YAHTZEE Board Game

Project 1

CIS-17C

Joseph Hernandez

November 2022

Contents

Introduction

Yahtzee has been a house hold delight since 1956, delivering joy to millions of people world wide. Having grown up paying countless games of Yahtzee with friends and family, I have decided to program one of my childhood favorites. Now, it is my delight to present the game of Yahtzee.

This program has taken roughly a week to develop, with approximately 6-8 hours each day. The program contains roughly 930 lines of code, and consisting of 6 individual classes - Enums, Dice, Score Card, Player, Game, and Menu.
You can find the github repository here.

Approach to Development

The approach that was taken to complete the game was to first break down the game into its constituent components, analyzing how each part operates both independently and in symbionts with the other classes. Once each part was taken out and analyzed, it was just a matter of then contemplating the data types of each class and how data would be access for each class.

Concepts

During the initial process of how the game will be structured, there had been several concepts of how the classes would be defined and the data types of each member variable. Initially, the design was to create a map for the score card. Where the key would be the name of each section and the value would be a second map with a key of the game (1-5) and the score for each category.

However, this approach would prove to be too convoluted and complicated. As iterating through each map with the appropriate score would not only be complex in code, but the complexity would have been $O(log^2 n)$, since it would be a nested loop to iterate through in order to print all the data onto a score card. A simple string array for each of the section names and an integer array for each of the games was the appropriate approach for something of this nature. This, then allowed for direct access of each element in each of the arrays, giving a complexity of $O(1)$.

To print each game and its points for each category, a 2D array was created, where the columns represent the number of games and the rows represent the category. Printing to a card has a time complexity of $O(n^2)$ since the maximum number of times it will iterate through will be $O(n)$ for the number of games and $O(n)$ for the number of categories.

```
for(int i=0;i<NumberofCategories;i++){
    for(int j=0;j<NumberofGames;j++){
        cout<<scoreCard[j][i]<<endl;
    }
}
```

The time complexity of filling a score cell is $O(1)$, since we direct access each element

```
scoreCard[Current Game][Score Category]=score;
```

Going with this approach seemed like the more appropriate way to tackle this problem. The use of maps would prove to be more beneficial in later problems.

Once the concept for how the score will be kept, it made it easy for the other classes to follow, since it allowed for a simple relationship between the classes. The class concepts were straight forward, each class would be solely responsible for each aspect of the game; this broke down to ultimately 6 classes.

Version Control

Versions were limited to each class implementation. With each new version of the game, each class was added.

Game Rules

Object

Roll dice for scoring combinations, and get the highest total score.

Game Summary

On each turn, roll the dice up to 3 times to get the highest scoring combination for one of the 13 categories. After you finish rolling, you must place a score or a zero in one of the 13 category boxes on your score card. The game ends when all players have filled in their 13 boxes. Scores are totaled, including any bonus points. The player with the highest total wins.

How to Play

Each player takes a score card. The player to go first is randomized. This simulates each player randomly rolling for the highest total.

Taking a Turn

One your turn, you may roll the dice up to 3 times, although you may stop and score after your first or second roll. Dice will auto-roll on start of each turn.

First Roll: Roll all 5 dice. Set any ”keepers” aside. You may stop and score now, or roll again. To set any keepers aside, first enter the number of dice you would like to place on hold (0-5). Next enter the dice number labeled Dice 1-5.

Second roll: Reroll ANY or ALL dice you want - even ”keepers” from the previous roll. You don't need to declare which combination you're rolling for; you may change your mind after any roll. You may stop and score after your second roll, or set aside any ”keepers” and roll a third time. To set any keepers aside, first enter the number of dice you would like to place on hold (0-5). Next enter the dice number labeled Dice 1-5.

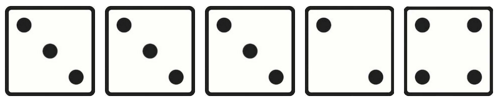
Third and Final Roll: Reroll ANY or ALL dice you want. After your third roll, you *must* fill in a box on your score card with a score or a zero. After you fill a box your turn is over.

