UNO!

By Samuel Gerungan

https://github.com/Sam-T-G https://github.com/Sam-T-G/UNO

Project Summary

This project was an invaluable journey through realizing the programming techniques we've learned in class. The process of creating a working and fully functioning project was a very effective way to learn the basics of programming by actively performing repetitions of the basic workflow we will do as programmers.

There were very important lessons to be learned by failing numerous amounts of times to create features that are seemingly simple, let alone making all of the components be cohesive and responsive in one running program. When programming in C++ it's imperative to be very sequential in our logic, as the way we as humans process information is all encompassing when it comes to the scope of our prior experiences and knowledge, whereas creating a computer program needs careful accuracy of declaring all the necessary variables and pieces to perform the desired functionality.

Below I document my journey in real time as to how the development process unfolded taking a chronological and journalistic approach to my writeup as I think it will be valuable to log my

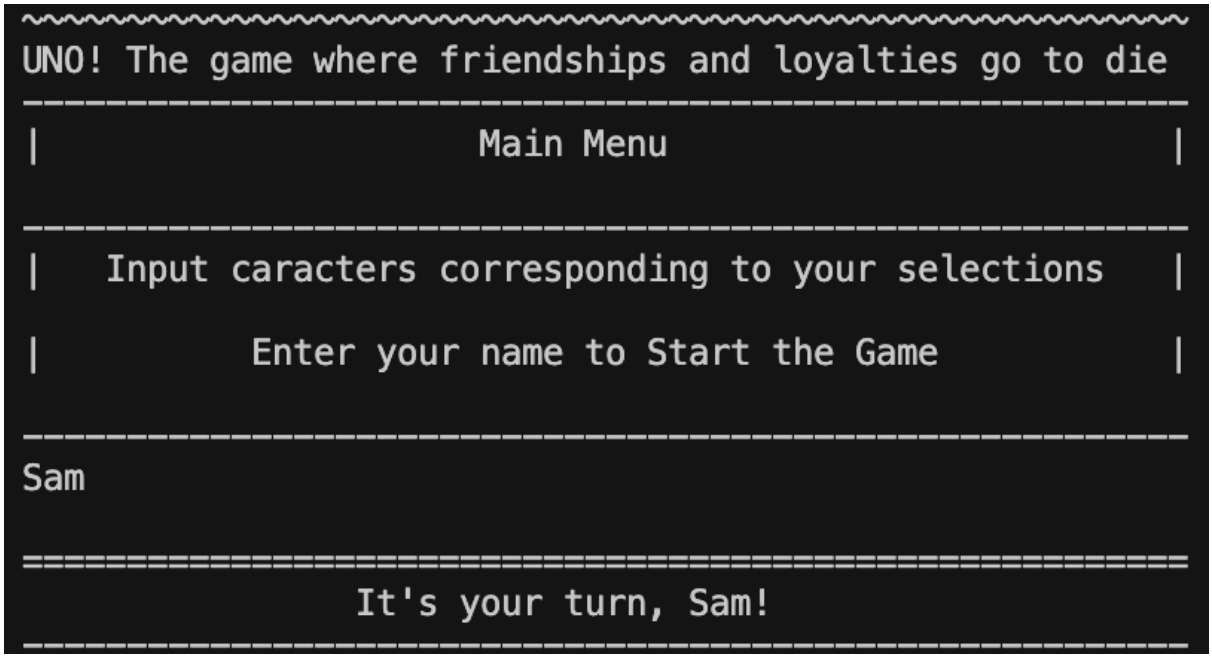
process in brainstorming, learning, testing, adjusting approach and strategies, and proper implementation of the features desired in my UNO game.

Table of Contents

Proof of Concept for Most Recent Version

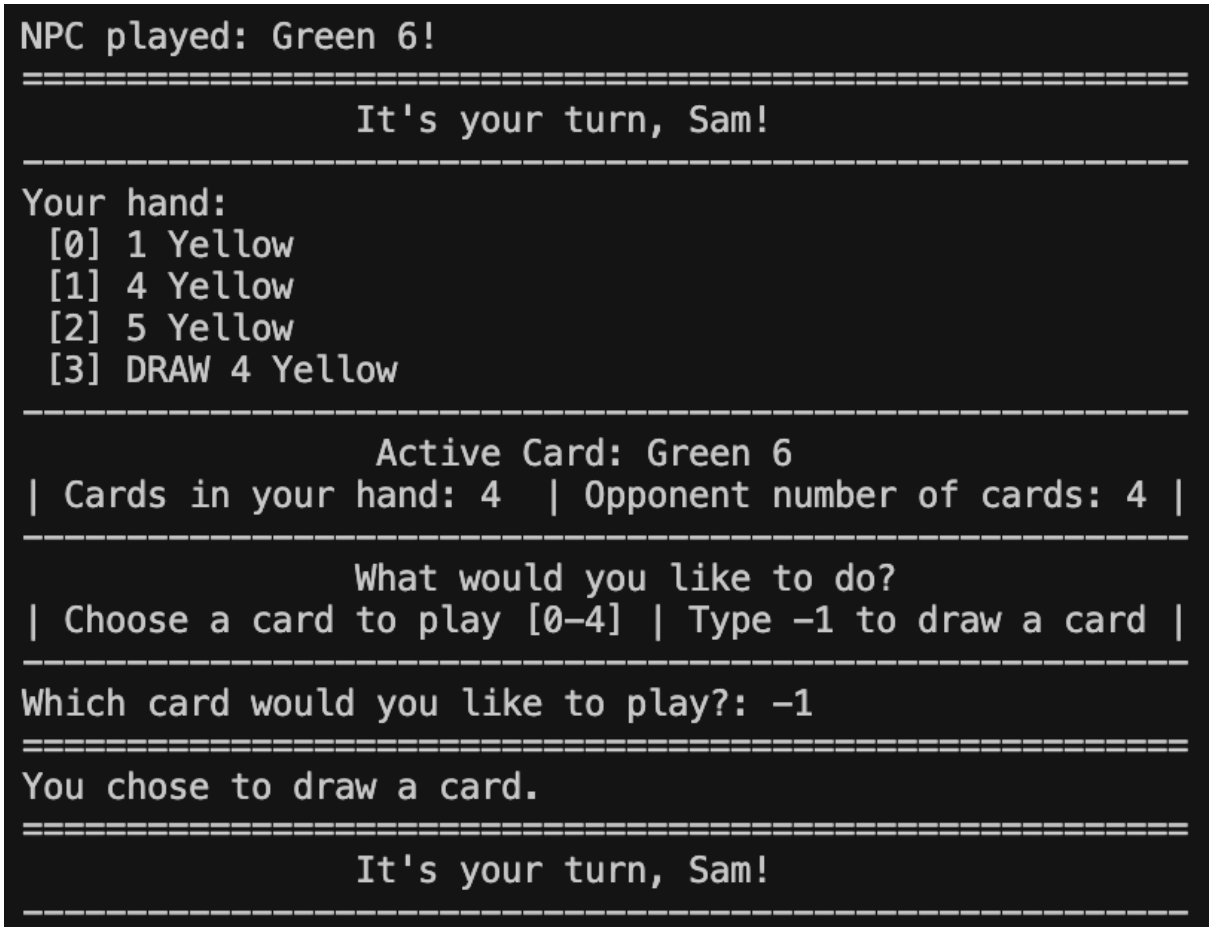
Here is a collection of screenshots that contain snippets of a full gameplay loop and show all the working functions of the UNO game.

> When the game is compiled and executed, the main menu prompts the user to enter their name to start the game.



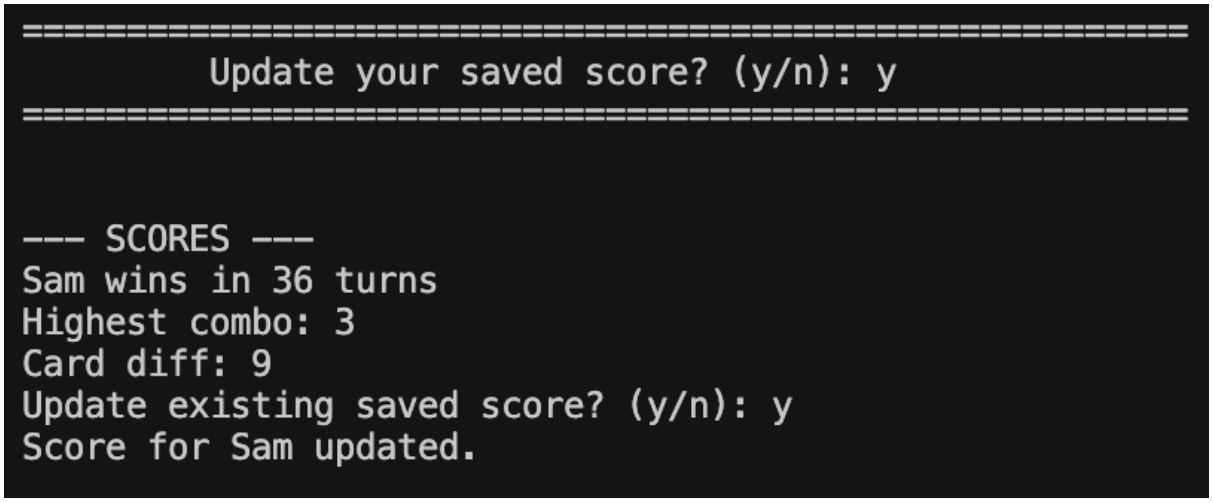
> The user and NPC are then dealt a hand of cards at the beginning of the game, and we are then shown the main user interface with all the appropriate information for the user to play during their turn.
>
> The user is shown whose turn it is, the cards in their hands, the active card in play as well as the value, and the amount of cards present in both players' hands.
>
> The user is then prompted with instructions on the moves they can perform.

> Below, we observe the prompt given when the NPC plays an appropriate card.
>
> We then are shown the cards available in hand, which are in descending sorted order, and all the appropriate information.
>
> We also display that the draw functionality behaves just as the real UNO game does, in which it passes the turn back to the player that drew the card.



> The following shows that the NPC plays within its best interest as well, in which it will play a card if possible, draw a card if they have no playable card, and intelligence programmed in which it can choose an alternate color to swap to when playing a WILD card.

> The following shows the victory sequence for the player and the game's score saving feature to output scores to a separate binary file.



> The following also shows that the program is able to read the binary file and show the score history of previous victories.

Project Phase 3 Development

> Implementing structures, memory allocation, and various other concepts

The goal in phase three of development is to further incorporate technologies that will streamline the effectiveness, functionality, and scalability of our UNO! game.

