Data Structures Project

Ritoban Roy-Chowdhury

October 30, 2020

Introduction

The Royal Game of Ur is the world's oldest board game discovered in Ancient Mesopotamia by Dr. Irving Finkel. It's a simple race game, similar to Ludo or Backgammon. I chose this because while the rules of the game are relatively simple, there are several nuances, especially in the displaying the game to the terminal, and certain aspects, like the path that the pieces follow, are particularly well modelled by certain data structures.

The development took about one week. Altogether, the code is 970 lines, comprised of 322 lines of C header files and 648 lines of C++ code. These make up 8 separate classes.

The code can be found on Github https://github.com/ritobanrc/Project1.

Approach to Development

I approached the development of this project modularly. First, I developed several utility classes, like `Display` and `SideData`. Then, I went through each game feature. I independently developed the board, and then the dice, and I slowly added functionality one step at a time, testing along the way. First I added the pieces and piece display code, and then I allowed the pieces to move, and then be captured, and then win logic, etc.

Throughout this, I tried to commit the code to Github often, ideally after each feature was developed, so I could easily roll back if something broke, or make a branch to work on a different feature for some time.

Game Rules

The game is a race game, with rules similar to Ludo or Parcheesi. Each player has 7 pieces, which they must get through the entire board. Each turn, the player rolls the 4 tetrahedral dice, each with two marks at two corners.
You move forward based on the number of marks. The first player to get all of their pieces off the board wins. However, if you land on one of your opponent's pieces, you "capture" that piece and it gets sent back. Additionally, the star squares let you roll a second time when you land on them, and the central star square is a safe space, where you can't be captured.

Description of Code

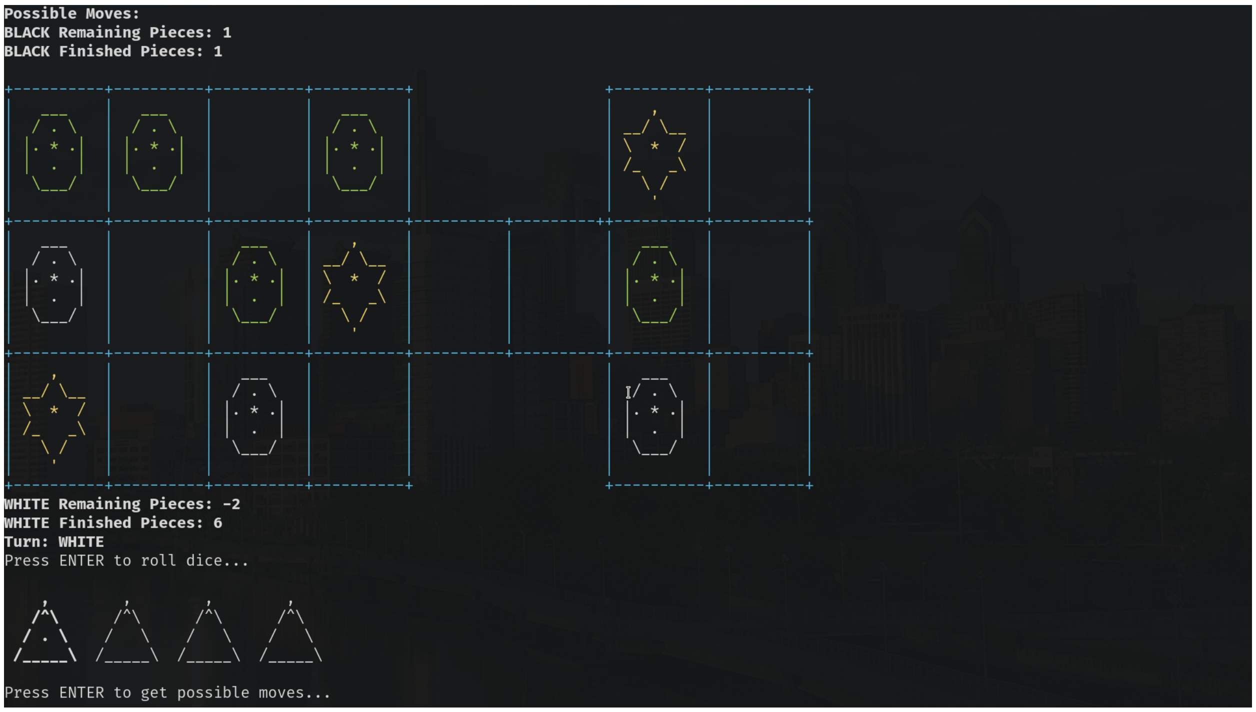
The code is organized into several classes, each with distinct roles.