Scoring

When you are finished rolling, decide which box to fill in on your score card. For each game, there is a column of 13 boxes. You must fill in a box on each turn; if you can't (or don't want to) enter a score, you must enter a zero. Fill in each box only once, in any order, depending on your best scoring option. To select the box to fill, type the name of the box; depending on your choice and your dice, the boxes will be filled according to the scoring combinations below.

Upper Section	**What to Score**
Aces (Ones)	Total of Aces only
Twos	Total of twos only
Threes	Total of threes only
Fours	Total of fours only
Fives	Total of fives only
Sixes	Total of sixes only

To score in the Upper Section, add only the dice with the same number and enter the total in the appropriate box. For example, the dice shown below you could score 9 in the *Threes* box, 2 in the *Twos* box or 4 in the *Fours* box.

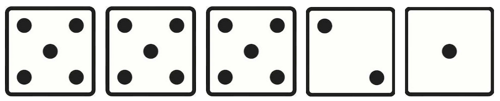


Your goal in the Upper Section is to score a total of at least 63 points, to earn a 35-point bonus. The bonus points are based on scoring on three of each number (Aces through Sixes); however, you may earn the bonus with ANY combination of scores totaling 63 points or more.

Each of the Lower Section scoring combinations is explained below in detail.

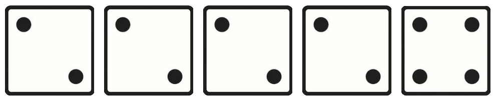
Upper Section	What to Score
3 of a Kind	Total of all 5 dice
4 of a Kind	Total of all 5 dice
Full House	25 Points
Small Straight	30 Points
Large Straight	40 Points
Yahtzee	50 Points
Chance	Total of all 5 dice

3 of a Kind: Score in this box only if the dice include 3 or more of the same number. For example, with the dice shown below you could score 18 points in the *3 of a Kind* box.



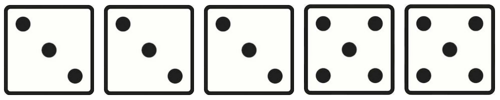
Other Scoring Options: You could instead score 18 in the *Chance* box, or you could score in the Upper Section: 15 in the *Fives* box, 2 in the *Twos* box or 1 in the *Aces* box.

4 of a Kind: Score in this box only if the dice include 4 or more of the same number. For example, with the dice shown below you could score 14 points in the 4 of a Kind box.



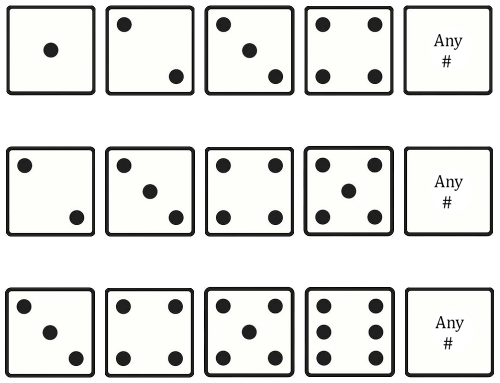
Other Scoring Options: You could instead score 14 in the *3 of a kind* box or in the *Chance* box - or you could score in the Upper Section: 8 in the *Twos* box, or 6 in the *Sixes* box.

Full House: Score in this box only if the dice show three of one number and two of another. Any Full House is worth 25 points. For example, with the dice shown below you could score 25 points in the Full House box.

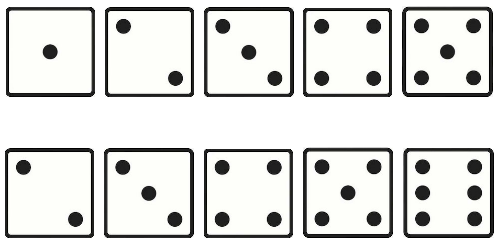


Other Scoring Options: You could instead score 19 in the *3 of a Kind* box or in the *Chance* box - or you could score in the Upper Section: 9 in the *Threes* box or 10 in the *Fives* box.