A major goal for this phase is to completely restructure the program's means of processing cards from using a two dimensional array to using structures. Expanding this goal will be a complete upheaval in the way our program stores and processes the game's information, but the functionality will largely be similar. This will completely re-imagine the current indexing system to be more dynamic, with the capability to process and display card quality and quantity much more efficiently and modularly.

Along the major program rewrite, we'll strive to also revamp the player UI to be more concise since we will not be printing out the entirety of the matrix included in the previous versions.

Version 3.1

Goals for UNO Version 3.1

For version 1, we will strive to create replicate the main game interface that exists within prior versions, and establish the new card and player indexing through structures.

Reorganizing Repository

Upon initial initialization of phase 3, I've come to the realization that he version indexing of the UNO! project is due for a revamp in order to more efficiently organize the development progress.

Past versions shall be renumbered to a main index to signify general version phase, and individual version iterations indexed in the following to indicate more minor release updates.

The indexing will reflect the following.

Phase 1:

- V 1.1
- V 1.2
- V 1.3
- V 1.4

Phase 2:

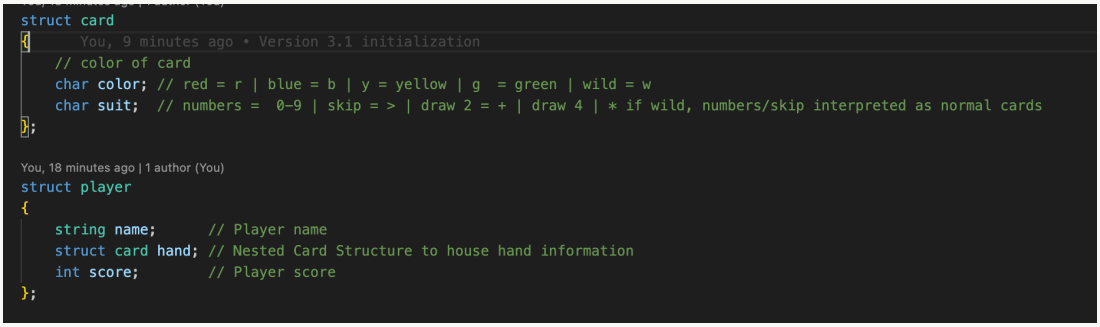
- V 2.1
- V 2.2
- V 2.3

Phase 3:

- V 3.1 (Current)

Pseudocode, Migration, and Restructuring

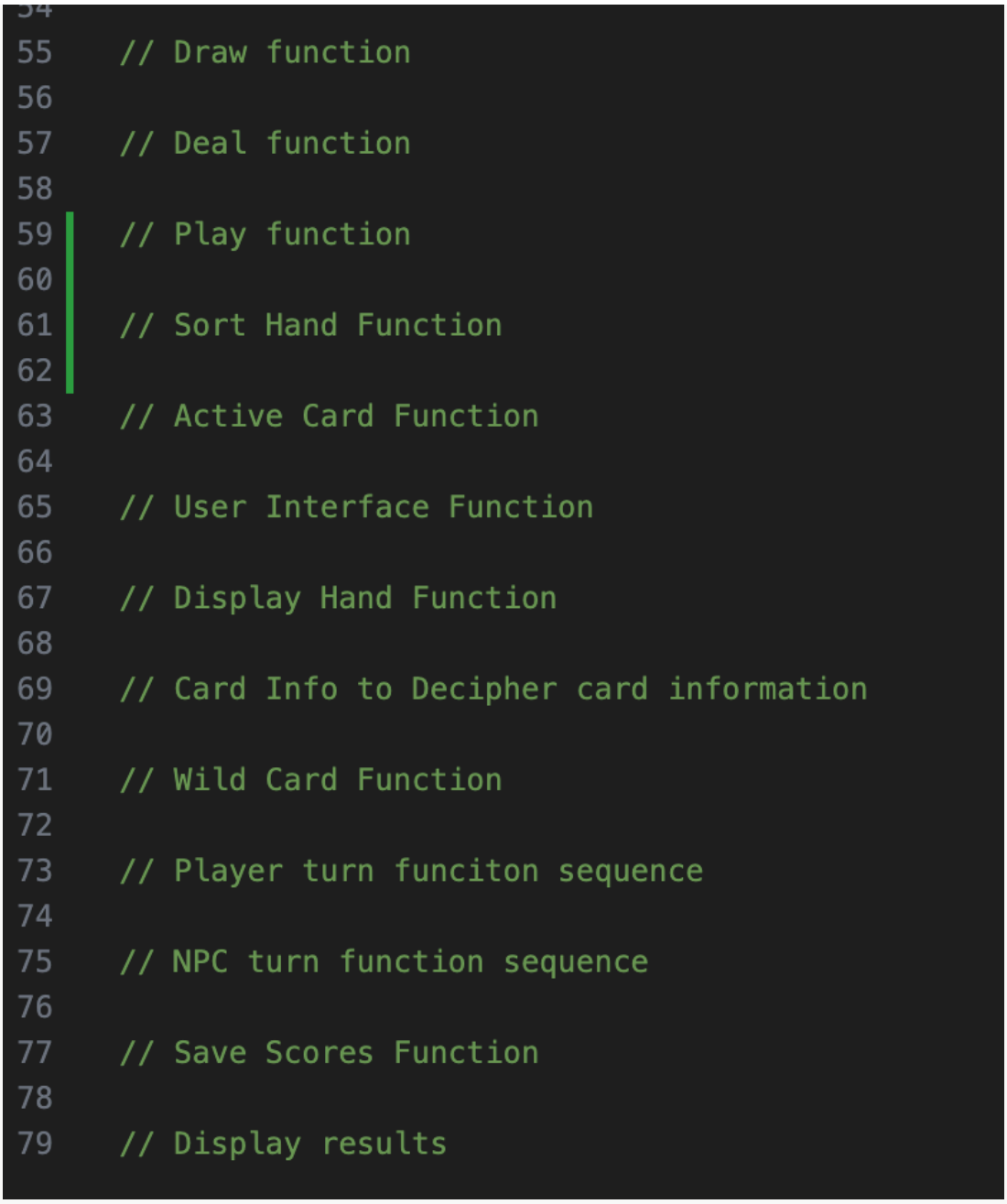
To start off version 3.1, we begin with organizing important features from prior iterations and re-imagining the structure of how the game should function.



We begin with the first two structures needed for the transition for our game. Cards will now be created utilizing the new card structures as objects that we will then nest within the player structure we've created. This small iteration also enables us to instantiate multiple playable characters within our game.

Furthermore, we can use the player structure to hold information for both human and computer players.
This will be implemented in a future iteration to maintain our goals of reaching our minimum viable product.

Finally, we create pseudocode in order to outline the general functions we need in order to create a functioning UNO! game as previous versions have established.



These general functions have been migrated but need to be re-structured and re-written in order to function within our new data structuring utilizing structures for both card and player organization.

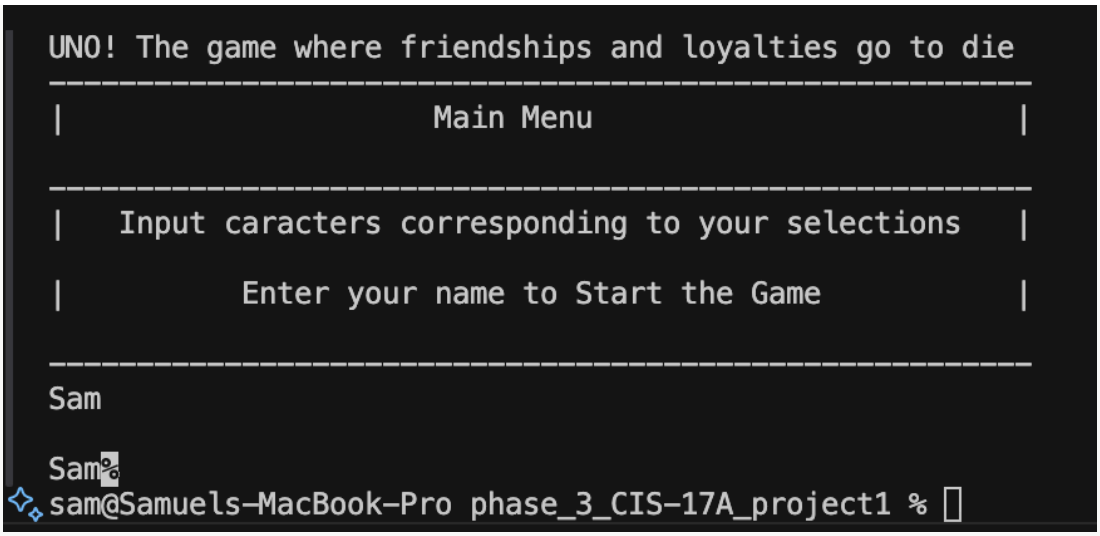
Structure Setup and Menu Migration

We'll start sequentially in our game restructure, starting with creating a main menu function which we modularly call the main menu for future purposes. We will then instantiate our first player object using the player structure and use the previously created main menu user name sequence in order to start our game off.

```
    // Map the inputs and outputs - Process
    Player player1; // Create a player 1 structure to hold player's information - later can be modularized
    menu(player1);  // pass player 1 structure into function
    cout << player1.name;
```

```
// Main menu function
void menu(Player &player1)
{
    for (int i = 0; i < 56; i++)
    {
        cout << "~";
    }
    cout << endl
         << "UNO! The game where friendships and loyalties go to die" << endl;
    for (int i = 0; i < 56; i++)
    {
        cout << "-";
    }
    cout << endl;
    cout << "|" << setw(30) << "Main Menu" << setw(25) << "|" << endl
         << endl;
    for (int i = 0; i < 56; i++)
    {
        cout << "-";
    }
    cout << endl
         << "|" << setw(3) << " " << "Input caracters corresponding to your selections" << setw(4) << "|" << endl
         << endl;
    cout << "|" << setw(10) << " " << "Enter your name to Start the Game" << setw(11) << " " << "|" << endl
         << endl;
    for (int i = 0; i < 56; i++)
    {
        cout << "-";
    }
    cout << endl;
    cin >> player1.name;
    cout << endl;
};      You, 57 seconds ago • Uncommitted changes
```

From this first step, we can test the output and ensure we are on the right track.