- Game
  - The majority of the game logic. Anything to do with moves is here, as well as things about the overall game state, like whose turn it is. It also contains the Board and the DiceRoller.
- Board

    - Contains a 2D array representing the board, as well as methods to show the board and associated data to help with showing the board.
- DiceRoller
    - Contains a bitset to represent the 4 dice, as well as methods to actually roll the dice and show the dice.
- Piece
    - Represents a piece on the board. The path taken by the piece is modelled by an iterator over a linked list.
- Square
    - Represents a square, as well as what Piece is on it. The most important function is `GetDisplayCharAt`, which is what the Board uses to figure out how to display the square.

- Display
    - A static class that contains several methods to help with displaying things with formatting
- Color
    - Provides a nice interface for dealing with ANSI color values.
- SideData
    - Represents a piece of data that exists for both sides (white and black), as well as a function to get a reference to one of them.

Sample



```
Possible Moves:
BLACK Remaining Pieces: 1
BLACK Finished Pieces: 1
```

```
WHITE Remaining Pieces: -2
WHITE Finished Pieces: 6
Turn: WHITE
Press ENTER to roll dice...
```

```
Press ENTER to get possible moves...
```

Figure 1: Screenshot of Gameplay

Checkoff Sheet

1. Container Classes
    - Sequences

        - `std::list<Square*>` is used to represent the path that is taken by the pieces through the Board.
        - `std::bitset<4>` is used to represent the 4 dice in the DiceRoller.
    - Associative Containers
        - `std::set<Piece*>` is used for the set of all the pieces currently on the board.
        - `std::map<std::string, Color>` is used as a map from strings to colors, so we can use the string “Red” to easily get the corresponding ANSI color value.
    - Container Adapters
        - `std::deque<Move>` is used to hold all the possible moves. Moves are added in to the queue, and then they can be popped off and dealt with one at a time.
2. Iterators
    - `std::list<Square*>::const_iterator` is used to represent the location of pieces. The piece location is a pointer into the path list, and can be incremented to move forward.
    - Game.cpp:107 – A random access iterator into the `deque` is used to go over all of the moves.
    - Game.cpp:165 – A bidirectional iterator into the `std::set<Piece>` is used to go over all of the pieces
    - Board.cpp:96 – A constant Random Access Iterator over the rows of the array is used.
    - Board.cpp:97 – And another constant Random Access Iterator over the columns of the 2D array.
3. Algorithms
    - Game.cpp:107 – `std::for_each` is used to iterate over the possible moves
    - DiceRoller.h:28 – `std::bitset<4>::count` is used to count the number of bits in the bitset.
    - Game.cpp:256 – `std::set::erase` is used to remove from the set.
    - Game.cpp:105 – `std::sort` is used to sort the moves by column, so they can be easily selected by the user.

Documentation

The majority of the game logic is in `Game::PlayGame`.

Pseudocode:

```
• Game::PlayGame
    – Display Basic information
    – Display Board
    – If turn is white
        ∗ Print “WHITE”
    – Else
        ∗ Print “BLACK”
    – Wait for ENTER to roll Dice
    – Roll Dice
    – If dice equals 0
        ∗ Print “Better luck next time”
        ∗ Call NextTurn to switch turn
        ∗ Return to beginning
    – If not against computer, or turn is white (not computer)
        ∗ Wait for ENTER to get possible moves
    – Get Possible Moves
    – If there are no possible moves
        ∗ Print “No possible moves”
        ∗ Call NextTurn to switch turn
        ∗ Return to beginning
    – Sort moves according to X position
    – Assign each move a number
    – If not against computer, or turn is white (not computer)
```

