Problem Statement:

Create the dice game Farkle using the C++ .NET framework. Allow the player to roll up to 6 dice, select from those dice to set aside for points, reroll the dice, and lose points on a FARKLE. After a player reaches 10,000 points, give all other players one turn to overtake the score.

Process and Pseudocode:

When selecting from one of the three game options I wanted a game that I could imagine making at that current point in time using what I already knew about C++ and other programming languages. LCR felt like a test in using linked lists and traversing the nodes from left to right, while Liar’s Dice felt more suited to a table top rather than a program. Looking back at it now I can think of some ways to do Liar’s Dice but still feel I would not have enjoyed the creation process as much as I did for Farkle.

Farkle appeared easy enough to do, with the most complication being keeping track of the dice, how many to roll, what points the player chose from those dice, etc. So I began with making a few methods in pseudocode based on the rules document. This consisted with the setup function that would get the number of players and their names, and the main looping function that rolls the dice. However I knew that I would need to add more functions in the future as I expanded on certain elements of the program. At the time I knew I wanted to have a way for the player to view a scoreboard, drop out of the game, quit the game, roll the dice, and cash the dice.

| Setup function: PRINT welcome to Farkle PRINT how many players will there be today? READ number of players PRINT give an alias **for** each player FOR all players  PRINT player name  WRITE name in player "profile" END FOR PRINT Game will start with Player 1 (alias)   Loop function: LOOP **while** input is **not** close  PRINT What is your command?  PRINT **commands** (roll / roll again, drop out, quit application, score dice) READ input command  CALCULATE input command  IF quit application  EXIT **switch**  ELSE IF roll / roll again IF has score  Roll again ELSE   Roll initial  ELSE IF score dice  IF can score dice  Add dice score to player score  ELSE IF drop out  PRINT Player quits the game  REMOVE player from game list  EXIT **switch**  ELSE IF quit application  PRINT application closing...  EXIT **switch**  END IF |
| --- |

This gave me a generalized groundwork for what I needed in terms of functions. I knew I needed a function to roll the dice, a class to hold the players information, an initializer function that sets the games initial state, a menu function that determines the users input.

| Global INITIALIZE Players array INITIALIZE current turn = -1 //position in array of players, -1 means game just started INITIALIZE current roll array || size of 6  Players Object INITIALIZE player name INITIALIZE player score = 0 INITIALIZE winner FALSE Constructor SET player name   On Scoreboard Function  IF score >= 500  RETURN TRUE   Add Score Function  Player score += points  IF player score >= 10000  SET winner TRUE    Loop function: LOOP **while** input is **not** close  PRINT What is your command?  PRINT commands (roll / roll again, drop out, quit application, score dice) READ input command CALCULATE FUNC input command     Calculate Input Function: CALCULATE input command  IF quit application  EXIT **switch**  ELSE IF roll / roll again IF has score  Roll again ELSE  Roll initial  ELSE IF score dice  IF can score dice  Add dice score to player score  ELSE IF drop out  PRINT Player quits the game  REMOVE player from game list  EXIT **switch**  ELSE IF quit application  PRINT application closing...  EXIT **switch**  END IF    Roll Dice Initial Function INITIALIZE gotPoints to FALSE PRINT Roll Dice WAIT **for** keypress FOR 6 dice  CALCULATE dice rolls  SET dice roll in array  IF roll is point AND got points is FALSE  SET got points TRUE  IF got points FALSE  RETURN farkle //probably will be an empty array? ELSE  RETURN points as array //having each point be in an array will help to determine how many dice contain points and how many can be rerolled. |
| --- |

At this point I was itching to actually program so I began to implement what I already had into the IDE.

