit(board[current], movePosition) || BitUtilities::checkBit(kings[current], movePosition)){
        return ;
    } else if(BitUtilities::checkBit(board[BitUtilities::flipNumber(current)], movePosition) || BitUtilities::checkBit(kings[BitUtilities::flipNumber(current)], movePosition)){ //if t
        std::vector<int> opponents;
        getJumpMoves(moves, Move(i, -1, false, isKing, opponents), current, board, kings, i, movePosition, BitUtilities::mergeBits(board[BitUtilities::flipNumber(current)], kings[BitU
    } else {
        //else just log a normal move
        std::vector<int> opponents;
        moves.push_back(Move(i, movePosition, false, isKing, opponents));
    }
}
```

Checks Validity routes accordingly

- **Handling Jumps -** Jumps determination is the most complicated portion of the code base. A recursive depth-first search approach was used to build the graph of each piece that has a single or sequence of jumps available to them. It also makes cases for king directionality and the ability to king during a jump sequence. This process is embodied in four functions: the main driver recursive function getJumpMoves, and three helper functions that determine if it is a valid jump, and the list of opponents that are available for a new location.

```cpp
//recursive function that builds the jump tree for each piece. Stores the individual Move struct for each jump in the moves vector<Move>.
bool MoveManager::getJumpMoves(std::vector<Move>& moves, Move move, PlayerCode current, unsigned int* board, unsigned int* kings, int i, int opponent, unsigned int currOpps){
    //if no open space or out or bounds return
    int dest;
    bool success;
    std::tie(success, dest) = checkJump(current, board, kings, i, opponent, currOpps);
    if(!success){
        return false;
    }
```

Base Case - Test if Jump is Valid

```cpp
//Cycle through available opponents using recursion
bool lastStop = false;
for(auto& opp : opponents){
    //check all leaves if a leaf is successful this is not the last stop.
    lastStop |= getJumpMoves(moves, move, current, board, kings, dest, opp, currOpps);
}
//If no leafs were successful record this jump as the last stop
if(!lastStop) moves.push_back(move);
```

Cycles Through Available Opponents Calling the Recursion for Each Opportunity

```cpp
//checks if the jump is successful returning a success flag and the index of the landing location in a tuple.
std::tuple<bool, int> MoveManager::checkJump(PlayerCode current, unsigned int* board, unsigned int* kings, int i, int oppPosition, unsigned int oppMap){
    int dest = -1;
    bool success = true;
    Border border = checkBorder(i);
    int row = (i/4)%2;
    int diff = abs(oppPosition-i);

    if(border == LEFT || border == LEFT_BOTTOM){
        dest = (i<oppPosition)?(i+9):(i-7);
    } else if( border == RIGHT || border == RIGHT_TOP){
        dest = (i<oppPosition)?(i+7):(i-9);
    } else {
        if(diff > 4) {
            dest = (i<oppPosition)?(i+9):(i-9);
        } else if(diff < 4) {
            dest = (i<oppPosition)?(i+7):(i-7);
        } else {
            dest = (i<oppPosition)?((row == 0)?(i+7):(i+9)):((row==0)?(i-9):(i-7));
            success &= (!(i%4 == 0 || i%4 == 3)); //TODO: NOt sure if works. prevents the edge cases that are not on the border but the adjacent side preventing attempt to jump over oppo
        }
    }
    //determine if dest is in bounds
    success &= ((dest < 31)&&(dest > 0));
    //determine if dest is open
    success &= (BitUtilities::checkBit(board[current], dest) == 0 && BitUtilities::checkBit(kings[current], dest) == 0) && BitUtilities::checkBit(oppMap, dest)==0;

    //return results
    return std::make_tuple(success, dest);
```

Checks the Location of the Piece Relative to its Jump Location to Determine if it is a Success