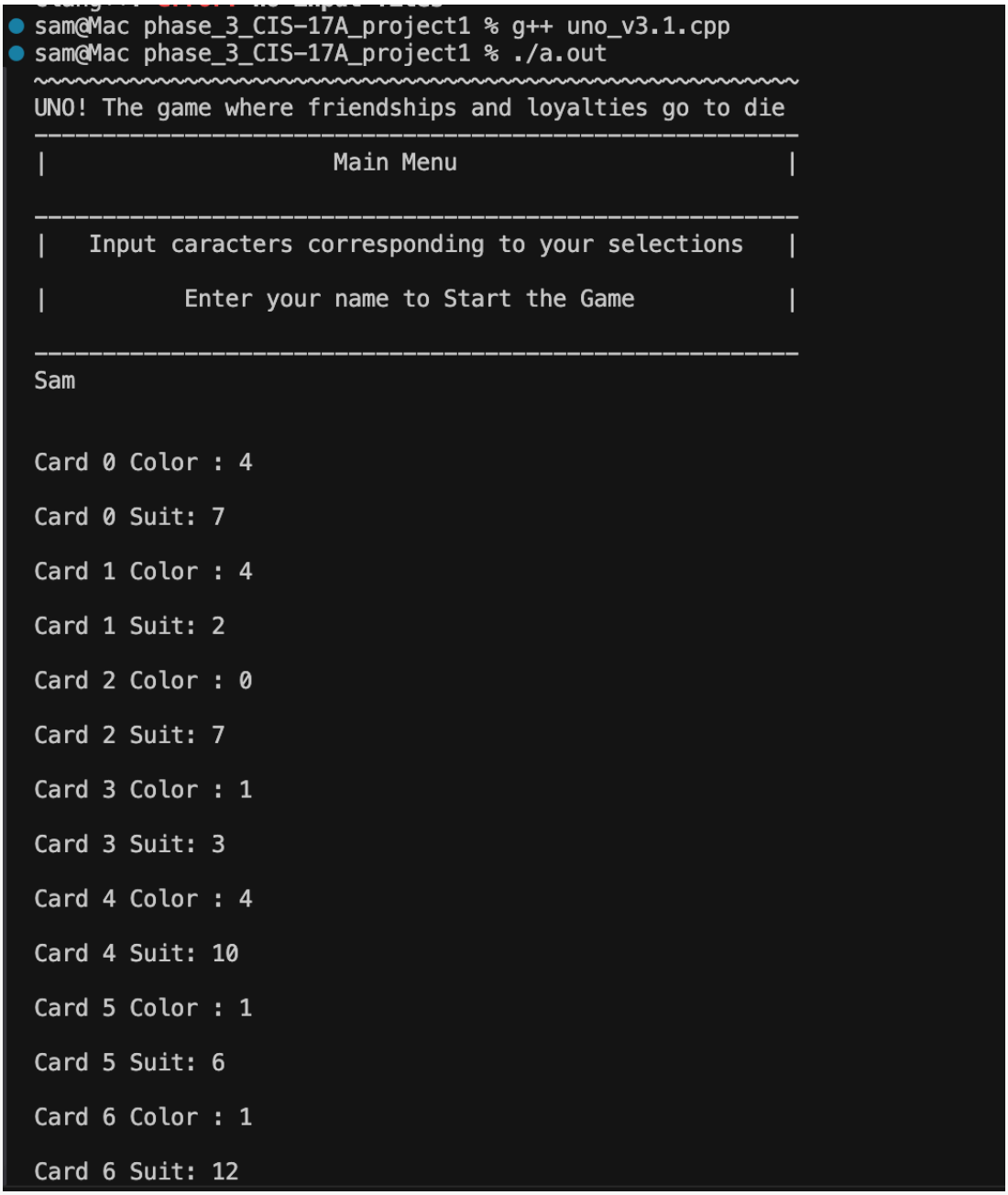
We now begin brainstorming how we want to organize the information that the game will utilize. To begin, we create a card structure that we assign a color and suit, then create an index to identify color and suit respectively. We'll also use enumerated data types to increase readability.

```cpp
enum CardColor
{
    RED,
    BLUE,
    YELLOW,
    GREEN,
    WILD
};

enum CardSuit
{
    ZERO,
    ONE,
    TWO,
    THREE,
    FOUR,
    FIVE,
    SIX,
    SEVEN,
    EIGHT,
    NINE,
    SKIP,
    DRAW_TWO,
    DRAW_FOUR
};
You, 53 seconds ago | 1 author (You)
struct Card
{
    // color of card
    CardColor color; // red = 0 | blue = 1 | 2 = yellow | 3  = green | wild = 4
    CardSuit suit;   // numbers =  0-9 | skip = 10 | draw 2 = 11 | draw 4 = 12 | * if wild, numbers/skip interpreted as normal cards
};

You, 53 seconds ago | 1 author (You)
struct Scores
{
    int numTrns; // Number of turns
    int hiCombo; // integer value to store highest number of combo
};

You, 54 seconds ago | 1 author (You)
struct Player
{
    string name;           // Player name
    vector<Card> hand;     // Nested Card Vector to house hand contents
    struct Scores plyrScr; // nested structure to store scores
};
```

Finally, we adjust our card drawing process such that a function will use the card structure to randomly generate a card and return the structure back.

```
// Draw function
Card draw()
{
    Card newCrd;
    newCrd.color = static_cast<CardColor>(rand() % 5);
    newCrd.suit = static_cast<CardSuit>(rand() % 13);
    return newCrd;
}

// Deal function
void deal(Player &p1, Player &npc)
{ // Deal 7 Starting cards for player and npc
    for (int i = 0; i < 7; i++)
    {
        p1.hand.push_back(draw());
        npc.hand.push_back(draw());
    }
};
```

We then create a simple deal function to handle the initial dealing of cards to both the player and the npc hands. When printing the contents, it should look like the following:


sam@Mac phase_3_CIS-17A_project1 % g++ uno_v3.1.cpp
sam@Mac phase_3_CIS-17A_project1 % ./a.out
UNO! The game where friendships and loyalties go to die
| Main Menu |
| Input caracters corresponding to your selections |
| Enter your name to Start the Game |
Sam
Card 0 Color : 4
Card 0 Suit: 7
Card 1 Color : 4
Card 1 Suit: 2
Card 2 Color : 0
Card 2 Suit: 7
Card 3 Color : 1
Card 3 Suit: 3
Card 4 Color : 4
Card 4 Suit: 10
Card 5 Color : 1
Card 5 Suit: 6
Card 6 Color : 1
Card 6 Suit: 12

Version 3.2

Goals for UNO Version 3.2

For version 2, we will strive to migrate the core of the game logic now that we've established the main game structure.

This next leg of development is the main chunk of progress to be made, albeit most of the logs has been created from the prior version 2.x models. I perceive the difficulty in this next segment to primarily lie in the re-imagining of the functions we've already written to interact with structures rather than a two dimensional array. In many of these cases, this method is a lot faster since the multiple steps of logic needed in order to read and manipulate cards in play for both the player and the NPC do not need to be processed through multiple nested loops and through the constant searching of a two dimensional array. We begin with migrating the basic functionality of card display, UI display and prompts, as well as turn handling.

To start out, we migrate the base logic for playing a wild card and handling color swapping via wild play.

```
// Wild Card Function
void wildCrd(Card &card)
{
    if (card.color == WILD)
    {
        int newClr;
        cout << "Choose a new color (0: RED, 1: BLUE, 2: YELLOW, 3: GREEN): ";
        cin >> newClr;
        card.color = static_cast<CardClr>(newClr);
    }
}
```

This was actually much simpler to execute than the original method since we simply change the color variable within our card structure to represent the new chosen color.

Next we create a modular function to display and translate our enum integer values into readable human strings by simply storing the potential values in arrays as a reference. We then create the edge case for wild cards and also a translation print for regular cards. Just to be safe, we have a failsafe in case the card information happens to be incorrect.

```
// Active Card Display Function
void dispCrd(Card &actvCrd)
{
    // String arrays for descriptive output
    string colors[] = {"Red", "Blue", "Yellow", "Green", "Wild"};
    string values[] = {"0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "SKIP", "DRAW 2", "DRAW 4"};

    if (actvCrd.color == WILD)
    {       You, 52 minutes ago • Logic Migration and Adaptation
        cout << "Wild" << endl;
    }
    else if (actvCrd.color >= 0 && actvCrd.color < 5 && actvCrd.suit >= 0 && actvCrd.suit < 13)
    {
        cout << values[actvCrd.suit] << " " << colors[actvCrd.color] << endl;
    }
    else
    {
        cout << "Invalid Card" << endl;
    }
}
```

Next we create the base overarching logic in main to handle the turn and win conditions. This was also much simpler to write because we can modularize a lot of the logic to take place in separate functions for the NPC and the Player.

```
bool turn = true; // Player starts first

// Display and output the results
while (!p1->hand.empty() && !npc->hand.empty())
{
    if (turn == true) // Player's turn
    {
        plyrTrn(*p1, *npc, actvCrd, turn);      You, 57 minu
    }
    else if (turn == false) // NPC's turn
    {
        npcTrn(*p1, *npc, actvCrd, turn);
    }
}

if (p1->hand.empty())
{
    cout << "You win!" << endl;
}
else
{
    cout << "NPC wins!" << endl;
}
```

The entirety of the main turn handling logic is shown above. We create a boolean to track whether or not it's the player's turn, then pass pointer variables to both npc and player turns so we can manipulate card draws for the opponents when playing draw cards. We then exit the while loop if either player's hands are empty and declare the winner.

Now we begin to modify our original turn handling functions, starting with the player.

```
// Player turn funciton sequence
void plyrTrn(Player &p1, Player &npc, Card &actvCrd, bool &turn)
{
    int choice;

    while (turn == true)
    {
        turn = false;             // Default set turn to false at the start of the loop
        usrInt(p1, npc, actvCrd); // Display the current game state
        cin >> choice;

        if (cin.fail())
        {
            cin.clear();
            cin.ignore(1000, '\n');
            cout << "Invalid input. Try again." << endl;
            turn = true;
        }
        else if (choice == -1)
        {
            cout << "You chose to draw a card." << endl;
            p1.hand.push_back(draw());
            turn = true; // Player goes again after drawing a card
        }
        else if (choice < 0 || choice >= static_cast<int>(p1.hand.size()))
        {
            cout << "Invalid choice. Pick a valid card index or -1 to draw." << endl;
            turn = true; // Allow another turn if the choice is invalid
        }
        else
        {
            play(p1, npc, choice, actvCrd, turn);
        }
    }
}
```

We migrate a majority of the logic, but adjust it so that we do not need to hold a variable to take inventory of the player hand count and also the process of drawing cards is simpler in that we utilize our draw function to generate a new card which we push to the end of the vector.