Small Straight: Score in this box only if the dice show any sequence of four numbers. Any Small Straight is worth 30 points. You could score 30 points in the Small Straight box with any of the three dice combinations shown below.

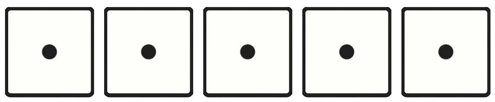


Large Straight: Score in this box only if the dice show any sequence of five numbers. Any Large Straight is worth 40 points. You could score 40 points in the Large Straight box with either of the two dice combinations shown below.



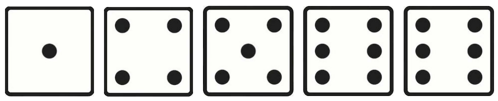
Other Scoring Options: You could instead score in the Small Straight box, the Chance box, or the appropriate Upper Section box.

YAHTZEE: Score in this box only if the dice show five of the same number (5 of a kind). A YAHTZEE example is shown below.



The first YAHTZEE you enter in the YAHTZEE box is worth 50 points. For each additional YAHTZEE you roll you earn a bonus (see YAHTZEE BONUS, below)!

Chance: Score the total of ANY dice in this box. this catch-all category comes in handy when you can't (or don't want to) score in any other category, and don't want to enter a zero. For example, you could score 22 points in the Chance box with the dice shown below.



YAHTZEE BONUS: If you roll a YAHTZEE and have already filled in the YAHTZEE box with a 50, you get a 100-point bonus! As long as you've score 50 in the YAHTZEE box, you get a YAHTZEE bonus for each additional YAHTZEE you roll. If you roll a YAHTZEE and have already entered zero in the YAHTZEE box, you do not earn a YAHTZEE bonus.

Ending A Game

Once each player has filled in all 13 category boxes, the game ends. Each player now adds up his or her score as follows:

Upper Section: Add up the Upper Section scores and enter in the TOTAL SCORE box. Enter the 35-point bonus in the BONUS box if you scored 63 points or more. Then enter the total in the TOTAL box.

Lower Section: Add up the Lower Section scores, and enter the total in the TOTAL of Lower Section box. Add 100 points for each YAHTZEE BONUS.

Grand Total: Add the Upper Section and Lower Section scores, and enter the total in this box. This is your score of the game.

How to Win

After the scores are tallied, the player with the highest Grand Total wins the game!

Solo Play

No competition around? Challenge yourself in solitaire play, and try to beat your previous scores!

Description of Code

Organization

The project is organized into the games constituent components: the face of each die, the dice, the player, the score card, the game itself, and a menu system for user interaction. The classes are associated through aggregation, in which some classes will hold other classes as object members. The relation of the classes are described in the following subsection.

Classes

Enum Class: This class holds the enumerated values for the face of each side of the die(1-6,ALL). ALL is for calculating all the values of the faces.

Dice Class: This class would keep track of the die themselves, their values, the type of dice hand present (aces, twos,sixes,3 of a kind, etc.)), rolling (randomizing) the dice and rolling the dice with some dice on "hold". The face of the die are randomized from the enumerated face values within the enumerated class.

Score Card Class: This class keeps track of the score for each player, and giving the final totals for each section. This include reading and writing the score card to a save file using the players name, i.e playerName.sav

Player Class: This class holds a score card and the name of the player. This class is also responsible for keeping track of the turn of the player throwing the dice.

Game Class: This class keeps track of all the players and checks whether each player has finished filling each category on the score card. This class also holds the dice and allows each player to use the dice.

Menu Class: This class is responsible for all the menus, reading from them from a text file and displaying to the player.

Checkoff Sheet

Container Classes

Sequences

List: This can be found within the Dice class. The list is used as a method of holding the dice.