| **int** currentTurn = -1; //What player currently has control **int** currentRoll[6] = { 0 }; //The scoring sides of the dice roll **int** tempScore = 0; //Used for when the current player rolls more than 6 dice //Function Declarations **void** **PrintCommands**();   **class** **Player** {  string \_name; //Alias for the player  **int** \_score = 0; //The amount of points the player has cashed  **bool** canWin = false; //If the player has enough points to win   //Constructor **public**:  Player(string name) {  **this**->\_name = name;  }   //Returns true if the player can be placed on the scoreboard,  // otherwise false  **bool** **OnScoreboard**() {  **if** (\_score >= 500) {  **return** true;  }   **return** false;  }   //Adds points to the scoreboard given the sum of points  **void** **AddScore**(**int** points) {  \_score += points;  **if** (\_score >= 10000) {  canWin = true;  }  } };  **void** **CalculateInput**(string command) {  //Quit program  **if** (command == "Q" || command == "q") {  **return**;  }  //Roll  **else** **if** (command == "R" || command == "r") {  **if** (tempScore == 0) {  //Roll Initial  cout << "STUB - Roll Dice Initial" << endl;  **return**;  }   //Roll Again  cout << "STUB - Roll Dice Again" << endl;  }  //Cash dice  **else** **if** (command == "C" || command == "c") {  cout << "STUB - Cash Dice" << endl;  //Loop through current roll and add score dice to temp score  //Add temp score to current players score  }  //View Scoreboard  **else** **if** (command == "S" || command == "s") {  cout << "STUB - Scoreboard" << endl;  }  //Drop out  **else** **if** (command == "D" || command == "d") {  cout << "STUB - Drop Out" << endl;  }  //Default  **else** {  cout << "Command not recognized..." << endl << endl;  } }  **void** **PrintCommands**() {  cout << "STUB - Print Commands" << endl; }  **void** **InitializeGame**() {  **int** numOfPlayers = -1;  **bool** set = false;   **do**  {  **if** (numOfPlayers < 2 && set) {  cout << "Farkle requires at least 2 players to start..." << endl;  }   cout << "How many players will be playing today? : ";  cin >> numOfPlayers;  set = true;  } **while** (numOfPlayers < 2); }  **void** **main**() {  string input = "";   InitializeGame();   **do** {  PrintCommands();   cout << "What is your command? ";  cin >> input;   //STUB - If current player is a winner, end game   CalculateInput(input);   } **while** (input != "Q"); } |
| --- |

At this point I began to look at creating a full function for the dice roll. My initial markup was simple and would be later modified to be more robust.

| **void** **RollDice**(**int** diceRoll[6], **int** diceToRoll) {   **switch** (diceToRoll) {  **case** 6:  diceRoll[5] = rand() % 6 + 1;  **case** 5:  diceRoll[4] = rand() % 6 + 1;  **case** 4:  diceRoll[3] = rand() % 6 + 1;  **case** 3:  diceRoll[2] = rand() % 6 + 1;  **case** 2:  diceRoll[1] = rand() % 6 + 1;  **case** 1:  diceRoll[0] = rand() % 6 + 1;  **break**;  **default**:  **break**;  } } |
| --- |

The idea was that if the player rolled the max amount of dice, every other dice would roll as well. Being the first concept that popped into my head it worked as intended but was redundant in that the relatively same line of code was used 6 times for each dice.

Now that the dice worked I looked at scoring the dice. My initial thought was to sort the dice faces in ascending order and loop through the array looking for duplicates.

| CALCULATE SORT dice FOR EACH dice   IF dice face is the same as the 2 previous dice faces  ADD 3 of a kind score  ELSE IF dice face is a 1  ADD 100 points  ELSE IF dice face is a 5  ADD 50 points |
| --- |

| //Sort the rolls in ascending order sort(currentRoll, currentRoll + 6);  //Loop through all the dice, starting with the highest **for** (**int** i = 5; i >= 0; i--) {  //3 of a kind  **if** (i > 2 &&  currentRoll[i] == currentRoll[i - 1] &&  currentRoll[i - 1] == currentRoll[i - 2]) {   //If the roll is a 1, do different algorithm  **if** (currentRoll[i] == 1)   tempScore += currentRoll[i] \* 1000;  **else**   tempScore += currentRoll[i] \* 100;   i -= 2;  }  //100 points  **else** **if** (currentRoll[i] == 1)  tempScore += currentRoll[i] \* 100;  //50 points  **else** **if** (currentRoll[i] == 5)   tempScore += currentRoll[i] \* 10; } |
| --- |

Having the dice and scoring work in a rudimentary way allowed me to begin working on the next functions which were changing the turn and resetting the dice.

For resetting the dice I thought of simply looping through the dice array and setting each value to zero.

| FOR EACH dice in array  SET dice at index to zero SET score to zero SET dice scored to zero |
| --- |

| //Reset all current turn values to their defaults **void** **ResetValues**() {  **for** (**int** i = 0; i < 6; i++) {  currentRoll[i] = 0;  }    score = 0;  diceScored = 0; } |
| --- |

As for getting to the next turn it was just incrementing the currentTurn variable I had established in the globals at the start of the project.

| //Set the turn to the next player in the list **void** **NextTurn**() {  currentTurn++;   **if** (currentTurn >= numOfPlayers)  currentTurn = 0;   ResetValues(); } |
| --- |

Now that the game was more complicated I was in need of the InitializeGame function to represent that. The player's turn was set so now I needed a way to know what player was supposed to play. This and the random time needed to be set for rolling the dice and the current turn initially set once all players have been initialized.