            * Display possible moves
            * Ask user for move selection
        – Else
            * Select Move Randomly
        – Apply Move
        – Clear the Possible Moves
        – If move landed on star
            * Print "You landed on a star."
            * Return to beginning *without* calling NextTurn.
        – Check for Win
        – Call Next Turn
        – Return to Beginning

Code

main.cpp

```
/*
 * File:   main.cpp
 * Author: ritoban
 *
 * Created on October 25, 2020, 6:55 PM
 */

#include "Display.h"
#include "Color.h"
#include "Board.h"
#include "Game.h"
#include "DiceRoller.h"

#include <iostream>
#include <map>
#include <string>
#include <cstdlib>
#include <ctime>


using namespace std;

int main(int argc, char** argv) {
    srand(0);

    Display::BeginColor(COLOR["Red"].AsFG());
    Display::BeginBold();
    cout << "Welcome to the Royal Game of Ur!";
    Display::EndFormat();
    Display::NewLine();

    cout << R"(
               The Royal Game of Ur is the world's oldest board game
               discovered in Ancient Mesopotamia by Dr. Irving Finkel.

               The game is a race game, with rules similar to Ludo or
               Parcheesi. Each player has 7 pieces, which they must get
```

```
            through the entire board. Each turn, the player rolls the
            4 tetrahedral dice, each with two marks at two corners.
            You move forward based on the number of marks. The first
            player to get all of their pieces off the board wins.
            However, if you land on one of your opponent's pieces, you
            "capture" that piece and it gets sent back.
            Additionally, the star squares let you roll a second time when
            you land on them, and the central star square is a safe space,
            where you can't be captured.

            Finally, for visibility, this game uses Green to show white pieces.

            HAVE FUN PLAYING!

            For more information, see this video https://www.youtube.com/watch?v=WZskjLq040I.

            To play against the computer, enter 1.
            To play against another human, enter 2.
            )" << std::endl;

    int sel;
    cin >> sel;
    std::cin.ignore();

    bool againstComputer = sel == 1;


    Game game;
    game.PlayGame(againstComputer);

    return 0;
}
```

Game.h

```
/*
 * File:   Game.h
 * Author: ritoban
 *
 * Created on October 25, 2020, 6:56 PM
 */

#ifndef GAME_H
#define GAME_H

#include <set>
#include <list>
#include <deque>
#include "Piece.h"
#include "Board.h"
#include "DiceRoller.h"
#include "SideData.h"
```

```
class Game {
public:
    Game();
    virtual ~Game();

    void PlayGame(bool);
    bool AddPiece(Side);

    struct Move {
        Piece* piece; // if nullptr, that means the piece must be created
        Square* target;
    };

    std::deque<Move> GetPossibleMoves();
    void ApplyMove(Move);
    void NextTurn();

    Board board;
    DiceRoller diceRoller;

    // This is public because there's a getter inside SideData
    SideData<std::set<Piece*>> pieces;
    SideData<int> remainingPieces;
    SideData<int> completedPieces;
    SideData<std::list<Square*>> path;

private:
    Side turn = white;
};

#endif /* GAME_H */
```

