Yahtzee in C++

Organization of Programming Language

Samman Bhetwal

Ramapo College of New Jersey

1. **Bug Report**
2. Unfixed Bugs:

* When the computer fills out a category, it does not display a message indicating its action; it simply fills the scorecard without notification.
* The computer did not advise the human player to “stand” after achieving a favorable dice roll, which could have resulted in a higher score.
* When the "5 Straight" category was already filled, and the computer rolled five consecutive dice, the program displayed that the computer was filling the "5 Straight" category but ultimately filled the "4 Straight" category on the scorecard. This inconsistency between the computer’s intended action and its actual action on the scorecard needs to be resolved.

1. **Features:**
2. Missing Features:

* In the initial roll to determine who plays first, the program prompts only the human player to manually enter the die value. For the computer, it simply rolls the die randomly without a manual entry option.
* After a player’s dice roll, the program displays all open categories as potential options, regardless of the roll outcome. Instead, it should analyze the roll and only display categories that match the actual dice values. For instance, if "Twos" is open but the player rolls 1, 3, 4, 5, and 6, "Twos" should not appear as an option since it’s not relevant to the roll.
* After a player’s dice roll, the program displays all open categories as potential options, regardless of the roll outcome. Instead, it should analyze the roll and only display categories that match the actual dice values. For instance, if "Twos" is open but the player rolls 1, 3, 4, 5, and 6, "Twos" should not appear as an option since it’s not relevant to the roll.

1. Extra features:

* Smart Pointers: Implemented shared pointer for the Player instances, allowing for shared ownership and automatic memory management, which simplifies resource handling and ensures proper cleanup when players are no longer needed.
* Optional: Used optional for attributes that may not always hold a value, such as unfilled categories in the ScoreCard. This eliminates the risk of null pointers and enhances code clarity by explicitly handling the absence of values.
* Immutable Classes: Design classes like Dice and ScoreCard as immutable by using const member variables initialized through constructors, promoting safer code by preventing unintended modifications.

1. **Classes and Data Structures**
2. **Game Class**

* The Game class manages the overall gameplay of Yahtzee, including the scorecard, current round, and participating players. It is responsible for initializing game states, serializing and deserializing game data, and managing the flow of rounds. It primarily serves to:
  + Initialize game settings with a scorecard, round number, and player list.
  + Handle serialization for saving and loading game states.
  + Determine if the game is over and if it ends in a draw.
  + Facilitate the progression of rounds by invoking the round class to manage player turns and score updates.

Data Structure:

* ScoreCard: An instance of the ScoreCard class that holds the current scores for each player, tracking their progress throughout the game.
* current\_round: An integer representing the current round number in the game.
* players: A vector of shared\_ptr<Player> containing all the players participating in the game, allowing easy management of player actions and scores.

1. **Round Class**

* The Round class manages the flow of a single round of Yahtzee, coordinating player turns and dice rolling while keeping track of the scorecard's state. It leverages the Turn class to facilitate individual player actions and determines the order of play based on player scores. The primary functions include:
  + Initiating a round, checking if the round is over, and updating the scorecard based on players' scores.
  + Ensuring that all players have the opportunity to play their turn and that the round concludes correctly once all players have completed their turns or the scorecard is full.

Data Structure:

* Player Queue: A queue<shared\_ptr<Player>> that determines the order of players during the round based on their current scores, allowing for a fair and organized sequence of play. This structure helps manage which player is next to take their turn, ensuring that the gameplay flows logically and smoothly.

1. **Turn Class**

* The Turn class is responsible for managing a player's turn during the game. It facilitates the process of rolling dice, keeping track of the player's decisions, and updating the scorecard accordingly. The main responsibilities are:
* Initiating a turn for a given player, allowing for up to three rolls of the dice, displaying available scoring categories, and determining which dice the player wishes to keep.
* Ensures that player interactions are handled smoothly, while also providing options for help and strategy guidance.

Data Structure:

* Kept Dice Vector: A vector<int> that stores the values of dice that the player has decided to keep during their turn. This is initialized as empty and is updated throughout the turn as the player rolls the dice and makes decisions.