Adding to the InitializeGame function:

| FOR EACH player  PRINT What is the player's name?  GET input  SET players name to input |
| --- |

| **void** **InitializeGame**() {  **bool** set = false; //Used to determine if number of players has been set   srand(time(NULL)); // initialize random seed:   **//…**  //Make dynamic array of players  players = **new** Player[numOfPlayers];   //Prompt for player aliases  **for** (**int** i = 0; i < numOfPlayers; i++)  {  string input;  cout << "What is Player " << i + 1 << "'s alias? ";  cin >> input;   players[i].SetName(input);  }   currentTurn = 0;   system("cls"); } |
| --- |

Now to save the points to the player. Currently the points are tied to the current turn as a global variable and does nothing other than that. The turn does not change and this is because the farkle has not been implemented and the cashing of the score is a stub.

| IF players score is 0 **and** the current score is less than 500  PRINT need 500 points to get on the scoreboard  RETURN  PRINT banked x points ADD score to players score PRINT current points  CALL **NextTurn**() |
| --- |

| **void** **CashScore**() {  **if** (players[currentTurn].GetScore() == 0 && score < 500) {  cout << "\nYou can not cash your score now!\nYou have to have at least 500 points to get on the scoreboard!" << endl;  **return**;  }    cout << "\nYou banked " << score << " Points!" << endl;  players[currentTurn].AddScore(score);  cout << "You now have " << players[currentTurn].GetScore() << " Points.\n" << endl;  NextTurn(); } |
| --- |

With this the game could work to some degree with players choosing to bank their points to end their turn. This made the problem that I now needed to display whose turn it was though, and a little flair in that it should show how many points they have and can bank.

| PRINT current turn PRINT players current points PRINT **if** on scoreboard **or** **not** **and** their points |
| --- |

| //Prints information on the players current turn and their info **void** **PrintTurnInfo**() {  system("cls");   cout << "It is " << players[currentTurn].GetName() << "'s Turn" << endl << endl;  cout << "You have " << score << " points that can be cashed." << endl;  **if** (players[currentTurn].OnScoreboard())  cout << "You are on the scoreboard with " << players[currentTurn].GetScore() << " points." << endl;  **else**  cout << "You are not on the scoreboard: get 500 points to enter the race!" << endl << endl; } |
| --- |

This seemed like a good time to go and do the game’s namesake - the FARKLE. This would be added to the ScoreDice function that checks each dice face rolled.

| IF points scored **this** turn is 0  PRINT FARKLE  CALL Next Turn function ELSE  ADD points scored **this** turn to the players current turn score  PRINT points earned |
| --- |

| **void** **SortDice**() {  **int** points = 0;   //Sort the rolls in ascending order  sort(currentRoll, currentRoll + 6);    //Loop through all the dice, starting with the highest  **for** (**int** i = 5; i >= 0; i--) {  //3 of a kind  **if** (i > 2 &&  currentRoll[i] == currentRoll[i - 1] &&  currentRoll[i - 1] == currentRoll[i - 2]) {   //If the roll is a 1, do different algorithm  **if** (currentRoll[i] == 1)   points += currentRoll[i] \* 1000;  **else**   points += currentRoll[i] \* 100;   i -= 2;  }  //100 points  **else** **if** (currentRoll[i] == 1)  points += currentRoll[i] \* 100;  //50 points  **else** **if** (currentRoll[i] == 5)   points += currentRoll[i] \* 10;  }  **if** (tempScore == 0) {  Cout << "FARKLE" << endl;  NextTurn();  }  **else** {  tempScore += points;  cout << "You earned " << points << " points!" << endl;  } } |
| --- |

I also went back and updated the RollDice function

| **void** **RollDice**(**int** diceRoll[6], **int** diceToRoll) {  **for** (**int** i = diceToRoll - 1; i >= 0; i--) {  diceRoll[i] = rand() % 6 + 1;  } } |
| --- |