Game.cpp

```
/*
 * File:   Game.cpp
 * Author: ritoban
 *
 * Created on October 25, 2020, 6:56 PM
 */

#include "Game.h"
#include "Piece.h"
#include "Display.h"
#include "SideData.h"
#include <set>
#include <list>
#include <deque>
#include <algorithm>
#include <utility>
#include <thread>
#include <cstdlib>
```

```
Game::Game() : pieces(std::set<Piece*>(), std::set<Piece*>()), remainingPieces(7, 7), completedPieces(0
    std::list<Square*>& whitePath = path.Get(white);
    std::list<Square*>& blackPath = path.Get(black);

    for (int i = 3; i >= 0; i--) {
        whitePath.push_back(board.GetSquare(i, 2));
        blackPath.push_back(board.GetSquare(i, 0));
    }

    for (int i = 0; i < 8; i++) {
        whitePath.push_back(board.GetSquare(i, 1));
        blackPath.push_back(board.GetSquare(i, 1));
    }

    for (int i = 7; i >= 6; i--) {
        whitePath.push_back(board.GetSquare(i, 2));
        blackPath.push_back(board.GetSquare(i, 0));
    }
}

std::string side_str(Side s) {
    if (s == white) {
        return "WHITE";
    } else {
        return "BLACK";
    }
}


Game::~Game() {
}

void Game::PlayGame(bool againstComputer) {
    while (true) {
        Display::BeginBold();
        std::cout << "BLACK Remaining Pieces: " << remainingPieces.Get(black) << std::endl;
        std::cout << "BLACK Finished Pieces: " << completedPieces.Get(black) << std::endl;
        Display::EndFormat();
        board.ShowBoard();
        Display::BeginBold();
        std::cout << "WHITE Remaining Pieces: " << remainingPieces.Get(white) << std::endl;
        std::cout << "WHITE Finished Pieces: " << completedPieces.Get(white) << std::endl;
        Display::EndFormat();

        Display::PrintBold("Turn: ");
        if (turn == white) {
            Display::BeginColor(COLOR["Green"].AsFG());
            Display::PrintBold("WHITE");
        } else {
            Display::BeginColor(COLOR["Black"].AsFG());
            Display::PrintBold("BLACK");
        }
        Display::EndFormat();
```

```
Display::NewLine();

Display::Print("Press ENTER to roll dice...");
std::cin.ignore();

diceRoller.RollDice();
diceRoller.ShowDiceRoller();

if (diceRoller.CountDice() == 0) {
    Display::BeginColor(COLOR.at("Red").AsFG());
    std::cout << "ZERO! Better luck next time!";
    Display::NewLine();
    Display::EndFormat();
    NextTurn();
    continue;
}

if (!againstComputer || turn == white) {
    Display::Print("Press ENTER to get possible moves...");
    std::cin.ignore();
}

auto moves = GetPossibleMoves();

if (moves.empty()) {
    Display::BeginColor(COLOR.at("Red").AsFG());
    Display::Print("No possible moves!");

    NextTurn();
    continue;
}

std::sort(moves.begin(), moves.end(), [](Move a, Move b){
        return a.target->xPos > b.target->xPos;
});


int moveNumber = 1;
std::for_each(moves.begin(), moves.end(),
    [&](Game::Move m){
        if (m.target != nullptr) {
            m.target->moveNumber = std::make_pair(moveNumber, true);
        } else {
            board.endMoveNumber.Get(turn) = moveNumber;
        }
        moveNumber++;
});

Display::PrintBold("Possible Moves: \n");

int moveSelected;
if (!againstComputer || turn == white) {
    board.ShowBoard();
```

```
            std::cout << "Please select a move between 1 and " << moves.size();
            Display::NewLine();

            std::cin >> moveSelected;
            std::cin.ignore();
        } else {
            moveSelected = rand() % moves.size() + 1;
        }
        Move move = moves.at(moveSelected - 1);
        this->ApplyMove(move);
        board.ClearPossibleMoves();

        if (move.target != nullptr && move.target->GetStar()) {
            Display::BeginColor(COLOR["Magenta"].AsFG());
            Display::PrintBold("You landed on a Star! You get another turn!");
            Display::NewLine();
            Display::EndFormat();

            continue;