**d. Player Class**

* The Player class represents a player in the game, managing their identity, interactions, and decision-making during their turn. It allows rolling dice, deciding which dice to keep, and pursuing scoring categories. The class supports operations like comparison and copying.

Data Structure:

* Name (string): A constant string that holds the player's name. It is initialized in the constructor and cannot be changed.

**e. Human Class**

* The Human class manages the gameplay for human players, allowing them to roll dice, select categories, and interact with the game. The class mainly manages the human player’s turn which includes rerolling dice and selecting categories to score. It also provides help to the player by suggesting the best possible move given the dice combinations. Additionally, it validates moves and updates the scorecard based on the player’s choice.

Data Structure:

* Kept Dice Vector: A vector<int> that stores the dice values the human player decides to keep and not reroll.
* Available Scores Vector: A vector<pair<int, int>> that holds possible score combinations and their respective points.

**f. Computer Class**

* The Computer class extends the Player class and represents the computer player in the game. It overrides methods to implement logic for dice rolling, decision-making, and score calculation based on the current game state. The class is designed to maximize the computer's score while minimizing differences between current and optimal rolls.

Data Structure:

* Scores Available Vector: A vector<pair<int, int>> that stores available score combinations and their corresponding points.
* Kept Dice Vector: A vector<int> that stores the dice the computer decides to keep.
* Final Rolls (vector<vector<int>>): A vector of vectors to store all possible final roll based on the kept dice. This structure allows the computer to evaluate multiple strategies at once.
* Scores (vector<pair<vector<int>, int>>): A vector of pairs where each pair contains a possible final roll and its corresponding score. This structure enables the computer to assess which roll would yield the highest score.
* Dice Needed (vector<int>): A vector to identify which dice the computer needs to keep to achieve the best possible roll, based on its scoring strategy.

**g. Dice Class**

* The Dice class handles the mechanics of rolling dice. It provides static methods to roll one or multiple dice, generating random values between 1 and 6 for each roll. This class is crucial for simulating the random nature of dice rolls in the game.

Data Structure:

* Dice Rolls (vector<int>): A vector that stores the results of the dice rolls. It is dynamically allocated based on the number of dice to be rolled, allowing for efficient management of multiple rolls.

h. Scorecard Class

* The ScoreCard class manages the scoring entries for a game of Yahtzee. It maintains a mapping of game categories to their corresponding score entries, allowing players to record their scores, track winners, and determine game outcomes.

Data Structure:

* Dice Rolls (vector<int>): A vector that stores the results of the dice rolls. It is dynamically allocated based on the number of dice to be rolled, allowing for efficient management of multiple rolls.
* map: Used to associate Category enums with ScoreCardEntry objects, allowing for quick access to the score entries based on category.
* optional: Utilized to represent the possibility of a score entry not being present (e.g., optional<ScoreCardEntry> indicates that a category may not yet have a score).
* shared\_ptr: Employed for managing player objects. This allows for shared ownership of player instances, preventing memory leaks and ensuring proper resource management.
* vector: Used to store a dynamic list of categories or players, allowing for easy manipulation and retrieval of multiple entries (e.g., vector<Category> for open categories).
* string: Used for textual representation of the scorecard and for serialization/deserialization processes, enabling easy storage and transfer of scorecard data.
* int: Represents numerical values such as scores, rounds, and points, facilitating calculations and comparisons.
* enum (Category): Defines a set of named constants representing different scoring categories in the game, providing clarity and type safety.
* struct (ScoreCardEntry): Represents an individual score entry in the scorecard, encapsulating points, the winning player, and the round number, which helps organize the data associated with each scoring category.

Data Structures:

* Utilize basic data types like strings and vectors for managing user input and game state.
* Separate vector<int>s representing the current values of the dice the player has, listing the dice values required to achieve the maximum score, and detailing the dice values needed to obtain the minimum score.
* A Category enum that indicates which scoring category the player is currently aiming for.
* An integer storing the highest score that can be obtained for the pursued category and an integer indicating the lowest score that can be secured in the pursued category.