With the ability to score points and lose there still needed to be a way to see who had what points - thus a scoreboard needed to be made. When first thinking up a way to do the scoreboard function I thought of arranging the players array based on their score. This ultimately would require a comparator function to compare the variables inside the player class as well, something I was not well versed in. I would need to do some research on how to make a custom comparator for the sort() function.

| PRINT Scoreboard: INITIALIZE temp array **for** players FOR EACH player  SET temp player at index to player at index SORT temp array **using** comparator FOR EACH player  IF player has a score over 500  PRINT player name, score, **and** position  ELSE  PRINT player name **and** unranked position |
| --- |

| //Prints the scoreboard to the console **void** **PrintScoreboard**() {  cout << "Scoreboard:" << endl;   //Create temp array to sort players by score while keeping play order  Player\* temp = **new** Player[numOfPlayers];   //Copy each object to the new array without copying address  **for** (**int** i = 0; i < numOfPlayers; i++)  {  temp[i] = players[i];  }   sort(temp, temp + numOfPlayers, ScoreComparer);   //for each player  **for** (**int** i = 0; i < numOfPlayers; i++)  {  //If they are on the board, print position and score  **if** (temp[i].OnScoreboard())  cout << "[" << i + 1 << "] : " << temp[i].GetName() << " : " << temp[i].GetScore() << endl;  //Otherwise print name without position  **else**  cout << "[-] : " << temp[i].GetName() << endl;  }   **delete**[] temp; //Clear memory } |
| --- |

<https://stackoverflow.com/questions/16894700/c-custom-compare-function-for-stdsort>

| //Used by the scoreboard to determine who has the higher score **bool** **ScoreComparer**(Player playerOne, Player playerTwo) {  **return** playerOne.GetScore() > playerTwo.GetScore(); } |
| --- |

This left me with the last major part of the program that I wanted to add which was allowing the player to drop out of the game - because why not. It was also around this time that I began separating each function between different classes - the game and the core. The core would handle more of the frontend work while the game consisted of adding and removing players, rolling the dice, and generally keeping track of the game state.

| DECREMENT player counter INITIALIZE temp array **for** players FOR EACH player on counter  IF the index is the current players turn AKA the player that wants to drop out  SKIP  SET temp array at index to player array at index  IF too few players  STOP GAME CALL **NextTurn**() |
| --- |

| //Removes the current player from the array of players and starts the next turn **void** Game::Dropout() **const** {  //Decrement number of players  numOfPlayers--;   **int** p = 0; //placeholder  Player\* temp = **new** Player[numOfPlayers]; //Temp dynamic array   //loop through the array and copy the elements  **for** (std::**size\_t** i = 0; i < numOfPlayers; i++) {  //If on the player that wants to drop out, increment placeholder  **if** (i == currentTurn) p++;   temp[i] = players[p];  p++;  }    **delete**[] players; //free memory  players = temp;   //If under the minimum number of players to play, end early  **if** (numOfPlayers < 2) **return**;   NextTurn(); } |
| --- |

Now with everything out of the way I just needed to make it so the player could win the game.

| INITIALIZE winner score INITIALIZE winner index position FOR EACH player   IF player score is over 10,000 **and** greater than winner score  SET winner score = player score  SET winner index position to loop index   IF winner index position is set  IF winner alias matches player alias  PRINT winner  ELSE  PRINT player is about to win |
| --- |

| //Checks the scores of every player and prints if a player is about to win or has won **int** Game::CheckForWin() **const** {  **int** winnerPos = -1;  **int** winnerScore = 0;   **for** (**int** i = 0; i < numOfPlayers; i++) {  //Get current players score  **int** score = players[i].GetScore();    //If that score is a winning score and it is the highest score thus far  **if** (score >= 10000 && score >= winnerScore) {  //set the new winning score  winnerScore = score;  //set the winning player  winnerPos = i;  }  }   **if** (winnerPos != -1) {  //If the winner has been set and every other player has rolled again  **if** (winner.GetName() == players[currentTurn].GetName()) {  cout << players[winnerPos].GetName() << " wins the game with " << players[winnerPos].GetScore() << " points!" << endl;  winner = players[winnerPos];  **return** 1;  }  **else** {  cout << players[winnerPos].GetName() << " is about to win with " << players[winnerPos].GetScore() << " points!" << endl;  winner = players[winnerPos];  }  }  **return** 0; } |
| --- |