Bit Vector: This can be found within the Score Card class and is used as a check method of both confirming whether a section has been previously filled so the player cannot double score/replace their score in that category and confirming whether the player has filled up their score card to end the game.

Associative Containers

Map: This can be found within the Dice class and is used as a way to associate the face value of the die to their respected ASCII art picture (stored as a string).

Hash: This can be found within the Score Card class and is used as a method to associate two strings, taking the subsequent hash of each sting and comparing which category is being chosen.

Container Adaptors

Stack: This can be found in the Score Card class. This is used as a method of storing whether the player has rolled a bonus Yahtzee.

Queue: This can be found in the Game class as a method of storing the players, with each player waiting in a sequence to take a turn at rolling the dice and scoring.

Iterators

Concepts

Some of the concepts of iterators were:

Forward Iterator: this was use in most cases with each container. In cases such as the list, it was used to step through each list element and retrieve the face value for each of the die.

Random Access Iterator: this iterator was used for the map container, accessing the picture for each of the face values.

Algorithms

Non-Mutating Algorithms

Count: This was used in the Player class and is used as a method of checking whether a player had entered duplicate dice numbers to hold.

Mutating Algorithms

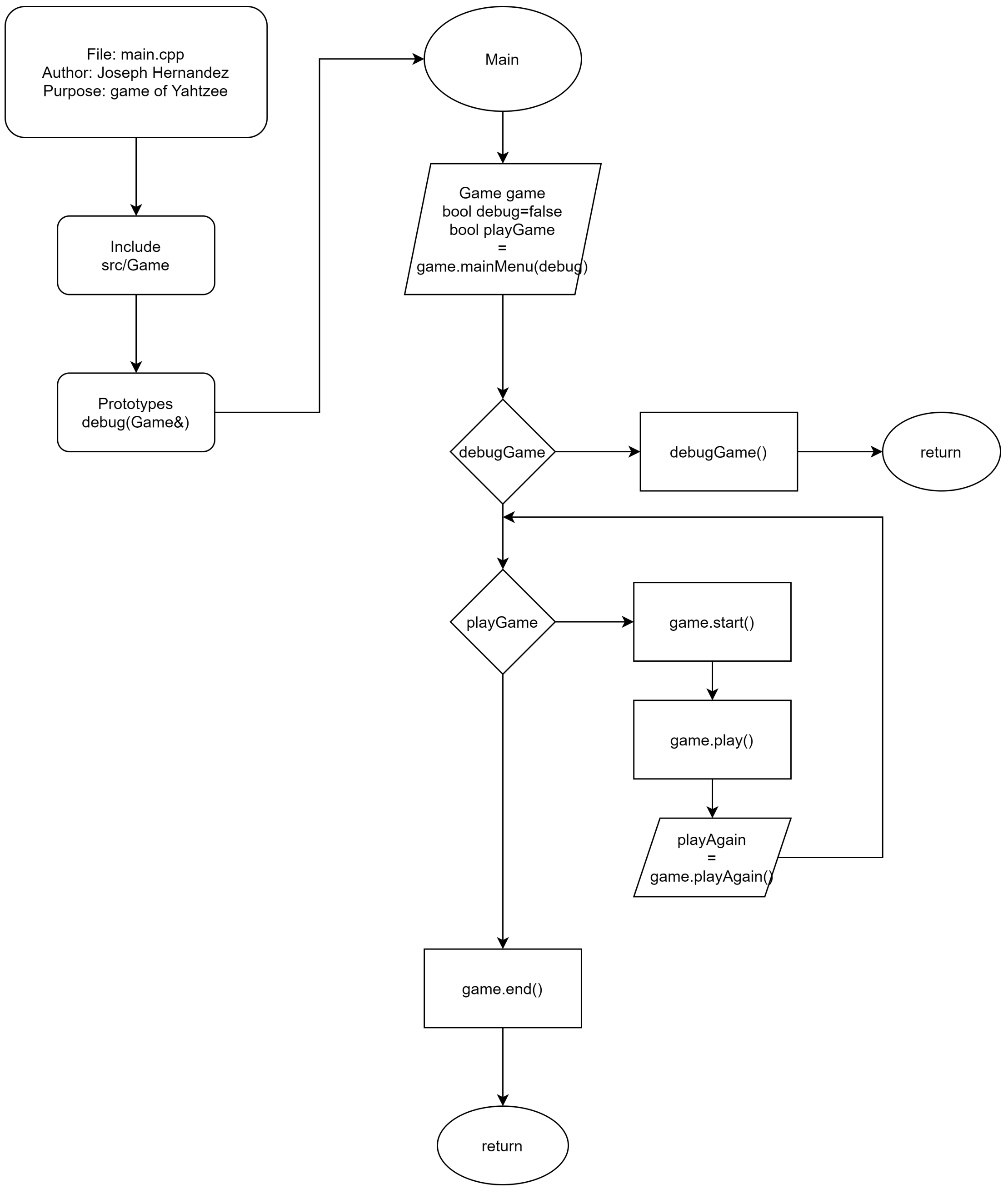
Fill: This was used to initialize the arrays in the Score Card class for the upper and lower section scores.