```cpp
tuple<bool, std::vector<int>> MoveManager::checkForOpponent(PlayerCode current, bool isKing, unsigned int oppMap, int i) {
    bool success = false;
    std::vector<int> oppPositions;
    int locPoint = -1;
    Border border = checkBorder(i);

    if(border == TOP || border == RIGHT_TOP) {
        if(border == TOP){
            success |= MoveManager::registerOpponent((i + steps[0][0][0]), oppMap, oppPositions);
            success |= MoveManager::registerOpponent((i + steps[0][0][1]), oppMap, oppPositions);
        } else {
            success |= MoveManager::registerOpponent((4+i), oppMap, oppPositions);
        }
    } else if((border == BOTTOM || border == LEFT_BOTTOM)) {
        if(border == BOTTOM){
            success |= MoveManager::registerOpponent((i + steps[1][1][0]), oppMap, oppPositions);
            success |= MoveManager::registerOpponent((i + steps[1][1][1]), oppMap, oppPositions);
        } else {
            success |= MoveManager::registerOpponent((-4+i), oppMap, oppPositions);
        }
    } else if(border == LEFT || border == RIGHT){
        success |= MoveManager::registerOpponent(((current == PLAYERONE)?(4+i):(-4+i)), oppMap, oppPositions);
        if(isKing) success |= MoveManager::registerOpponent(((current == PLAYERTWO)?(4+i):(-4+i)), oppMap, oppPositions);
    } else {
        success |= MoveManager::registerOpponent((i + steps[current][((i/4)%2)][0]), oppMap, oppPositions);
        success |= MoveManager::registerOpponent((i + steps[current][((i/4)%2)][1]), oppMap, oppPositions);
        if(isKing){
            success |= MoveManager::registerOpponent((i + steps[BitUtilities::flipNumber(current)][((i/4)%2)][0]), oppMap, oppPositions);
            success |= MoveManager::registerOpponent((i + steps[BitUtilities::flipNumber(current)][((i/4)%2)][1]), oppMap, oppPositions);
        }
    }

    return make_tuple(success, oppPositions);
}
```

Similar to the Build Move Map, the Location of the Piece Determines if There are Opponents Adjacent and Adds Them to a Vector

- **User Selection -** All available moves and jumps are presented to the user using uppercase letters to represent the piece's destination. The user selects a letter used as a key to determine all the pieces and the captured enemies on that move. The user then selects the desired move they would like to make. This selection is then used to execute the move.

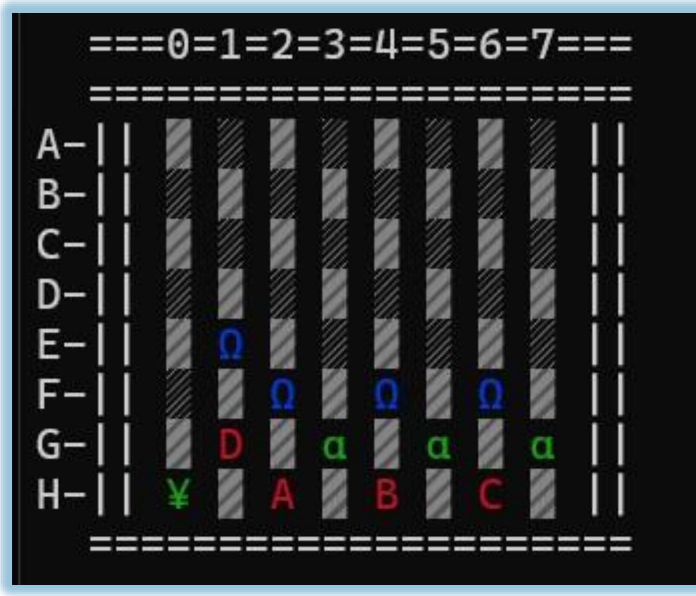
```cpp
//prompt user to select a letter of the board that has an open move. Then select the specific token to move. Returning a tuple containing the coordinates.
std::tuple<char, int> MoveManager::getUserSelection(map<char, vector<Move>>& charToPiece){
    string keyInput;