We also chose to compartmentalize the play function as follows:

```
// Play card function
void play(Player &p1, Player &npc, int choice, Card &actvCrd, bool &turn)
{
    // store selected card
    Card slctd = p1.hand[choice];

    // Error check for card range chosen
    if (choice < 0 || choice >= p1.hand.size())
    {
        cout << "Invalid choice!" << endl;
        return;
    }
```

Above, we've imported some variables created in the player function, as well as create a selected card copy in which we store the information that the player chooses.

We then create the main validation segment in which we compare the selected card color and suit such that playing a match of either will validate the card played. This segment is also vastly different than that of our 2-D array matrix indexing from the prior versions in that we simply erase the card once validated, then condense the array using a combination of the erase and begin functions.

```
// Validate play: same color, same suit (number/action), or wild
if (slctd.color == actvCrd.color || slctd.suit == actvCrd.suit || slctd.color == WILD)
{
    actvCrd = slctd;                          // Update active card
    p1.hand.erase(p1.hand.begin() + choice); // Remove played card
    cout << "You've played a ";
    dispCrd(actvCrd); // Display active card in human-readable format

    // Handle special cards
    if (slctd.suit == 10) // SKIP
    {
        cout << "SKIP played! It's your turn again!" << endl;
        turn = true; // Player goes again
    }
    else if (slctd.suit == 11) // DRAW 2
    {
        cout << "DRAW 2 played! Opponent draws 2 cards!" << endl;
        npc.hand.push_back(draw()); // draw two cards
        npc.hand.push_back(draw());
        cout << "Opponent now has " << npc.hand.size() << " cards!" << endl;
        turn = true; // Player goes again
    }
    else if (slctd.suit == 12) // DRAW 4 (if added)
    {
        cout << "DRAW 4 played! Opponent draws 4 cards!" << endl;
        for (int i = 0; i < 4; i++) // loop to process 4 card draw
        {
            npc.hand.push_back(draw());
        }
        cout << "Opponent now has " << npc.hand.size() << " cards!" << endl;
        turn = true; // Player goes again
    }

    // Handle color choice if Wild
    if (slctd.color == WILD)
    {
        wildCrd(actvCrd); // call wild card function
    }
}
else
{
    cout << "Invalid play: Card does not match active card by color or number!" << endl;
}      You, 1 hour ago • Logic Migration and Adaptation
}
```

As we can observe above, we also can simply create much more efficient logic than having to traverse through a 2-D matrix as was done previously and we simply draw cards and push to the end of the array of either the player or the NPC.

```
while (!valid) // Check if valid play has been made
{
    if (i < npc.hand.size())
    {
        Card c = npc.hand[i]; // Make a copy of card at given index

        if (c.color == actvCrd.color || c.suit == actvCrd.suit || c.color == WILD)
        {
            actvCrd = c;
            npc.hand.erase(npc.hand.begin() + i);

            cout << "NPC played: " << crdInfo(actvCrd) << "!" << endl;

            // Default: player's turn next
            turn = true;

            if (c.color == WILD)
            {
                CardClr newClr = static_cast<CardClr>(rand() % 4);
                actvCrd.color = newClr;
                string colors[] = {"Red", "Blue", "Yellow", "Green"};
                cout << "NPC plays a WILD and chooses " << colors[newClr] << "!" << endl;
            }

            // Handle special cards
            if (c.suit == SKIP)
            {
                cout << "NPC played SKIP! You lose a turn." << endl;
                turn = false;
            }
            else if (c.suit == DRAW_TWO)
            {
                cout << "NPC played DRAW 2! You draw 2 cards." << endl;
                for (int i = 0; i < 2; ++i)
                    p1.hand.push_back(draw());
                turn = false;
            }
            else if (c.suit == DRAW_FOUR)
            {
                cout << "NPC played DRAW 4! You draw 4 cards." << endl;
                for (int i = 0; i < 4; ++i)
                    p1.hand.push_back(draw());
                turn = false;
            }

            valid = true; // Set Valid to true if valid play has been made
        }
```

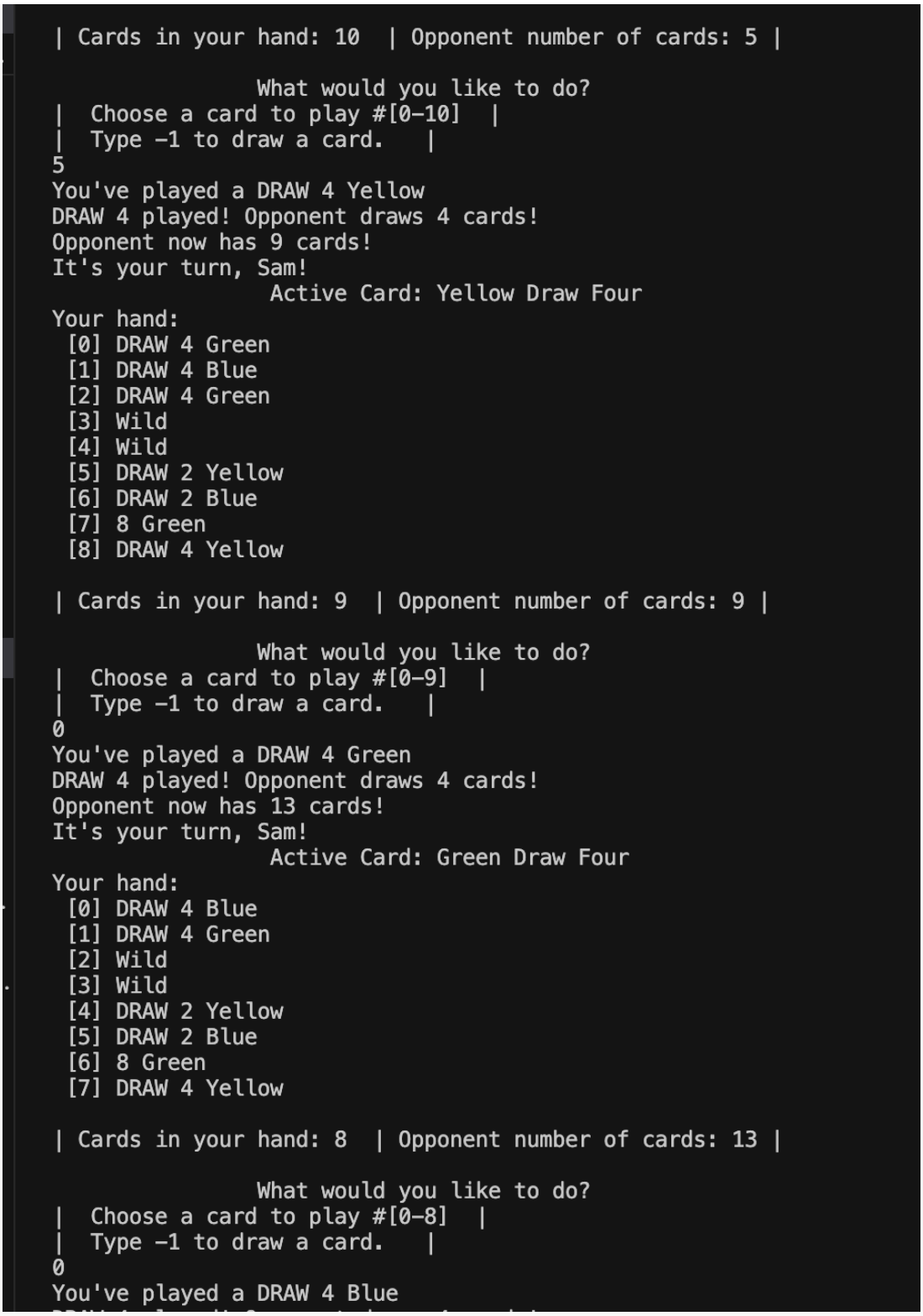
Something to note in the snippet above, we are able to massively cut back on the opponent AI logic in that we can reduce the external function into a single line in which the NPC can directly manipulate the active color in a single line of code.

```
else
{
    Card drawn = draw();
    npc.hand.push_back(drawn);
    cout << "NPC draws a card!" << endl;

    if (drawn.color == actvCrd.color || drawn.suit == actvCrd.suit || drawn.color == WILD)
    {
        actvCrd = drawn;
        npc.hand.pop_back(); // play the drawn card
        cout << "NPC plays the drawn card: " << crdInfo(actvCrd) << "!" << endl;

        if (drawn.color == WILD)
        {
            CardClr newClr = static_cast<CardClr>(rand() % 4);
            actvCrd.color = newClr;
            string colors[] = {"Red", "Blue", "Yellow", "Green"};
            cout << "NPC chooses " << colors[newClr] << "!" << endl;
        }

        // If it's a special card, handle turn
        if (drawn.suit == SKIP || drawn.suit == DRAW_TWO || drawn.suit == DRAW_FOUR)
        {
            if (drawn.suit == SKIP)
                cout << "NPC played SKIP! You lose a turn." << endl;
            else if (drawn.suit == DRAW_TWO)
            {
                cout << "NPC played DRAW 2! You draw 2 cards." << endl;
                for (int i = 0; i < 2; ++i)      You, 41 minutes ago • Base Functionality an
                    p1.hand.push_back(draw());
            }
            else if (drawn.suit == DRAW_FOUR)
            {
                cout << "NPC played DRAW 4! You draw 4 cards." << endl;
                for (int i = 0; i < 4; ++i)
                    p1.hand.push_back(draw());
            }

            turn = false; // NPC goes again
        }
        else
        {
            turn = true; // Player's turn
        }

        valid = true;
    }
```

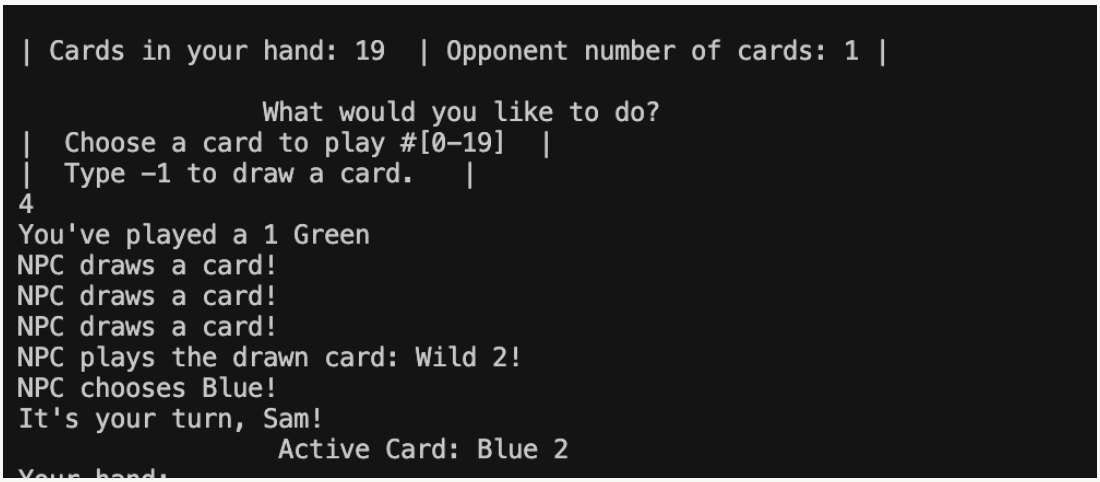
All other logic is similar to the player play verification and organization functions.