1. **Log**

Sept 1 – Sept 4

* Contacted Bibhu, Ankit, Samip, and Anish to discuss their project structures and seek advice on best practices for code organization. Discussed possible functions that could streamline game state tracking, scoring, and player interactions. (2 hours)
* Attempted to run Bibhu’s project on my computer to observe his implementation and class designs, including ways to track scores and manage game flow in an interactive environment. (2 hours)  
  Total Hours: 4 hours

Sept 5 – Sept 10

* Researched shared pointers for players and optional types in C++, enhancing my understanding of memory management and preventing leaks. This research informed my use of shared pointers within the project’s player management and score handling. (2 hours)
* Created the Scorecard class, initializing it with the current round integer and scorecard to set up structured score tracking. Began mapping out functions like is\_over and is\_draw to handle game-ending conditions. (2 hours)  
  Total Hours: 2 hours

Sept 10

* Explored version control options in Visual Studio to track project changes. Set up a Git repository to manage code revisions and maintain a history of development as I implemented major changes like scorekeeping and game state tracking. (1.5 hours)  
  Total Hours: 1.5 hours

Sept 12

* Implemented a basic scorecard display that allows users to view scores in real-time. This included functions like show\_scores, displaying each player’s name and score to make score tracking accessible throughout gameplay. (2 hours)  
  Total Hours: 2 hours

Sept 15

* Developed dice-rolling functionality and visual feedback for each die result, using helper functions like roll\_die and roll\_dice to manage individual rolls and store results. (1.5 hours)  
  Total Hours: 1.5 hours

Sept 16

* Enabled rolling dice up to three times per turn with the option to keep certain dice between rolls. Created prompts for users to select dice to keep, integrating functions such as get\_dice\_to\_keep and get\_yes\_no for enhanced interaction. (2 hours)  
  Total Hours: 2 hours

Sept 18

* Added the ability for users to enter scores and select categories, updating the scorecard and ensuring scoring rules were correctly implemented with helper functions like show\_category\_pursuits and get\_category\_selection to guide users through the scoring process. (2 hours)  
  Total Hours: 2 hours

Sept 21

* Expanded gameplay to support 12 rounds of dice rolling, scoring, and scorecard updates, establishing the core gameplay loop. (2 hours)  
  Total Hours: 2 hours

Sept 22

* Implemented validation to ensure users select valid categories and scores, adding error handling and user feedback with functions like show\_categories to display available categories. (1.5 hours)  
  Total Hours: 1.5 hours

Sept 24

* Refactored the code (2 hours) to follow an object-oriented design for better maintainability. Restructured classes and methods such as get\_player\_scores and show\_scores, encapsulating related functions within relevant classes. (2 hours)  
  Total Hours: 4 hours

Sept 26

* Enabled turn-based play, where both human and computer players roll a single die, setting up the interaction flow. This was the initial step in creating a fully interactive game with both players. (3.5 hours)  
  Total Hours: 3.5 hours

Sept 30

* Completed the tournament loop, enabling a full game session and adding a manual dice input feature to allow flexibility in testing. This included functions such as wants\_to\_stand and show\_help. (5 hours)  
  Total Hours: 5 hours

Oct 2

* Integrated the computer player, initially relying on human input for moves. This setup helped structure computer decision-making in future iterations. (1.5 hours)  
  Total Hours: 1.5 hours

Oct 6

* Added turn-switching functionality for human and computer players. Resolved a turn-switching bug by brainstorming with ChatGPT, leading to implementing a queue structure to streamline turn management. (1.5 hours)  
  Total Hours: 1.5 hours

Oct 8

* Brainstormed computer strategies with ChatGPT for achieving Yahtzee, prioritizing dice categories based on roll results. Created functions like atleast\_n\_same and contains\_n to identify and evaluate dice combinations dynamically. (2.5 hours)  
  Total Hours: 2.5 hours