    cout << "\nEnter the letter of the desired move. _>>> ";
    while(true){
        cin >> keyInput;
        if(!isdigit(keyInput[0]) && charToPiece.find(toupper(keyInput[0])) != charToPiece.end()){
            break;
        }
        cout << "\nNot a valid move. Try again! =>>> ";
    }

    string pieceInput;
    cout << "\nBelow is a list of pieces that can move to your selection.\n";
    int i = 0;
    for(auto& move : charToPiece[toupper(keyInput[0])]){
        cout << "\n" << i << " -- " << getCoor(move.start);
        if(!move.opponent.empty()) {
            cout << " Captured Opponents: ";
            for(auto& opp : move.opponent){
                cout << getCoor(opp) << " ";
            }
        }
        i++;
    }
    cout << "\n-------> ";

    while(true){
        cin >> pieceInput;
        if(!isdigit(pieceInput[0]) && (isdigit(pieceInput[0]) < 0 || isdigit(pieceInput[0]) >= charToPiece[toupper(keyInput[0])].size())){
            cout << "Invalid -------->";
        } else {
            break;
        }
    }
    cin.get();
    return std::make_tuple(toupper(keyInput[0]), (pieceInput[0]-'0'));
}
```

The user is prompted for a letter and then a number. Basic input validation is implemented.



Display of letters to choose from to move

```
Enter the letter of the desired move. _>>> a

Below is a list of pieces that can move to your selection.

0 -- G3
------->
```

Prompting the user to select a specific move

- **Execute Move -** The move is processed after the user has been presented with available moves and chooses a selection. In this process, the pieces' bit index is flipped at the starting and ending points. If there are opponents captured, their bit indexes will be turned off. Additionally, if a piece lands on its base, it will be kinged, as well as if it was kinged during a jump sequence.

```cpp
//flips all the required bit per move based on the Move struct that is passed to the func
void MoveManager::executeMove(PlayerCode current, unsigned int* board, unsigned int* kings, Move move){
    //determines if the player's piece needs to flip the kings board or main board
    if(move.mustKing){
        swapPosition(current, board, move);
        kingPiece(current, board, kings, move.end);
    } else if(move.isKing) swapPosition(current, kings, move);
    else swapPosition(current, board, move);

    //flip all opponents bits that are captured during the move
    if(move.jump){
        for(auto& opp : move.opponent){
            if(BitUtilities::checkBit(board[BitUtilities::flipNumber(current)], opp)) board[BitUtilities::flipNumber(current)] = BitUtilities::flipBit(board[BitUtilitie
            else kingPiece(static_cast<PlayerCode>(BitUtilities::flipNumber(current)), board, kings, opp);//kings[BitUtilities::flipNumber(current)] = BitUtilities::fli
        }
    }

    //king the player's piece if necessary
    if(!move.isKing){
        Border border = checkBorder(move.end);
        if(checkBase(border, current)) kingPiece(current, board, kings, move.end);
    }
}

//checks if piece is in base
```

All that hard work is finally executed.

View Management

The View class is static and primarily responsible for all console displays. It displays the board, scores, player turn, labels, headers, etc. Additionally, it implements color coding and renders all board states.

Key Components:

- **Display Board —** Renders the board based on the given parameters. It will register the location of both players' standard and king pieces, potential moves based on each player's turn, the score of the game, and the board layout.

```cpp
//print the board
void View::displayBoard(unsigned int* board, unsigned int* kings, unsigned int moveBoard, map<int, char>& indexToChar, int score[]) {
    int row = 0;
    char rowMarker = 'A';
    string playerLabel = "              Player One      Player Two";
    string scoreStr = "                " + std::to_string(score[0]) + "               " + std::to_string(score[1]);
    cout << "  ===0=1=2=3=4=5=6=7===" << endl;
    cout << "  ====================";
    for(int i = 0 ; i < 32 ; i++){

        if(i%4 == 0){
            (i==0)?cout << "\nA-|| ": cout << "||"<<((i==8)?playerLabel:"") << ((i==16)?scoreStr:"") << "\n" << rowMarker <<"-|| "; //every 4 steps start a new line and print borders
            rowMarker++;
            row++; //increment the row index every 4 steps
        }