The last task to wrap up this version is to test a play-through of the game and the expected interactions.

Matching card color as well as number:



Above we demonstrate turn handling pertaining to skipping opponent turns when playing skip cards, we demonstrate matching suit to adjust active color, and we demonstrate proper hand content number display.



Above we also demonstrate the NPC logic in action. When the NPC does not have a playable card, the NPC will opt to draw a card until acquiring a playable card. We also demonstrate the NPC's capability to play a wild card and choose a color to set as the new active color.

```
NPC plays a WILD and chooses Blue!
It's your turn, Sam!
                 Active Card: Blue 9
Your hand:
 [0] Wild
 [1] 8 Green

| Cards in your hand: 2  | Opponent number of cards: 28 |

               What would you like to do?
|  Choose a card to play #[0-2]  |
|  Type -1 to draw a card.   |
0
You've played a Wild
Choose a new color (0: RED, 1: BLUE, 2: YELLOW, 3: GREEN): 3
NPC played: Green 9!
It's your turn, Sam!
                 Active Card: Green 9
Your hand:
 [0] 8 Green

| Cards in your hand: 1  | Opponent number of cards: 27 |

               What would you like to do?
|  Choose a card to play #[0-1]  |
|  Type -1 to draw a card.   |
0
You've played a 8 Green
You win!
sam@Mac phase_3_CIS-17A_project1 %
```

Finally we observe the final sequence in a play-through in which the player wins and the game congratulates the player for the win.

> UNO Version 3.2 has now been deployed to GitHub at
>
> https://github.com/Sam-T-G/UNO/releases/tag/v3.2

https://github.com/Sam-T-G/UNO/releases/tag/v3.2

Version 3.3

Goals for UNO Version 3.3

For the final version in the 3.X series, we will focus on creating a new scoring system we will calculate within the scores structure.

For scores, we will track two different metrics - amount of turns taken to win, and also introduce the high combo metric, which will keep track of the highest amount of turns any player will sequentially perform.

We will then export the scores on to an external binary file to store high scores, and furthermore, make finishing touches to improve game UI.

Our first priority will be to implement a new scoring system that will keep track of three metrics:

- Difference of card amount between player and NPC hands
- Amount of turns needed to win
- Highest combo chain performed against opponent

```
if (p1->hand.empty())
{
    cout << "You win!" << endl;
    calcScrs(*p1, *npc); // Save only if player wins
}
else // NPC wins and score not saved
{
    cout << "NPC wins!" << endl;
}

readScrs(); // View history of past game scores
```

First, we'll implement a calculate scores function and a read scores function that will be called at the end of the main function once a winner has been declared. The calculate scores will only write to the binary scores sheet if the player wins, and simply prompts that the NPC is victorious otherwise.

We then include a universal read scores function to read the binary file that has been created to store high scores.

The victory prompt, the score writing process, and the reading of prior scores is displayed below:

```
You've played a 9 Yellow
You win!

--- SCORES ---
Sam wins in 11 turns
Highest combo: 3
Card diff: 8

=== SCORE HISTORY ===
Record 1:
  Player  : Sam
  Turns   : 11
  HiCombo : 3

sam@Samuels-MacBook-Pro phase_3_CIS-17A_project1 %
```

Implemented a feature to sort the player's hand for a better user experience.

```
// Sort Hand Function
void srtHnd(Player &p1)
{
    sort(p1.hand.begin(), p1.hand.end(), [](const Card &a, const Card &b)
        {
        if (a.color == b.color) {
            return a.suit < b.suit;
        }
        return a.color < b.color; });
}
```

In practice, the new user hand is organized as displayed below:

```
It's your turn, Sam!
                Active Card: Yellow 8
Your hand:
 [0] 4 Red
 [1] 5 Red
 [2] 6 Red
 [3] SKIP Red
 [4] 3 Blue
 [5] SKIP Blue
 [6] 9 Yellow
 [7] 9 Yellow
 [8] Wild

| Cards in your hand: 9  | Opponent number of cards: 2 |
```

The sort order organizes by card color, and card numbers within each respective color in ascending order of value.

UI Cleanup and Tweaks

Added a small dialogue to improve input clarity and spacing.

```
turn = false;             // Default set turn to false at the start of the loop
usrInt(p1, npc, actvCrd); // Display the current game state
cout << "Make a move: ";  // Prompt for visual Clarity
cin >> choice;            // Input player choice
```

Before:
The “0” line is the user input which is very hard to discern among the other text existing.

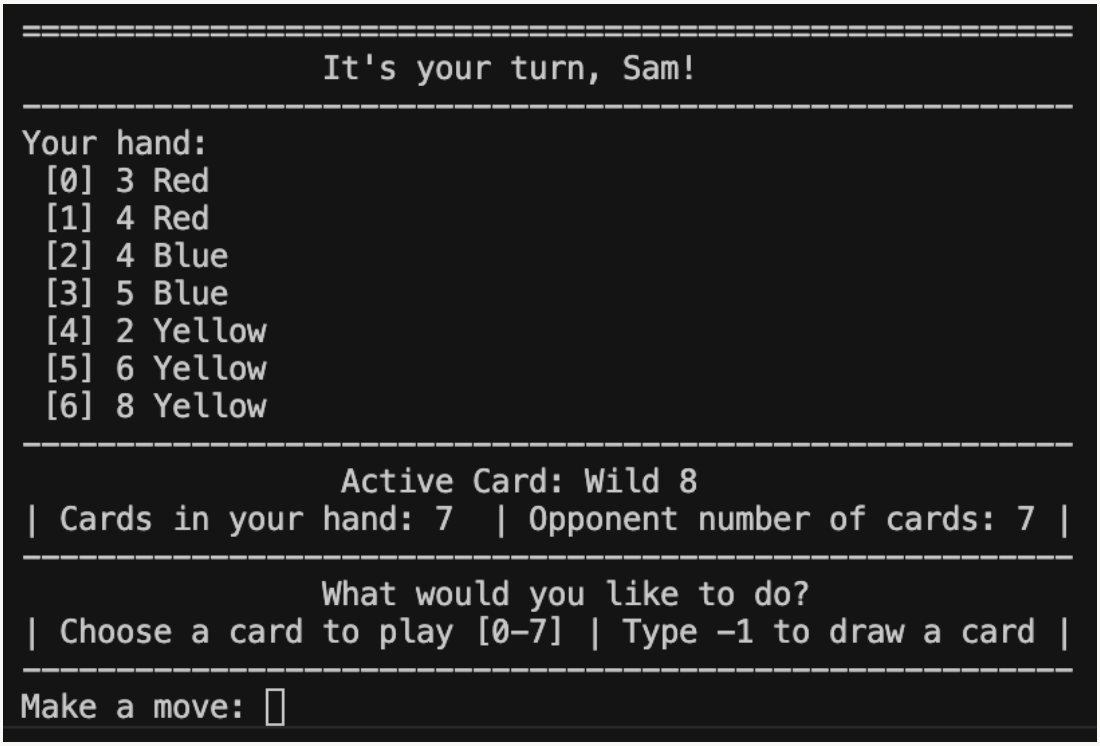
```
|  Choose a card to play #[0-7]  |
|  Type -1 to draw a card.   |
0
You've played a 5 Red
NPC played: Red 3!
It's your turn, Sam!
                    Active Card: Red 3
```

After:

```
| Cards in your hand: 7  | Opponent number of cards: 7 |

               What would you like to do?
|  Choose a card to play #[0-7]  |
|  Type -1 to draw a card.   |
Make a move: 2
You've played a 6 Blue
NPC played: Wild 4!
NPC plays a WILD and chooses Green!
It's your turn, Sam!
```

Further clarity improvements:

And even more UI formatting for a better user experience - featuring borders to compartmentalize data.



We also update the wild card play function so that it has better readability, and also now takes in either uppercase or lowercase character inputs to represent the color the user wants to swap to.

The program is structured as follows:

```
void wildCrd(Card &card)
{
    if (card.color == WILD)
    {
        char newClr;
        bool valid = false;

        while (!valid)
        {
            cout << "------------------------------------------------------" << endl
                 << setw(18) << " " << "Choose a new color!" << endl
                 << setw(6) << " " << "R = Red, B = Blue, Y = Yellow, G = Green" << endl
                 << "------------------------------------------------------" << endl
                 << "New Color: ";
            cin >>
                newClr;
            newClr = toupper(newClr); // Handle lowercase input
            cout << "------------------------------------------------------" << endl;

            switch (newClr)
            {
            case 'R':
                card.color = RED;
                cout << "You selected: Red" << endl;
                valid = true;
                break;
            case 'B':
                card.color = BLUE;
                cout << "You selected: Blue" << endl;
                valid = true;
                break;
            case 'Y':
                card.color = YELLOW;
                cout << "You selected: Yellow" << endl;
                valid = true;
                break;
            case 'G':
                card.color = GREEN;
                cout << "You selected: Green" << endl;
                valid = true;
                break;
            default:
                cout << "Invalid color selection. Please try again." << endl;
            }
            cout << "======================================================" << endl;
        }
    }
}
```

Note: There is an invalid color selection in which the user is prompted to make another selection. An example is shown in the prior screenshot above.

Fixed a bug in which when the user selects a mis-matched card or color, the player's turn ends.

```
else
{
    cout << "Invalid play: Card does not match active card by color or number!" << endl;
    turn = true; // Let the player try again
}
```

This is simply done above by flipping the user turn back to true in the final else statement.