        }

        if (completedPieces.Get(turn) == 7) {
            Display::BeginBold();
            std::cout << side_str(turn) << " WINS!!!!!!" << std::endl;
            return;
        }


        NextTurn();
    }

}

std::deque<Game::Move> Game::GetPossibleMoves() {
    int rolled = diceRoller.CountDice();
    auto moves = std::deque<Game::Move>();

    if (rolled == 0) {
        return moves; // welp you miss a turn!
    }

    if (remainingPieces.Get(turn) > 0) {
        auto thisSidePath = this->path.Get(turn).begin();
        // we can move up to rolled number of times
        // start at 1 because we're adding a new piece to the board
        for (int i = 1; i < rolled; i++) {
          thisSidePath++;
        }
        if ((*thisSidePath)->piece == nullptr) {
          // it's empty!
          auto m = Game::Move();
          m.piece = nullptr;
          m.target = *thisSidePath;
          moves.push_back(m);
        }
    }


    std::set<Piece*>& thisSidePieces = pieces.Get(turn);

    for (auto piece = thisSidePieces.begin(); piece != thisSidePieces.end(); piece++) {
        auto pos = (*piece)->GetPosition();
        // advance the iterator by however much we rolled
        int n = rolled;
        auto end = this->path.Get((*piece)->side).cend();
        while (n > 0 && pos != end) {
            n--;
            pos++;
        }

        if (pos == end) {
            if (n == 0) {
                // Yay! The piece made it to the end
                auto m = Game::Move();
                m.piece = *piece;
                m.target = nullptr;
                moves.push_back(m);
            }
            // We rolled something past the end, this is not a valid move
            continue;
        }

        Square* s = *pos;
        if (s->piece == nullptr || (s->piece->side != (*piece)->side && s->GetStar() == false)) {
            // either we're at the end, we're moving to an empty square, or we can capture a piece.
            auto m = Game::Move();
            m.piece = *piece;
            m.target = *pos;
            moves.push_back(m);
        }
    }

    return moves;
}

/// Tries to add a piece to the board. Returns false if the start square is already occupied.
bool Game::AddPiece(Side s) {
    Piece* piece;

    if (board.startSquare.Get(s)->piece == nullptr) {
        piece = new Piece(s, path.Get(s).cbegin());
        pieces.Get(s).insert(piece);
        remainingPieces.Get(s)--;
        board.startSquare.Get(s)->piece = piece;
    } else {
        return false;
    }
    return true;
```

```
}

void Game::ApplyMove(Game::Move m) {
    if (m.piece == nullptr) {
        std::list<Square*>::iterator pos = path.Get(turn).begin();
        while (*(pos) != m.target) {
            pos++;
        }

        Piece* piece = new Piece(turn, pos);
        (*pos)->piece = piece;
        pieces.Get(turn).insert(piece);
        remainingPieces.Get(turn)--;

    } else {
        Square* currentPos = *(m.piece->GetPosition());
        currentPos->piece = nullptr;

        if (m.target == nullptr) {
            Display::BeginColor(COLOR["Magenta"].AsFG());
            Display::Print("Success! ");
            Display::Print(side_str(turn));
            Display::Print(" completes 1 piece! \n");
            Display::EndFormat();

            this->pieces.Get(turn).erase(m.piece);

            completedPieces.Get(turn)++;
            return;
        }

        m.piece->AdvanceUntil(m.target);
        if (m.target->piece != nullptr) {
            // this piece just got captured!
            Side capturedSide = m.target->piece->side;
            this->pieces.Get(capturedSide).erase(m.target->piece);
            this->remainingPieces.Get(capturedSide) += 1;
        }
        m.target->piece = m.piece;
    }
}

void Game::NextTurn() {
    if (turn == white) {
        turn = black;
    } else {
        turn = white;
    }
}
```