        if(BitUtilities::checkBit(board[0], i)) (row%2==1)?cout << boardParts[1] << " " << getColorCode(GREEN) << pieces[0] << getColorCode(DEFAULT) << " " : cout << getColorCode(GREE
        else if(BitUtilities::checkBit(kings[0],i)) (row%2==1)?cout << boardParts[1] << " " << getColorCode(GREEN) << kingPieces[0] << getColorCode(DEFAULT) << " " : cout << getColor
        else if(BitUtilities::checkBit(board[1], i)) (row%2==1)?cout << boardParts[1] << " " << getColorCode(BLUE) << pieces[1] << getColorCode(DEFAULT) << " " : cout << getColorCode(
        else if(BitUtilities::checkBit(kings[1],i)) (row%2==1)?cout << boardParts[1] << " " << getColorCode(BLUE) << kingPieces[1] << getColorCode(DEFAULT) << " " : cout << getColorCo
        else if(!indexToChar.empty() && BitUtilities::checkBit(moveBoard, i)) (row%2==1)?cout << boardParts[1] << " " << getColorCode(RED) << indexToChar[i] << getColorCode(DEFAULT) <
        else (row%2==1)?cout << boardParts[1] << " " << boardParts[0] << " ": cout << boardParts[0] << " " << boardParts[1] << " ";
    }
    cout << "||\n  ====================" << endl;
}
```

This function is leveraged by other functions dependent on its use case.

- **Display Current Player**—This displays to the users whose turn it is. It also displays the characters and colors of that player's pieces.

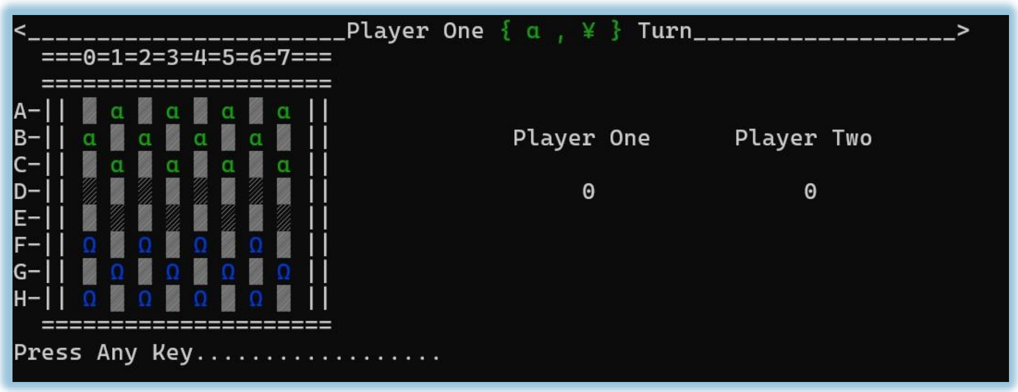
```
//display current player
void View::displayCurrentPlayer(PlayerCode current){
    string player = (current == PLAYERONE)?"One":"Two";
    string playPiece = (current == PLAYERONE)?"{ " + std::string(1,pieces[0]) + " , " + std::string(1, kingPieces[0]) + " }":"{ " + std::string(1,pieces[1]) + " , " + std::string(1, ki
    Color currColor = (current == PLAYERONE)?GREEN:BLUE;
    cout << "<_______________________Player " << player << " " << getColorCode(currColor) << playPiece << getColorCode(DEFAULT) << " Turn___________________>\n";
}
```

Players Turn Code

Presentation to user

- **Display Current State and Moves -** The different functions display the board by accessing the displayBoard function. One shows the game's current state, and the other shows the game's current state, including the player's available moves.

Including the moves



Current Game State

- **Display Winner -** Receives the given strings from the game controller and displays those parameters to the user.