Organization

Sort: this was used in the Dice class to sort the dice in ascending order to display to the player (making the dice easier to read and tell what relation the dice are).

Documentation

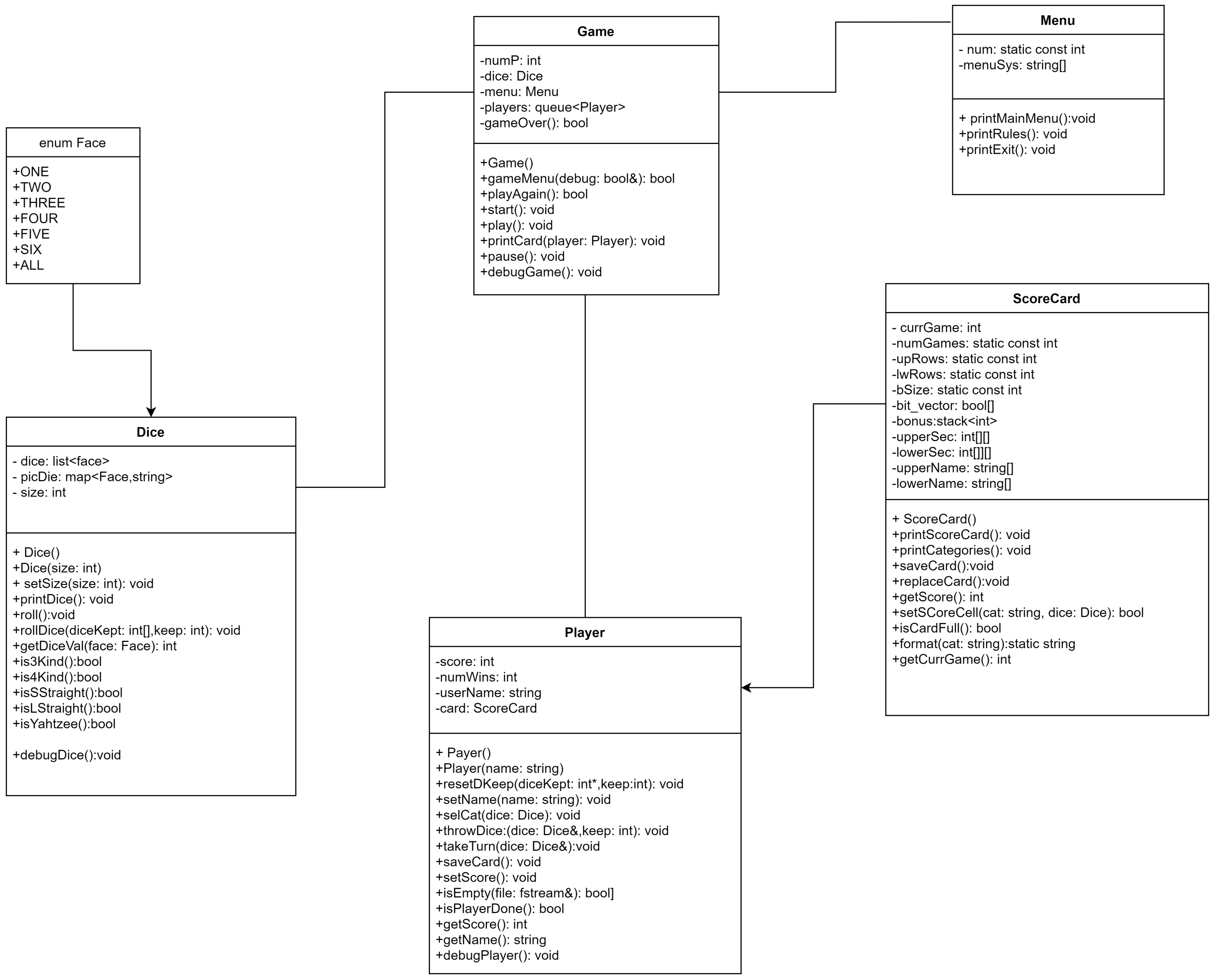
Flow Chart



Pseudocode

```
print menu
dec ← input
**if** dec $\equiv$ 8675309 **then**
    $debug \leftarrow true$
**end if**
**while** dec $> 3 \| $dec $< 1$ **do**
    dec ← input
**end while**
**if** dec $\equiv$ 2 **then**
    Print Rules
**else**
    **if** dec $\equiv$ 1 **then**
        return true
    **else**
        return false
    **end if**
**end if**
play game ← $return$
**if** $debug \equiv true$ **then**
    debugGame
    return false
**end if**
**while** $playGame$ **do**
    $gameStart$
    $gamePlay$
    playGame ← $playAgain$
**end while**
$gameEnd$
```

UML Diagram


Game
-numP: int
-dice: Dice
-menu: Menu
-players: queue<Player>
-gameOver(): bool
+Game()
+gameMenu(debug: bool&): bool
+playAgain(): bool
+start(): void
+play(): void
+printCard(player: Player): void
+pause(): void
+debugGame(): void
Menu
- num: static const int
-menuSys: string[]
+ printMainMenu():void
+printRules(): void
+printExit(): void
enum Face
+ONE
+TWO
+THREE
+FOUR
+FIVE
+SIX