Huzzah! It’s done. To make certain I got everything for the game I made a checklist from the games rule documents - only to find that the player needs to be able to choose what combination of point dice to set aside. At this point I had it set aside the highest value of any point dice present. But Farkle is about strategy, choosing dice to take now versus rerolling the dice for more points. So I did what any good programmer would do - make things more complicated by adding more work for myself. I went and looked at the official rules for Farkle and realized that there were a load more combinations of point dice that could be made: 4 of a kind, 5 of a kind, 6 of a kind, 3 sets of 2, 2 sets of 3, 4 of a kind + a pair, a straight. These combinations would also net the player thousands of points in exchange of being locked out after some dice are set aside, making them much harder to get. But why the hell not, let's add them in! This led me to reworking the ScoreDice function.

Looking at what I already had I guessed that I could keep the 3 of a kind and single detection, only needing to add more statements to check for the other combinations. On top of that I needed to allow the player to select from the available combinations to set aside the dice, get the associated score, and be a happy little camper. Oh the joy. My initial thought was using an array as a table. C# has multidimensional arrays and as far as I could tell so does C++, however I wanted the array to be dynamic which means it can not be multidimensional. So instead I created a way to allow for a dynamic array to act like a multidimensional array by setting each few indices in the index as a row associated with the combination. Next was what to store in each row. I knew I needed to have the index to allow for the combination to be selected from a list of combinations. Next was a code to identify what type of combination it was so that points could be allocated on selection. At first this code was hardset in the lines of code but I kept changing the code so I made an enum to handle all the codes in one spot. The codes would also help differentiate between multiple types of a given combination. Four sets of two could be the first 4 dice and the last 2 dice, or the first 2 dice and the last 4 dice.

| **enum** DiceComboCodes {  SixofKind = 15,  TwosetsThree = 14,  FoursetsTwoT1 = 13,  FoursetsTwoT2 = 12,  ThreesetsTwo = 11,  Straight = 10,  FiveofKindT1 = 9,  FiveofKindT2 = 8,  FourofKindT1 = 7,  FourofKindT2 = 6,  FourofKindT3 = 5,  ThreeofKind = 4,  SingleFive = 3,  SingleOne = 2,  Reroll = 1,  Stop = 0 }; |
| --- |

Last in the array is the starting index in the dice array. This is mainly used by the SingleFive and SingleOne codes to determine what dice to move from the current roll array to the scored dice array. This left me with the current array key:

| //(0i) Input binding | (1i)Type of Cash | (2i) Start Index in array |
| --- |

With input binding being what integer key the user will input, type of cash being the combo code, and start index being the start index in the dice array.

Then akin to my original dice roll design I made a switch statement looking at the amount of dice being rolled, with 6 dice being at the top and 1 die being at the bottom. Each switch case would look through each combination for the given amount of dice by comparing each dice face with one another in an if statement. If a combination is found, then the selection array is increased by 3 units and the given information is added. The code is the same for each combination with differences in the index and code. Here is a snippet of Six of a Kind:

| //Resize array to add new combination ResizeIntArray(selection, (combinations + 1) \* 3); selection[combinations \* 3] = combinations; //Binding selection[combinations \* 3 + 1] = DiceComboCodes::SixofKind;//Code selection[combinations \* 3 + 2] = 0; //Index  combinations++; |
| --- |

<https://stackoverflow.com/questions/8056746/copying-from-one-dynamically-allocated-array-to-another-c>

| **void** GameInfo::ResizeIntArray(**int**\*& orig, **int** size) {  **int**\* resized = **new** **int**[size]; //make new temp dynamic array   copy(orig, orig + size, resized); //copy from original to new    **delete**[] orig; //free memory from original array  orig = resized; //set original array to new array } |
| --- |

This resize array method was also repurposed for resizing the players arrays.

Once all the combinations are added, a check is done to see if there are any combinations - if not it is a farkle and the next turn starts. But if there are combinations then another switch checks the users input for the code associated with the binding they entered.

| **switch** (selection[input \* 3 + 1]) {...} |
| --- |

Here is a snippet of one of the cases. Since all these cases share the same amount of points given there is no break between cases.

| // 1-6 straight  // 3 sets of 2  // 4 sets + 2 (2)  // 4 sets + 2 (1)  **case** DiceComboCodes::Straight:  **case** DiceComboCodes::ThreesetsTwo:  **case** DiceComboCodes::FoursetsTwoT1:  **case** DiceComboCodes::FoursetsTwoT2:    **for** (**int** i = 0; i < 6; i++) {  savedRoll[diceScored] = roll[i];  diceScored++;  diceToScore--;  roll[i] = 0;  }   pointsScored += 1500;  **break**; |
| --- |

With the player able to play Farkle with some strategy they only needed to know the rules and possible combinations. At this point we had done the lab assignment that looks at reading text files so I created a helper FileReaderclass and imported what I had already done.