```
void View::displayWinner(string player, string message){
    cout << "****************** Player " << player << " Wins **********************\n";
    cout << message;
}
```

Winner

- **Color Coding -** A switch statement combined with an Enum of color names with their indexes assigned to the index of a string array that contains the ANSI escape codes for a given character. So, this function is called using the Enum color name, and the ANSI code is returned.

```
//returns the escape sequence for each color based on 
string View::getColorCode(Color color){
    switch(color){
        case GREEN: return colorSequence[GREEN];
            break;
        case BLUE: return colorSequence[BLUE];
            break;
        case RED: return colorSequence[RED];
            break;
        case DEFAULT: return colorSequence[DEFAULT];
        default:
            return colorSequence[DEFAULT];
    }
}
```

Switch Selector

```
const string View::colorSequence[4] = {"\033[32m", "\033[34m", "\033[31m", "\033[39m"}; //contains the const ansi escape codes for colors
```

Codes

- **Pause and Clear -** These two functions serve a general purpose. One pauses the screen, waiting for a button to be pressed, and the other clears the screen.

```cpp
//pause the screen and wait for user to press a key
void View::pause(){
    cout << "Press Any Key.................";
    cin.get();
}

void View::clear(){
    cout << "\033[2J\033[H";
}
```

Player Object

The player class contains all the essential data structures related to each player. As well as the index code for each player.

Key Components:

- **Main and King boards -** These unsigned integers are used to represent the location of the player's pieces, allocated to the instance of each player. They are initialized to 0 until the game modifies them at another point. These integers are referenced to an array in the game controller that allows them to be modified throughout the game process.
- **Player Code -** This is an Enum value that represents the index for the player throughout all game-related arrays.
- **Constructor -** Builds the object.

```cpp
class Player {
private:
    unsigned int main; //main player board for regular pieces
    unsigned int king; //king player board for all kinged pieces
    PlayerCode playerCode; //playerCode is used for index selection

public:
    Player(PlayerCode code) : main(0), king(0), playerCode(code) {}

    //get the reference of the main player board
    unsigned int& getMain() { return main; }
    //return the reference of the king board
    unsigned int& getKing() { return king; }
    //return playerCode
    PlayerCode getCode() { return playerCode; }
};
```

Player Class

Shared Types

This header defines all shared data structures and Enums used throughout the game, including player codes, board boundaries, colors, and moves.

Key Components:

- **Total Pieces -** Stores the number of total pieces a player should start with.
- **Player Code -** Represents the index for each player when calling array values.
- **Board Boundaries -** Stores the state that an index is based on its location in the board
- **Color -** Stores the state of color
- **Move Struct -** Stores all the required information to move a piece. Such as start, destination, if it is a jump, whether the piece is a king, whether the piece should be kinged, and all defeated opponents.

```cpp
//This enum is used to control boundary state
enum Border {
    TOP,
    BOTTOM,
    LEFT,
    LEFT_BOTTOM,
    RIGHT,
    RIGHT_TOP,
    OPEN
};

//These enums values will represent the color code
enum Color {
    GREEN,
    BLUE,
    RED,
    DEFAULT
};

//this struct will store the coordinates of a move
struct Move {
    int start; //starting position of the move
    int end; //ending location of the move
    bool jump; //signifies that is a jump move over an opponent
    bool isKing; //signifies that the players piece is a king
    bool mustKing; //signifies if the piece got kinged during a jump sequence
    std::vector<int> opponent; //stores the index of all opponenets that are captured during the move

    //constructor for Move struct
    Move(int start, int end, bool jump, bool king, std::vector<int> opponent) : start(start), end(end), jump(jump), isKing(king), mustKing(false), opponent(opponent) {}
};

#endif
```

Displays Some of those Shared Properties

Bit Utilities

The BitUtilities class contains all the required helper functions that perform bit manipulation. This is essential for creating an efficient board state for a checkers game. The abilities of these functions range from checking to counting to flipping bits.

Key Components:

- **Flip Number—**This function accepts an int and returns an int. Its function is to flip the least significant bit. It is primarily used for index flipping on arrays between players.