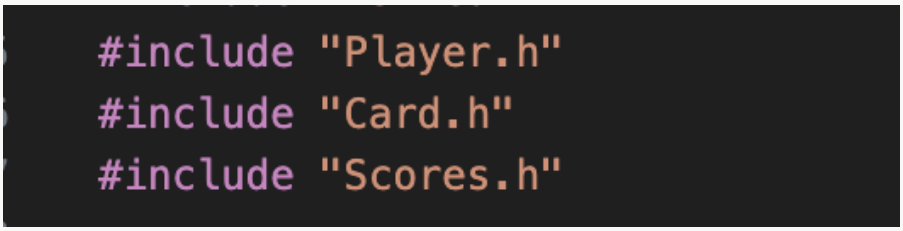
Fixed another bug where NPC wins off drawn card, and plays a special card to finish the game.

```
// Edge case for when NPC wins with last drawn card
if (npc.hand.empty())
{
    cout << "NPC plays the final drawn card and wins!" << endl;
    return;
}
```

Along with the above, also implemented edge case fix for a bug that occurs when the player wins off of playing a wild card.

```
actvCrd = slctd;                          // Update active card
p1.hand.erase(p1.hand.begin() + choice); // Remove played card
cout << setw(16) << " " << "You've played a ";
dispCrd(actvCrd); // Display active card in human-readable format

if (p1.hand.empty())
{
    cout << setw(16) << " " << "You've played your last card!" << endl;
    return; // Exit early — game ends after this play
}
```

The next step, we strive to clean up the main code by compartmentalizing various structures and player functions to external files. First, we've implemented external files to house structures with their respective file names.



Next, we've modularized the main player functionality to an external Player.cpp file.

```cpp
#include "Player.h"
#include "Card.h"

extern Card draw();

void Player::rstCmb()
{
    cmb = 0;
}

void Player::updCmb()
{
    cmb++;
    if (cmb > cmbMx)
        cmbMx = cmb;
}

void Player::drwCrd()
{
    hand.push_back(draw());
}

int Player::hndSze() const
{
    return hand.size();
}
```

Here, we handle the combo counter and also implement a reset combo function, we handle the draw card function, and a function to return the current hand size.

We then fix a bug and simultaneously utilize our new player functions of drwCrd() and rstCmb (draw card and reset combo). The bug occurred in the way the program handled combo counter resets - it never reset when the player chose to draw a card.

```
else if (choice == -1)
{
    cout << "You chose to draw a card." << endl;
    p1.drwCrd();
    p1.rstCmb(); // Reset combo streak
    turn = true; // Player goes again after drawing a card
}
```

The scores now accurately reflect the proper metrics and combo resets.

Record 1 shows the bugged hi-combo play-through, and record 2 now shows the proper combo amounts.

```
You win!
Update your saved score? (y/n): y

--- SCORES ---
Sam wins in 32 turns
Highest combo: 4
Card diff: 13

=== SCORE HISTORY ===
Record 1:
  Player  : Sam
  Turns   : 22
  HiCombo : 37

Record 2:
  Player  : Sam
  Turns   : 32
  HiCombo : 4

sam@Samuels-MacBook-Pro phase_3_CIS-17A_project1 %
```

Next, we create a separate file to house the scores structure.

```
#ifndef SCORES_H
#define SCORES_H

struct Scores
{
    int trns;
    int cmbHi;
};

#endif
```

Following that, we implement a feature in which we can prompt the user whether or not they want to save the recent score to the high scores page.

```
if (p1->hand.empty())
{
    cout << "You win!" << endl;
    char upd;
    cout << "Update your saved score? (y/n): ";
    cin >> upd;
    if (tolower(upd) == 'y')
    {
        calcScrs(*p1, *npc);
    }
}
```

We then create a new SaveData structure and copy the data into our new structure in order to save the data into our binary file.

```
// Save Scores Function
void calcScrs(Player &p1, Player &npc)
{
    p1.scr.trns = p1.trns;
    p1.scr.cmbHi = p1.cmbMx;

    int diff = npc.hand.size(); // card difference

    cout << "\n--- SCORES ---\n";
    cout << p1.name << " wins in " << p1.scr.trns << " turns\n";
    cout << "Highest combo: " << p1.scr.cmbHi << '\n';
    cout << "Card diff: " << diff << "\n";

    // Prepare SaveData object
    SaveData data;
    strncpy(data.name, p1.name.c_str(), sizeof(data.name));
    data.name[sizeof(data.name) - 1] = '\0'; // ensure null-terminated
    data.scr = p1.scr;

    ofstream out("scores.dat", ios::binary | ios::app);
    if (out)
    {
        out.write(reinterpret_cast<char *>(&data), sizeof(SaveData));
        out.close();
    }
    else
    {
        cerr << "Failed to write to scores.dat\n";
    }
}
```

Constructs and Concepts Checklist

Chapter	Section	Concept	Points for Inclusion	Location in Code (line)	Comments

9		**Pointers/Memory Allocation**			
	1	Memory Addresses			
	2	Pointer Variables	5	main - 64, 65	
	3	Arrays/Pointers	5	Player.h - line 13	
	4	Pointer Arithmetic			
	5	Pointer Initialization			
	6	Comparing			
	7	Function Parameters	5	main - 453	
	8	Memory Allocation	5	main - 64, 65	
	9	Return Parameters	5	main - 152 - 157	
	10	Smart Pointers			
10		**Char Arrays and Strings**			
	1	Testing			
	2	Case Conversion			
	3	C-Strings	10	main - 34	
	4	Library Functions			
	5	Conversion			
	6	Your own functions			
	7	Strings	10	main - 293	
11		**Structured Data**			
	1	Abstract Data Types			
	2	Data			
	3	Access			
	4	Initialize			
	5	Arrays	5	214	

	6	Nested	5	Player.h - 17	
	7	Function Arguments	5	main - 601	
	8	Function Return	5	main - 152-158	
	9	Pointers	5	main 64-68	
	10	Unions ****			
	11	Enumeration	5	Card.h - 4-26	
12		**Binary Files**			
	1	File Operations			
	2	Formatting	2	main - 644-651	
	3	Function Parameters	2	main - 601-628	
	4	Error Testing			
	5	Member Functions	2	Player.cpp - 6-25	
	6	Multiple Files	2	main - 15-16	
	7	Binary Files	5	main - 619	
	8	Records with Structures	5	main - 32, 630	
	9	Random Access Files	5	main - 667-684	
	10	Input/Output Simultaneous	2	main - 667	
		Total	100		

Project Initialization

This project is an exercise on applying basic programming skills used in our CIS-5 class. This first iteration cannot use functions or arrays and should provide a base game foundation which will then be expanded upon in Project 2 using more advanced programming practices.

Creating my desired product is not fully realizable within the parameters of this project. Nevertheless, the objective is to execute competency with the listed concepts in the objectives spreadsheet.

Upon initial review and consideration, the project's initial scope should be as follows:

The Base program should have a random card generator of either numbers and/or colors for both hands of the player and the computer, to cross reference and interact with the current card in play. Upon initial intuition, generating colors to compare should be much simpler than generating numbers to put into play. There are only four colors as compared to the 10 individual numbers (0-9).

Future implementation would consist of using 2D arrays in order to assign number values to base scope of color cards.

The program should contain a menu to prompt to start or exit the game.

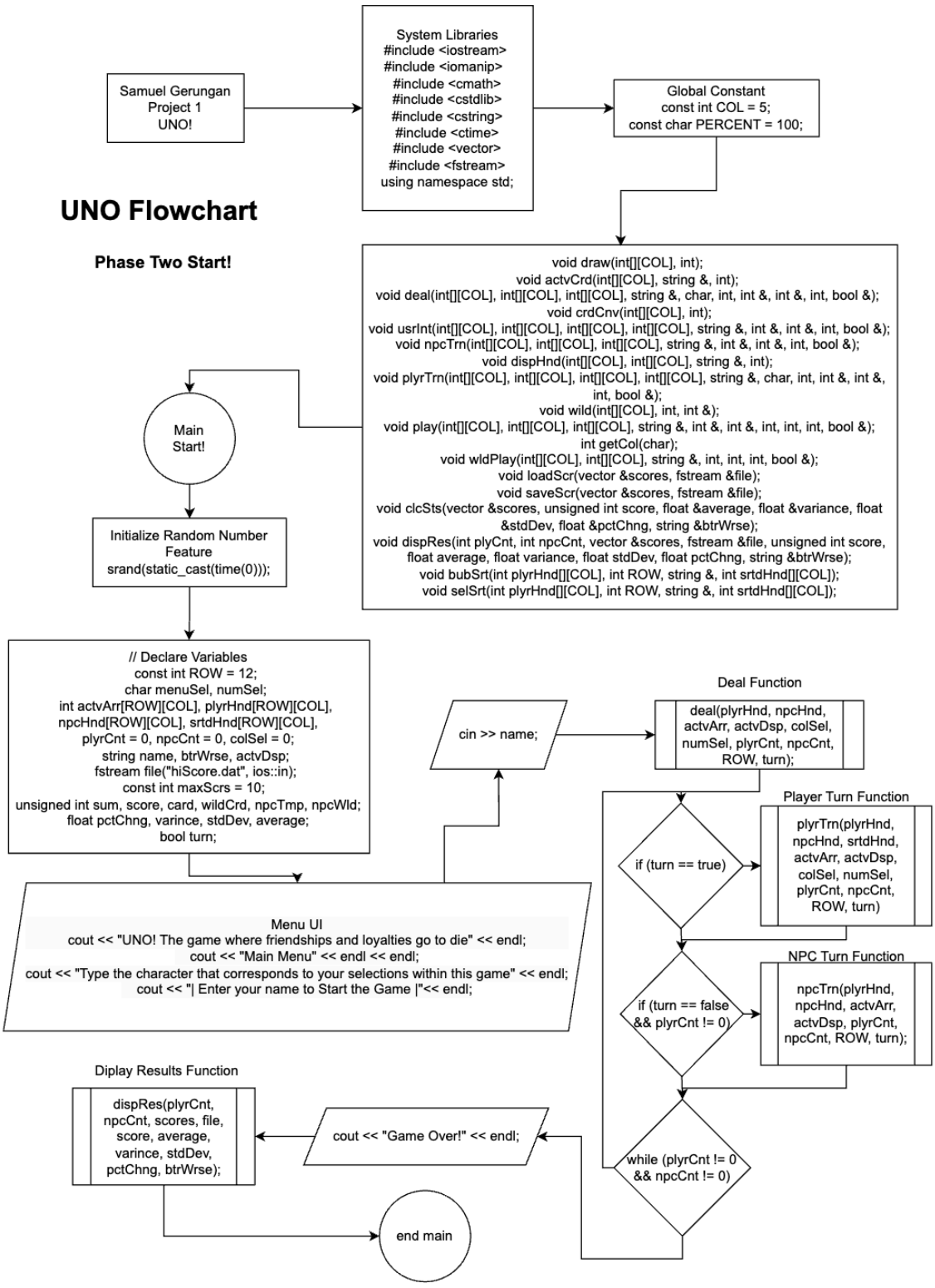
When the user selects start, the random generator will deal 5 cards to the player of random colors.