Board.h

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

/*
 * File:   Board.h
 * Author: ritoban
 *
 * Created on October 25, 2020, 6:56 PM
 */

#ifndef BOARD_H
#define BOARD_H

#include <array>
#include "Square.h"
#include "SideData.h"

constexpr int BOARD_WIDTH = 8;
constexpr int BOARD_HEIGHT = 3;

class Board {
public:
    Board();
    virtual ~Board();

    void ShowBoard();
    Square* GetSquare(int x, int y) { return grid[y][x]; }
    void ClearPossibleMoves();

    SideData<Square*> startSquare;
    SideData<int> endMoveNumber = SideData<int>(-1, -1);
private:
    std::array<std::array<Square*, BOARD_WIDTH>, BOARD_HEIGHT> grid;

};

#endif /* BOARD_H */
```

Board.cpp

```
/*
 * File:   Board.cpp
 * Author: ritoban
 *
 * Created on October 25, 2020, 6:56 PM
 */

#include <iostream>
#include <algorithm>

#include "Board.h"
#include "Color.h"
```

```
#include "Display.h"
#include "Square.h"

Board::Board() : startSquare(nullptr, nullptr) {
    for (int row = 0; row < BOARD_HEIGHT; row++) {
        for (int col = 0; col < BOARD_WIDTH; col++) {
            if (row != 1 && (col == 4 || col == 5)) {
                this->grid[row][col] = nullptr;
                continue;
            }
            this->grid[row][col] = new Square(col);
        }
    }

    grid[0][0]->SetStar(true);
    grid[2][0]->SetStar(true);
    grid[1][3]->SetStar(true);
    grid[0][6]->SetStar(true);
    grid[2][6]->SetStar(true);

    startSquare.Get(white) = grid[2][3];
    startSquare.Get(black) = grid[0][3];
}

Board::~Board() {
}

void Board::ShowBoard() {
    Display::BeginColor(COLOR["Blue"].AsFG());
    Display::NewLine();

    for (int row = 0; row < BOARD_HEIGHT; row++) {
        int start = row == 0 ? 0 : 1;
        for (int y = start; y < CELL_HEIGHT; y++) {
            for (int col = 0; col < BOARD_WIDTH; col++) {
                for (int x = 0; x < CELL_WIDTH; x++) {
                    char c;
                    if (grid[row][col] == nullptr) {
                        if (x == CELL_WIDTH - 1 || (x == 0 && grid[row][col - 1] == nullptr)) {
                            continue;
                        }
                        if (y == CELL_HEIGHT - 1 && row < BOARD_HEIGHT - 1 && grid[row + 1][col] != null
                            c = grid[row + 1][col]->GetDisplayCharAt(x + 1, 0);
                        } else {
                            // end space
                            if (col == 5 && x == 6 && y == 3) {
                                if (endMoveNumber.Get(white) > 0 && row == 2) {
                                    c = endMoveNumber.Get(white) + '0';
                                } else if (endMoveNumber.Get(black) > 0 && row == 0) {
                                    c = endMoveNumber.Get(black) + '0';
                                } else {
                                    c = ' ';
                                }
                            } else {
                                c = ' ';
                            }
                        }
                    }
                    else {
                        if (x == 0) {
                            if (col == 0 || grid[row][col - 1] == nullptr) {
                                c = grid[row][col]->GetDisplayCharAt(x, y);
                            } else {
                                continue;
                            }
                        } else {
                            c = grid[row][col]->GetDisplayCharAt(x, y);
                        }
                    }

                    std::cout << c;
                    Display::BeginColor(COLOR["Blue"].AsFG());
                }
            }
            Display::NewLine();
        }
    }
    Display::EndFormat();
}

void Board::ClearPossibleMoves() {
    this->endMoveNumber.Get(white) = -1;
    this->endMoveNumber.Get(black) = -1;
    for (auto row = grid.cbegin(); row != grid.cend(); row++) {
        for (auto square = row->cbegin(); square != row->cend(); square++) {
            Square* s = *square;
            if (s != nullptr && s->moveNumber.second) {
                s->moveNumber.second = false;
            }
        }
    }
}
```

Piece.h

```
/*
 * File:   Piece.h
 * Author: ritoban
 *
 * Created on October 26, 2020, 4:07 PM
 */

#ifndef PIECE_H
#define PIECE_H

#include <list>
#include "Square.h"
#include "SideData.h"
```

```
class Piece {
public:
    Piece(Side, std::list<Square*>::const_iterator);
    Piece(const Piece& orig);
    virtual ~Piece();

    std::list<Square*>::const_iterator GetPosition() { return path; }
    void AdvanceUntil(Square*);

    Side side;
private:
    // This is a pointer into the path of where the piece is right now.
    std::list<Square*>::const_iterator path;


};

#endif /* PIECE_H */
```

Piece.cpp

```
/*
 * File:   Piece.cpp
 * Author: ritoban
 *
 * Created on October 26, 2020, 4:07 PM
 */

#include "Piece.h"
#include <list>

Piece::Piece(Side s, std::list<Square*>::const_iterator path) : side(s), path(path) {
}