<https://cplusplus.com/reference/fstream/fstream/?kw=fstream>

<https://www.guru99.com/cpp-file-read-write-open.html>

| //Reads a file of the given file name and returns the contents of said file as a string. string FileReader::ReadFile(string fileName) {  string output;  fstream file;   //Open file to read  file.open(fileName, ios::in);   //If file exists  **if** (file) {  **char** out;   **while** (true) {  //If end of file is found, break loop  **if** (file.eof()) {  **break**;  }   //Read each line and print to console  **for** (string line; getline(file, line);) {  output.append(line + "\n");  }  }   output.append("\n\n");  }  **else** **return** "";   //Close file once done  file.close();   **return** output; } |
| --- |

This would allow me to have a text file containing the rules instead of having to hardcode the information.

Once I had everything up and working I recruited a friend of mine to test and give feedback on the current state of my program.

While he understood that this was a program to simulate a dice game he had no idea what Farkle was so it was a good test to see if the rules document I had made helped to explain things and if the program was intuitive enough. I watched him as he played the game and realized the rules document needed to be reworded slightly to increase clarity and had to show some more key information about the game.

He also felt that the amount of text on the screen was overwhelming at times, especially when rolling the dice. This was because each instance of a combination added a keybind input. So if there were three 1’s there would be 3 of a kind and 3 instances of the single 1. He suggested finding a way to consolidate the 3 instances into a single instance and simply show it again as long as it was available when the menu updates after selecting a dice combination. He also did not see the special combination to stop scoring dice at the bottom of the list that returns the user to the main menu. At the time going back to the main menu was the only way to reroll the dice after scoring. He stated that the inability to reroll the dice from the rolling screen was a tad annoying and if there was an option to reroll from said screen it would speed up the process. For readability he questioned if the text color could be changed, and jokingly stated I should take this program and remake it in a game engine to render a 3D model of the dice.

All of his comments were good feedback.

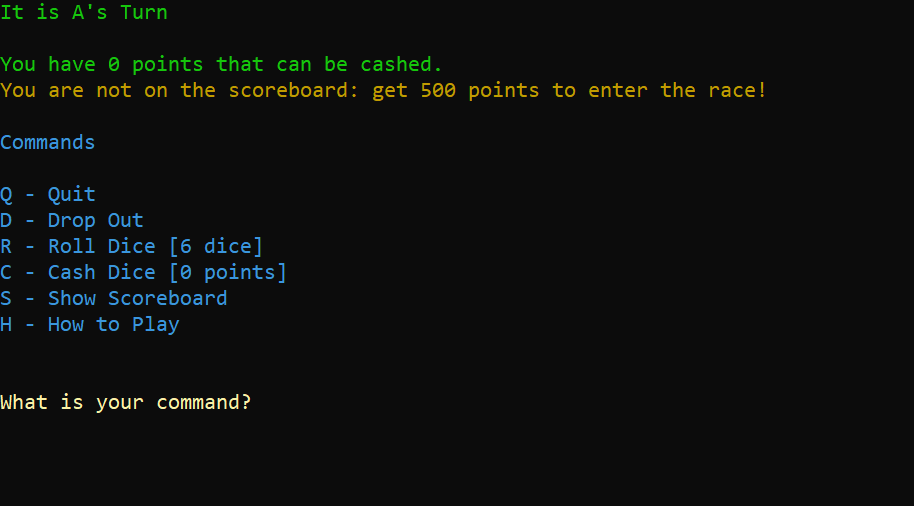
I started with coloring the text. This required a rework of my current cout printing to the console window. I started with looking up if it was even possible to recolor the text, and surprisingly it is. I did not think that the windows cmd supported colored text but I guess it does.