Game rules dictate that each player can take their turn in order to play a card with the same qualities or will have to draw a card from the deck. The player will be prompted during their turn to either manually draw a card or play one of the cards they may have in their hands. The user should also have the option to end the game at any time.

Program Theory and Flowchart

This segment serves to show the general flow of how the UNO game will function. We begin by initializing all the libraries and variables we will be utilizing throughout our program and initializing our main function.

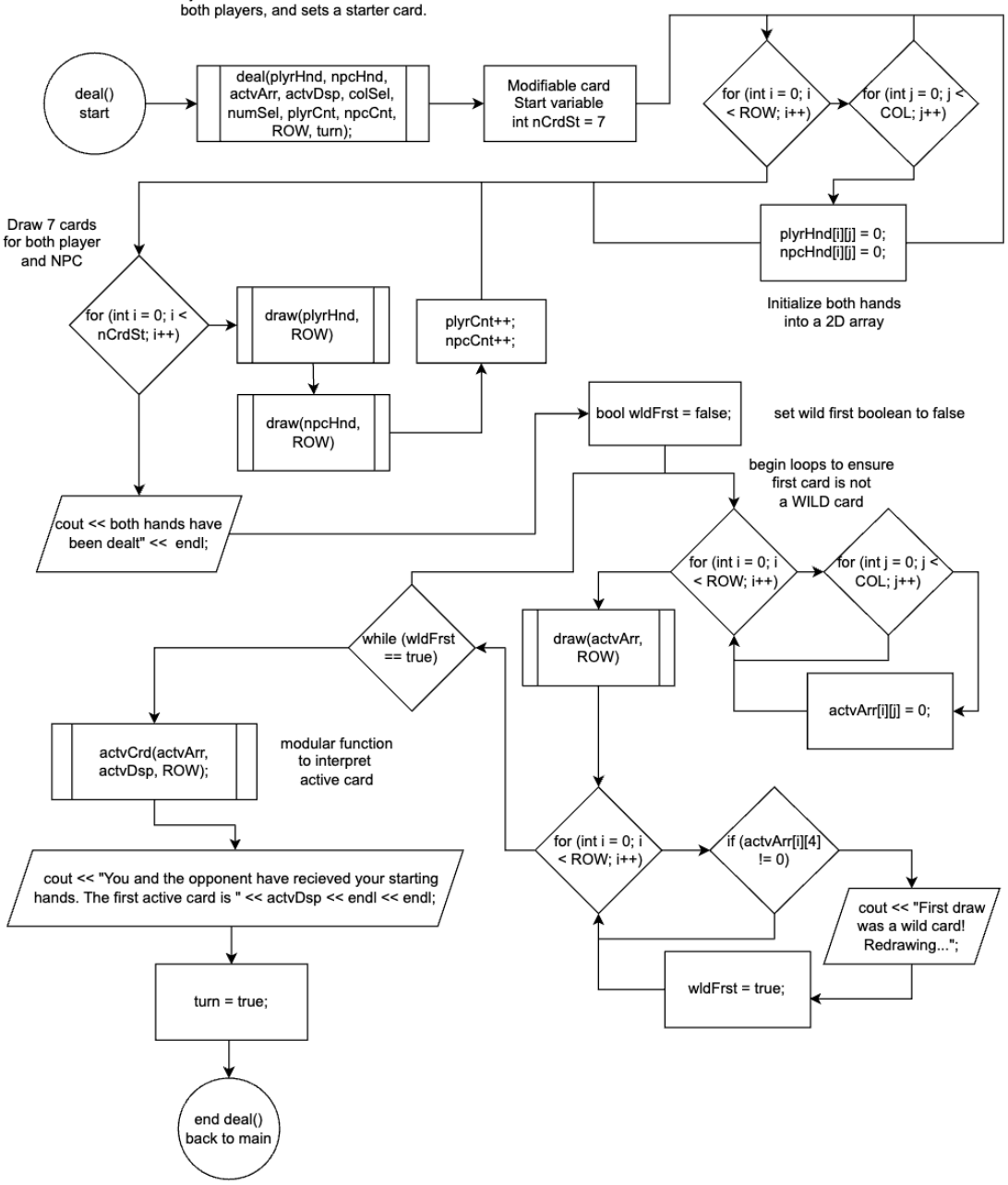
Within main, contains the very basic loop of how the program works, in which it creates a simple boolean to check if the player turn is true or false in order to handle turn passing between the player and the NPC. We then call our main functions of the player turn and the NPC turn and have a check to see if either player hands are at 0, in which a winner will be decided. Once a winner has been decided, we prompt the program to call the functions in order to fetch, calculate, and display the victory (or loss) sequence and show analytics of the last 10 victories.


Samuel Gerungan
Project 1
UNO!
System Libraries
#include <iostream>
#include <iomanip>
#include <cmath>
#include <cstdlib>
#include <cstring>
#include <ctime>
#include <vector>
#include <fstream>
using namespace std;
Global Constant
const int COL = 5;
const char PERCENT = 100;
UNO Flowchart
Phase Two Start!
void draw(int[][COL], int);
void actvCrd(int[][COL], string &, int);
void deal(int[][COL], int[][COL], int[][COL], string &, char, int, int &, int &, int, bool &);
void crdCnv(int[][COL], int);
void usrInt(int[][COL], int[][COL], int[][COL], int[][COL], string &, int &, int &, int, bool &);
void npcTrn(int[][COL], int[][COL], int[][COL], string &, int &, int &, int, bool &);
void dispHnd(int[][COL], int[][COL], string &, int);
void plyrTrn(int[][COL], int[][COL], int[][COL], int[][COL], string &, char, int, int &, int &, int, bool &);
void wild(int[][COL], int, int &);
void play(int[][COL], int[][COL], int[][COL], string &, int &, int &, int, int, int, bool &);
int getCol(char);
void wldPlay(int[][COL], int[][COL], string &, int, int, int, bool &);
void loadScr(vector &scores, fstream &file);
void saveScr(vector &scores, fstream &file);
void clcSts(vector &scores, unsigned int score, float &average, float &variance, float &stdDev, float &pctChng, string &btrWrse);
void dispRes(int plyCnt, int npcCnt, vector &scores, fstream &file, unsigned int score, float average, float variance, float stdDev, float pctChng, string &btrWrse);
void bubSrt(int plyrHnd[][COL], int ROW, string &, int srtdHnd[][COL]);
void selSrt(int plyrHnd[][COL], int ROW, string &, int srtdHnd[][COL]);
Main
Start!
Initialize Random Number
Feature
srand(static_cast(time(0)));
// Declare Variables
const int ROW = 12;
char menuSel, numSel;
int actvArr[ROW][COL], plyrHnd[ROW][COL],
npcHnd[ROW][COL], srtdHnd[ROW][COL],
plyrCnt = 0, npcCnt = 0, colSel = 0;
string name, btrWrse, actvDsp;
fstream file("hiScore.dat", ios::in);
const int maxScrs = 10;
unsigned int sum, score, card, wildCrd, npcTmp, npcWld;
float pctChng, varince, stdDev, average;
bool turn;
Menu UI
cout << "UNO! The game where friendships and loyalties go to die" << endl;
cout << "Main Menu" << endl << endl;
cout << "Type the character that corresponds to your selections within this game" << endl;
cout << "| Enter your name to Start the Game |"<< endl;
cin >> name;
Deal Function
deal(plyrHnd, npcHnd,
actvArr, actvDsp, colSel,
numSel, plyrCnt, npcCnt,
ROW, turn);
Player Turn Function
if (turn == true)
plyrTrn(plyrHnd,
npcHnd, srtdHnd,
actvArr, actvDsp,
colSel, numSel,
plyrCnt, npcCnt,
ROW, turn)
NPC Turn Function
if (turn == false
&& plyrCnt != 0)
npcTrn(plyrHnd,
npcHnd, actvArr,
actvDsp, plyrCnt,
npcCnt, ROW, turn);
while (plyrCnt != 0
&& npcCnt != 0)
cout << "Game Over!" << endl;
Diplay Results Function
dispRes(plyrCnt,
npcCnt, scores, file,
score, average,
varince, stdDev,
pctChng, btrWrse);
end main

The deal function is very important to create so that we can call upon it when a new game is initialized. We initialize both player and NPC hands using a 2D array then continue to call on the draw function shown above in order to deal the amount of cards dictated by the *nCrdStr* variable - which in standard UNO is 7 cards for the first hand. We also include a failsafe so that the first card placed CANNOT be a wild card, as in the standard game of UNO, must be reshuffled to put a standard card in play. Once the initial hands have been dealt, we pass the first turn to the player.