```
//return flipped bit value
int BitUtilities::flipNumber(int value){
    return value^1;
}
```

LSB Flip

- **Flip Bit** – This function accepts an unsigned int and an int, then returns an unsigned int. Its purpose is to flip the bit at the given index and return. Primarily used for moving and capturing pieces.

```
//return the unsigned int after flipping the bit at the given location
unsigned int BitUtilities::flipBit(unsigned int value, int position){
    return value ^ (1 << position);
}
```

Given index flip

- **Set Bit** – This function accepts an unsigned int and an int, then returns an unsigned int. It aims to set the bit at the given index to 1 and return. Primarily used for turning bits on.

```
//set bit at given location to 1
unsigned int BitUtilities::setBit(unsigned int value, int position){
    return value | (1 << position);
}
```

Only turns bit on

- **Check Bit** - This function accepts an unsigned int and an int, then returns an int. Its purpose is to check if the bit is equal to 1. Returning 1 if the bit is 1 and 0 if the bit is 0. Primarily used for checking if a piece is at that location.

```
//check value of bit return 1 or 0
int BitUtilities::checkBit(unsigned int num, int position){
    return (num & (1 << position));
}
```

Am I a 1

- **Count Bits** – This function receives an unsigned int and returns an integer based on how many bits are on. Primarily used for determining the score.

```cpp
//counts the number of 1 bits in the given number
int BitUtilities::countBits(int player, unsigned int* board){
    int count = 0;
    for(int i = 0 ; i < 32 ; i++){
        count += (BitUtilities::checkBit(board[player], i))?1:0;
    }
    return count;
}
```

Cycles through each bit using checkBit

- **Merge Bits** – This function receives two unsigned integers and returns an unsigned int. It will create a union based on all bits turned on in each integer. Used to merge the opponents' king and standard piece locations when determining jump moves.

```cpp
//merge the 1's in two unsigned integers and return a new integer.
unsigned int BitUtilities::mergeBits(unsigned int a, unsigned int b){
    unsigned int merged = 0;
    for(int i = 0 ; i < 32 ; i++){
        if(BitUtilities::checkBit(a, i) || BitUtilities::checkBit(b, i)) merged = BitUtilities::flipBit(merged, i);
    }
    return merged;
}
```

Merger

Testing

Many functions were implemented to test the functionality of each feature before proceeding. Additionally, log statements were inserted at critical locations to determine how the system was reacting at each moment without using the debugger. The VS Code debugger was utilized for more complex solutions to determine how the system performed.

```cpp
//Test if a parameters are functioning for the moveMapParameters
void testMoveMapParts(unsigned int movesBoard, std::map<int, char>& indexToChar, std::map<char, vector<Move>>& charToPiece){
    cout << movesBoard << endl;

    for(const auto& pair : indexToChar){
        cout << "Key: " << pair.first << ", Value: " << pair.second << endl;
    }

    for(const auto& pair : charToPiece){
        cout << "Key: " << pair.first << endl;
        cout << "Moves: " << endl;

        for(const auto& move : pair.second){
            cout << " Start: " << move.start
                << " End: " << move.end
                << " Jump: " << (move.jump?"Yes":"No")
                << " Opponent: " << " Add Opponents" << endl;
        }
    }
}
```

Testing the Collections for Move Controller

```
//test selection function
void testSelection(std::tuple<char, int> selection){
    char row;
    int col;
    std::tie(row, col) = selection;
    cout << "Selected Coordinate: " << row << " " << col;
}
```

Testing user selection

Future Improvements

- I would like to implement a GUI interface
- AI compatibility for Player
- Implement it so Different Chatgpt and other AI systems can compete against each other
- Determine that all edge cases are covered in the code
- Slim down the decision structures
- Refactor until it is perfect

Citations

There was not a plethora of information on how to do this project. I did it solo, utilizing ChatGPT when I had questions primarily on syntax since I had not coded in C++ in a while.