<https://www.codespeedy.com/color-text-output-in-console-in-cpp/#:~:text=In%20this%20article%2C%20we%20will%20discuss%20how%20we,output%20console.%20Handle%20var_name%20%3D%20GetStdHandle%28STD_OUTPUT_HANDLE%29%3B%20SetConsoleTextAttribute%28var_name%2C%20color_code%29%3B>

I used what I found to create a function that takes in a string, and 2 integers for the foreground and background colors. This function would print the colored text then return the color back to white, so in the event I needed to do a normal cout it would not be in the last color used.

| //Takes in a string and a foreground and background color to print to the console **void** GameInfo::PrintWithColor(string text, **int** foregroundColor, **int** backgroundColor) **const** {  HANDLE hConsole = GetStdHandle(STD\_OUTPUT\_HANDLE);   //set to custom color  SetConsoleTextAttribute(hConsole, foregroundColor + backgroundColor \* 16);  cout << text;   //Reset to white  SetConsoleTextAttribute(hConsole, 15); } |
| --- |

Once this was working properly I went through the program and changed all the cout instances to function calls - getting me this output.

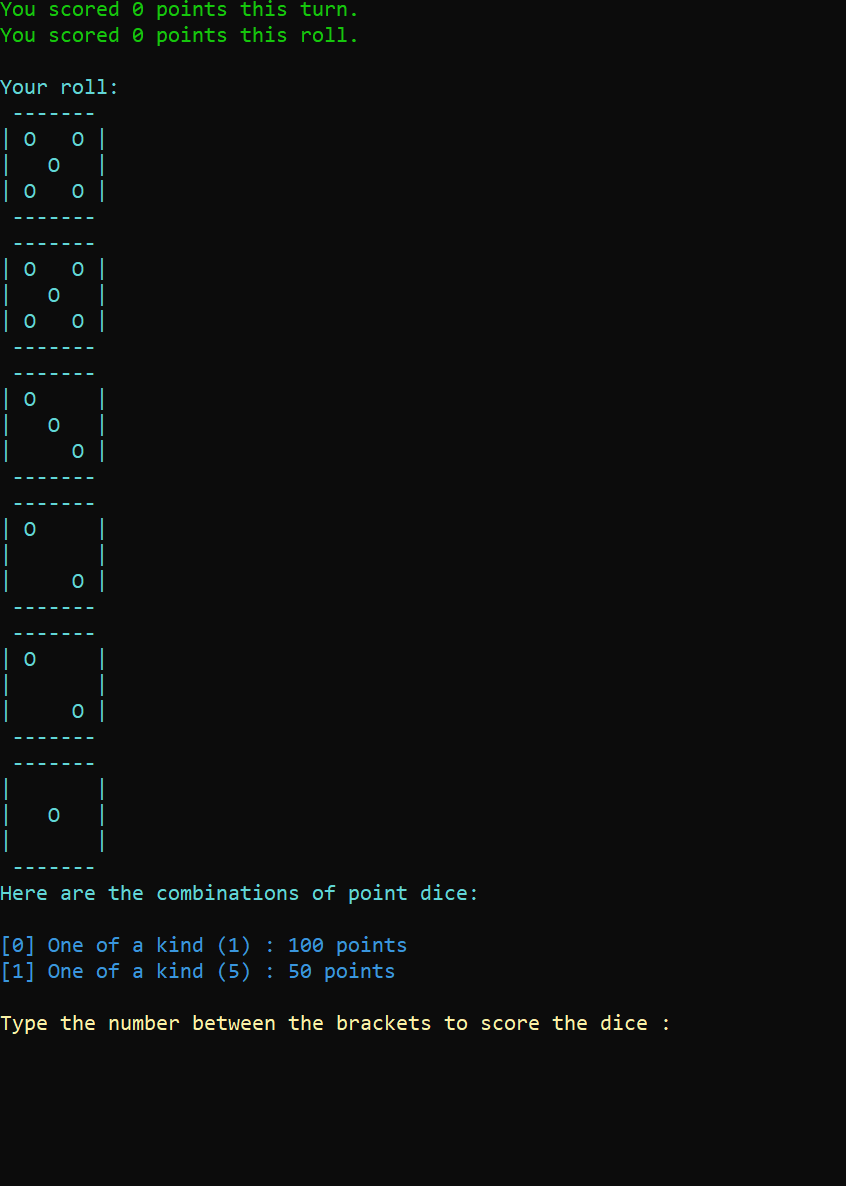


The colors helped differentiate between what kind of information was being given. I stuck to a color code: Green for related to the player, yellow for questions, blue for input options, red for something bad, turquoise and brown for special information.