Piece::Piece(const Piece& orig) {
}

Piece::~Piece() {
}

void Piece::AdvanceUntil(Square* s) {
    while(*path != s) {
        path++;
    }
}
```

Square.h

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

/*
 * File:   Square.h
 * Author: ritoban
 *
 * Created on October 26, 2020, 8:37 AM
 */

#ifndef SQUARE_H
#define SQUARE_H

#include <utility>

class Piece;

class Square {
public:
    Square(int);
    Square(const Square& orig);
    virtual ~Square();

    char GetDisplayCharAt(int, int);
    void SetStar(bool val) { isStarSquare = val; }
    bool GetStar() { return isStarSquare; }

    Piece* piece;
    // The int describes the move number to display, the bool is true if it should be shown.
    std::pair<int, bool> moveNumber;
    int xPos; // for sorting
private:
    bool isStarSquare;

};


const int CELL_WIDTH = 12;
const int CELL_HEIGHT = 8;

#endif /* SQUARE_H */
```

Square.cpp

```
/*
 * File:   Square.cpp
 * Author: ritoban
 *
 * Created on October 26, 2020, 8:37 AM
 */

#include "SideData.h"
#include "Square.h"
#include "Display.h"
```

```
#include "Piece.h"
#include <string>
#include <vector>
#include <utility>

Square::Square(int x) : xPos(x) {
    this->isStarSquare = false;
    this->piece = nullptr;
    this->moveNumber = std::make_pair(0, false);
}

Square::Square(const Square& orig) {
}

Square::~Square() {
}

char Square::GetDisplayCharAt(int i, int j) {
    bool horizontalEdge = i == 0 || i == CELL_WIDTH - 1;
    bool verticalEdge = j == 0 || j == CELL_HEIGHT - 1;

    const std::vector<std::string> star ={
R"(     ,    )",
R"(  __/ \__ )",
R"(  \  *  / )",
R"(  /_   _\ )",
R"(    \ /   )",
R"(     '    )" };

    const std::vector<std::string> pieceGFX ={
R"(    ___     )",
R"(   / . \    )",
R"(  |. * .|   )",
R"(  |  .  |   )",
R"(   \___/    )",
R"(            )" };


    if (horizontalEdge && verticalEdge) {
        return '+';
    } else if (horizontalEdge) {
        return '|';
    } else if (verticalEdge) {
        return '-';
    } else {
        if (this->moveNumber.second && i == 6 && j == 3) {
            return this->moveNumber.first + '0';
        }
        if (this->piece != nullptr && (j - 1) < pieceGFX.size() && i < pieceGFX[0].size()) {
            if (this->piece->side == white) {
                Display::BeginColor(COLOR["Green"].AsFG());
            } else {
                Display::BeginColor(COLOR["Black"].AsFG());
            }
            return pieceGFX[j - 1][i];
        }
        else if (isStarSquare && (j - 1) < star.size() && i < star[0].size()) {
            Display::BeginColor(COLOR["Yellow"].AsFG());
            return star[j - 1][i];
        } else {
            return ' ';
        }
    }
}
```

DiceRoller.h

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

/*
 * File:   DiceRoller.h
 * Author: ritoban
 *
 * Created on October 26, 2020, 12:20 PM
 */

#ifndef DICEROLLER_H
#define DICEROLLER_H

#include <set>
#include <bitset>

class DiceRoller {
public:
    DiceRoller();
    DiceRoller(const DiceRoller& orig);
    virtual ~DiceRoller();

    void RollDice();
    void ShowDiceRoller();
    int CountDice() { return dice.count(); };
private:

    std::bitset<4> dice;
};

#endif /* DICEROLLER_H */
```

DiceRoller.cpp

```
/*
 * File:   DiceRoller.cpp
 * Author: ritoban
 *
 * Created on October 26, 2020, 12:20 PM
 */

#include "DiceRoller.h"
#include "Display.h"
#include <set>
#include <cstdlib>
#include <iostream>
#include <vector>

DiceRoller::DiceRoller() {
    dice.reset();
}

DiceRoller::DiceRoller(const DiceRoller& orig) {
}

DiceRoller::~DiceRoller() {
}

void DiceRoller::RollDice() {
    dice.reset();
    for (int i = 0; i < dice.size(); i++) {
        dice.set(i, rand() % 2);
    }
}

void DiceRoller::ShowDiceRoller() {
    const std::vector<std::string> diceArt ={
R"(    ,    )",
R"(   /^\   )",
R"(  /   \  )",
R"( /_____\ )" };

    std::vector<std::string> diceWithPip = diceArt;
    diceWithPip[2][4] = '.';