Oct 10

* Developed initial strategies for four and five dice sequences. Functions such as get\_sequence and longest\_sequence\_length were explored to support evaluating roll results and guiding reroll decisions. (3.5 hours)  
  Total Hours: 3.5 hours

Oct 12

* Continued brainstorming full-house strategies, documenting scenarios to determine whether pursuing a full house or three of a kind would yield more points. Collaborated with ChatGPT to implement a function that compares these values for strategic decisions. (1.5 hours)  
  Total Hours: 1.5 hours

Oct 14

* Attempted to implement sequence strategies but faced challenges when the initial code prioritized Yahtzee even when achieving other sequences. Refined the approach with ChatGPT, adding functions like full\_house and atleast\_four\_same to better support diverse strategy decisions. (2 hours)  
  Total Hours: 2 hours

Oct 15

* Conducted additional research on Yahtzee strategies to improve computer decision-making. Documented algorithms for various dice combinations, using generate\_combinations and dice\_combinations to support evaluating and optimizing roll results. (2 hours)  
  Total Hours: 2 hours

Oct 17

* Implemented a basic decision-making algorithm for the computer, focusing on high-value combinations. Tested this in a simulated environment, assessing functions such as count\_unique and num\_repeats for evaluating roll characteristics. (2.5 hours)  
  Total Hours: 2.5 hours

Oct 20

* Enhanced the computer logic to prioritize certain combinations over others based on the dice state, updating reroll logic with functions like get\_dice\_permutation to assess potential results. (2 hours)  
  Total Hours: 2 hours

Oct 22

* Developed and tested an evaluation method for the computer’s current dice rolls to determine which strategy to pursue. This phase focused on building a scoring hierarchy for different roll values. (2 hours)  
  Total Hours: 2 hours

Oct 24

* Finalized the computer strategy, thoroughly testing player interactions, scoring updates, and roll evaluations. Implemented game serialization with save\_game\_procedure and deserialize functions to allow game saving and loading for a complete gaming experience. (2.5 hours)  
  Total Hours: 2.5 hours

**5. Generative AI Assistance**

Location in Code:

Class: Computer

Function: get\_dice\_to\_keep ()

Functionality: When the computer gets dice values which fulfills the Four Straight and Five Straight categories, it used to roll again to aim for Yahtzee (if Yahtzee available) even if these sequence categories were empty. This needed to be fixed.

Nature of Assistance:

I had the idea in mind and shared it with ChatGPT. I needed to confirm or get an outside opinion on the idea to reconfirm the robustness of the computer strategy. I wrote down the code and was facing small errors which I used break points and ChatGPT to debug.

Description of Help:

The AI provided guidance on using cout statementsto determine where I was facing the problem, determine if the computer was actually keeping the dice values I wanted it to keep and I thought it would keep. This helped me test out my computer strategy so that it would not make a mistake of rolling the dice values trying to get Yahtzee when it already has 4 Straight or 5 Straight dice values.

This solution ensures that the computer will prioritize Yahtzee but not to the expense of other high point yielding sequence categories.

* 1. **Screenshots**

A screenshot of a computer game

Description automatically generated

Fig: First roll showing who goes first

A screenshot of a computer game

Description automatically generated

Fig: Showing Computer going for Five Straight and filling the appropriate category and human’s turn after computer’s.

A screenshot of a computer game

Description automatically generated

Fig: Manual entry of the dice and showing potential categories based on the dice combination. Also displays the option for human to ask for help.

A screenshot of a computer program

Description automatically generated

Fig: When user says yes, computer provides help and asks if the user wants to stand or not.

A screenshot of a computer program

Description automatically generated

Fig: Loading a saved game. The player with less scores goes first.

A screenshot of a computer game

Description automatically generated

Fig: Human fills the scorecard and program asks to save the game.

A screenshot of a computer game

Description automatically generated

Fig: Manually input the dice values for computer, computer filling category, displaying scorecard, and asking to save the game or not.

A screenshot of a computer game

Description automatically generated

Fig: When all the categories are filled, the tournament ends displaying who won the game