Next I looked into the multiple instances of single point dice 1 and 5. I just added a check to look and see if the combination already exists in the selection or not and if it does then to skip adding it again to the selection. It was at this time that I also changed how each selection was printed to the console. Instead of printing each one when the selection is added to the list, I instead append the text to an output string. This was because the header string showing the possible combinations would show when a Farkle would occur which annoyed me. This also feels as though it cleans up the code by having only one spot where the cout is rather than multiple in the program.

| //If code is already in selection, skip **if** (selection[combinations \* 3 - 2] == DiceComboCodes::SingleOne) {  **break**; }  output.append("[" + to\_string(combinations) + "] One of a kind (" + to\_string(roll[i]) + ") : 100 points\n");  //Resize array to add new combination ResizeIntArray(selection, (combinations + 1) \* 3); selection[combinations \* 3] = combinations; //Binding selection[combinations \* 3 + 1] = DiceComboCodes::SingleOne;//Code selection[combinations \* 3 + 2] = i; //Index  combinations++; |
| --- |

Next was a joke on my friend's part, but I am a serious programmer who forgets to eat because he is so engrossed in his work - rendered dice. No I am not going to go into a game engine and make 3d renders but I can do ASCII art. I just went in and made an enum in a header file containing ASCII renditions of the dice faces. Here is a screenshot of it in the program. We can also see the single instances of the 5 showing up in the selection as well.



Last was rerolling while on the above screen. This was easy enough as I had all the framework established already. I added a code to the possible combinations, and had it print alongside the stop scoring combination that I had already implemented.

Printing and adding combination to selection:

| //If points have been scored **if** (pointsScored > 0) {  //add option to reroll dice  PrintWithColor("[" + to\_string(combinations) + "] Reroll Dice\n", Colors::brown, Colors::black);   ResizeIntArray(selection, (combinations + 1) \* 3);  selection[combinations \* 3] = combinations; //Binding  selection[combinations \* 3 + 1] = DiceComboCodes::Reroll;//Code  selection[combinations \* 3 + 2] = -1; //Index   combinations++;   //add option to stop scoring  PrintWithColor("[" + to\_string(combinations) + "] Stop Scoring\n", Colors::brown, Colors::black);   ResizeIntArray(selection, (combinations + 1) \* 3);  selection[combinations \* 3] = combinations; //Binding  selection[combinations \* 3 + 1] = DiceComboCodes::Stop;//Code  selection[combinations \* 3 + 2] = -1; //Index   combinations++; } |
| --- |

Selection cases:

| //Roll dice again **case** DiceComboCodes::Reroll:  currentScore += pointsScored;   PrintWithColor("You got " + to\_string(pointsScored) + " points this roll.\nYou now have " + to\_string(currentScore) + " points to cash.\n\n", Colors::green, Colors::black);   RollDice();  **return**; // stop scoring **case** DiceComboCodes::Stop:  input = -2;  **break**; |
| --- |

Other things of note are some minor changes made when splitting functions between classes.

Dropout was turned into RemovePlayer() on the Game class. The only issues I came across with this was later in the programs development cycle. An error would be thrown whenever the last player in the array would drop out as the program would continue to look for their name at the current turns index. Looking at the program now and what I had copied and shown earlier it looks as though I forgot to copy the NextTurn() call, or otherwise had it and it did not work as intended. Now it looks to see if currentTurn is greater than or equal to the player count and loops it back to zero.

Multiple getters were added to the Game class to allow the FarkleCore to access specific variables.

This brings the program up to its current point I’m pretty sure.

Errors:

While I did not run into too many bugs, some of which I may not remember, the ones that stood out I do remember. The main “bug” that I worked on for hours was with the logic in choosing what score dice combinations were available given the dice rolled. As mentioned it mainly consisted of logical errors that were not flagged in the compiler but gave issue nonetheless. Most of my time was spent on testing and bug fixing this error.

The other bug that stood out to me was with resizing the player array when removing a player from the array. I found that in certain cases when removing a player the program would crash and throw an error from the backend C++ code, which I had and still have no idea how to fully comprehend. Stepping through the code I found that it occurred when grabbing the players name for showing their stats when on the main menu, and it only occurred when removing the last player from the array. This led me to believe that the code was trying to access a part of the array outside the bounds. Looking through the code this turned out to be the case, and by adding an out of bounds check to the RemovePlayer() function it was fixed.

Beyond that it was just trial and error when using a function from a library I typically don’t use - such as with the text color. But I don’t really call these bugs or errors just growing pains as I learn something new.