+ALL
Dice
- dice: list<face>
- picDie: map<Face,string>
- size: int
+ Dice()
+Dice(size: int)
+ setSize(size: int): void
+printDice(): void
+roll():void
+rollDice(diceKept: int[],keep: int): void
+getDiceVal(face: Face): int
+is3Kind():bool
+is4Kind():bool
+isSStraight():bool
+isLStraight():bool
+isYahtzee():bool
+debugDice():void
ScoreCard
- currGame: int
-numGames: static const int
-upRows: static const int
-lwRows: static const int
-bSize: static const int
-bit_vector: bool[]
-bonus:stack<int>
-upperSec: int[][]
-lowerSec: int[][]
-upperName: string[]
-lowerName: string[]
+ ScoreCard()
+printScoreCard(): void
+printCategories(): void
+saveCard():void
+replaceCard():void
+getScore(): int
+setSCoreCell(cat: string, dice: Dice): bool
+isCardFull(): bool
+format(cat: string):static string
+getCurrGame(): int
Player
-score: int
-numWins: int
-userName: string
-card: ScoreCard
+ Payer()
+Player(name: string)
+resetDKeep(diceKept: int*,keep:int): void
+setName(name: string): void
+selCat(dice: Dice): void
+throwDice:(dice: Dice&,keep: int): void
+takeTurn(dice: Dice&):void
+saveCard(): void
+setScore(): void
+isEmpty(file: fstream&): bool]
+isPlayerDone(): bool
+getScore(): int
+getName(): string
+debugPlayer(): void