    Display::NewLine();
    for (int y = 0; y < diceArt.size(); y++) {
        for (int d = 0; d < dice.size(); d++) {
            if (this->dice[d]) {
                Display::PrintBold(diceWithPip.at(y));
            } else {
                std::cout << diceArt.at(y);
                //Display::PrintBold(diceArt.at(y));
            }
        }
        std::cout << std::endl;
    }

    Display::NewLine();
}
```

Display.h

```
/*
 * File:   Display.h
 * Author: ritoban
 *
 * Created on October 25, 2020, 7:10 PM
 */

#ifndef DISPLAY_H
#define DISPLAY_H

#include <iostream>
#include "Color.h"

class Display {
    public:
      static void BeginBold() { std::cout << "\033[1m"; }

      static void BeginUnderline() { std::cout << "\033[4m"; }

      static void BeginColor(std::uint8_t col) { std::cout << "\033[" << unsigned(col) << "m"; }

      static void NewLine() { std::cout << std::endl; }

      static void EndFormat() { std::cout << "\033[0m"; }


      template <typename T> static void Print(T x) {
        std::cout << x;
      }

      template <typename T> static void PrintBold(T x) {
        Display::BeginBold();
        std::cout << x;
        Display::EndFormat();
      }

      template <typename T> static void PrintUnderlined(T x) {
        Display::BeginUnderline();
        std::cout << x;
        Display::EndFormat();
      }
};

#endif /* DISPLAY_H */
```

SideData.h

```
/*
 * File:   SideData.h
 * Author: ritoban
 *
 * Created on October 27, 2020, 3:11 PM
 */

#ifndef SIDEDATA_H
#define SIDEDATA_H

#include <string>

enum Side { white, black };

/// A small wrpaper class that represents a piece of data that exists for both sides, and a convience m
template<typename T>
class SideData {
    public:
        SideData(T white, T black) : whiteData(white), blackData(black) {
        }
        T whiteData;
        T blackData;

        T& Get(Side s) {
            if (s == white) {
                return this->whiteData;
            } else {
                return this->blackData;
            }
        }
};


#endif /* SIDEDATA_H */
```

Color.h

```
/*
 * File:   Color.h
 * Author: ritoban
 *
 * Created on October 25, 2020, 7:26 PM
 */

#include <map>
#include <string>
#include <cinttypes>

#ifndef COLOR_H
#define COLOR_H


/* This class represents a Color -- specicially a 4 bit color based on ANSI escape sequences. See https
class Color {
public:
    Color() { };
    //Color(std::string);
    Color(uint8_t);
    Color(const Color& orig);

    virtual ~Color();

    uint8_t AsFG() { return inner + 30; }
    uint8_t AsBG() { return inner + 40; }
    static std::map<std::string, Color> CreateColorMap();


    uint8_t inner;
};

extern std::map<std::string, Color> COLOR;


#endif /* COLOR_H */
```

Color.cpp

```
/*
 * File:   Color.cpp
 * Author: ritoban
 *
 * Created on October 25, 2020, 7:26 PM
 */

#include "Color.h"

#include <map>
#include <string>

std::map<std::string, Color> COLOR = Color::CreateColorMap();

Color::Color(uint8_t val) : inner(val) {
}


Color::Color(const Color& orig) {
}

Color::~Color() {
}

// Creates a map of all of the possible 4-bit ANSI colors.
std::map<std::string, Color> Color::CreateColorMap() {
    std::map<std::string, Color> m;
    m["Black"] = Color(0);
    m["Red"] = Color(1);
    m["Green"] = Color(2);
    m["Yellow"] = Color(3);
    m["Blue"] = Color(4);
    m["Magenta"] = Color(5);
    m["Cyan"] = Color(6);
    m["White"] = Color(7);

    // TODO: Add the Bright Versions, if necessary

    return m;
}
```