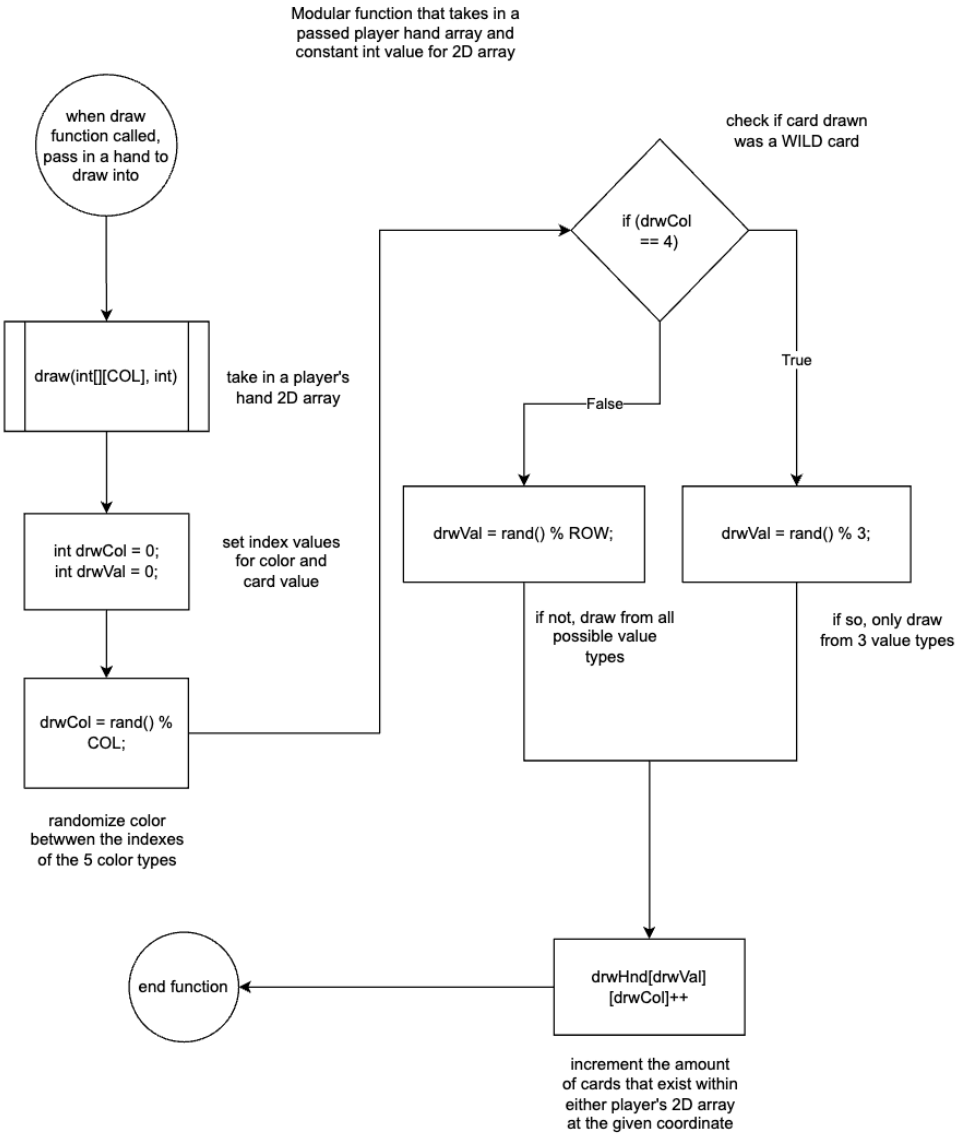
Deal Funcion

This function deals according to modular and modifiable values set by user. Dictates the first hands of both players, and sets a starter card.

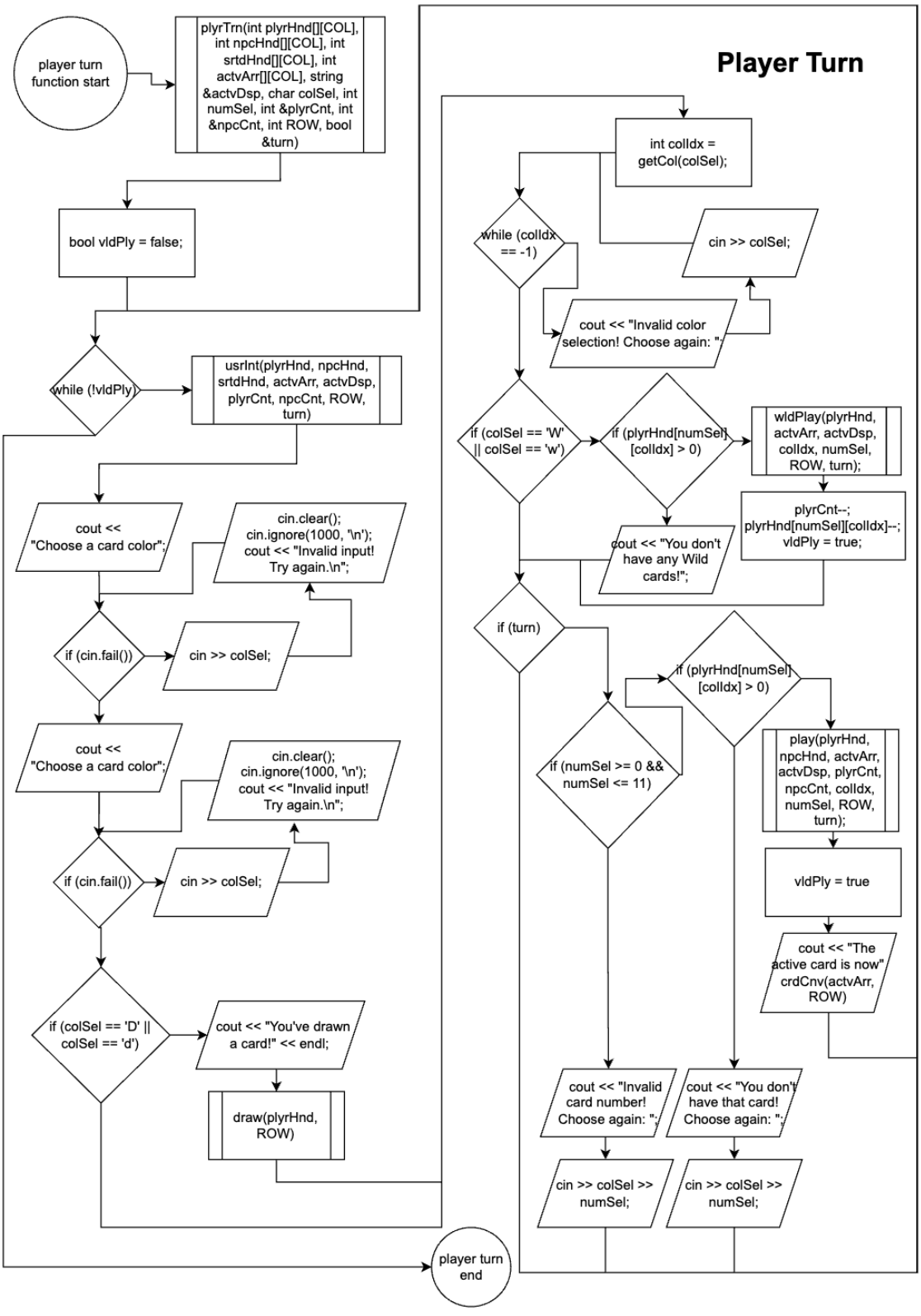


Below is the card draw function that fuels many of the functions within our program. The idea was to create a modular function which can be called multiple times throughout the program. The criteria in developing this function was so that we are able to pass the 2D array of either the player or the NPC and the draw function would be able to properly add another random value into the given array's index coordinates of *i* and *j*. The card draw function also has to be intelligent enough to discern the card type so that when a wild card is drawn, the function will assign it only a value of 0-2 instead of having the entire spectrum of card values.

Card Draw Function



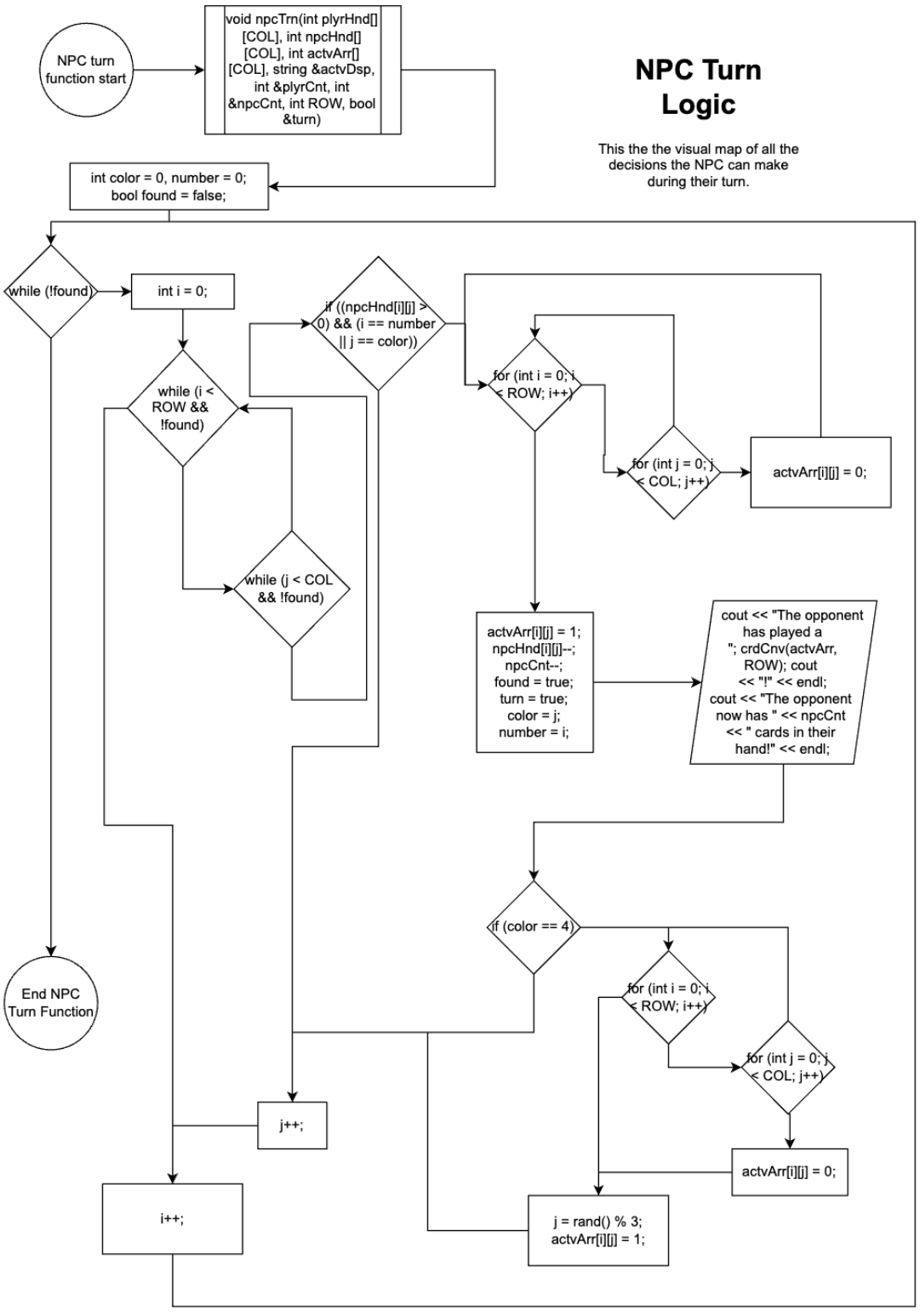
The base logic of the player turn begins below and is a giant nest using a while loop that checks to see if a valid play has been made. The idea is that we want to create incremental checks through each possible decision made by the player. This function accesses a plethora of other functions through our program and also has failsafes at every step to ensure that the decision made by the player is valid. If and only if the player makes a valid play, the boolean of valid play is set to true which then allows the program to pass the turn to the NPC


Player Turn
player turn function start
plyrTrn(int plyrHnd[][COL], int npcHnd[][COL], int srtdHnd[][COL], int actvArr[][COL], string &actvDsp, char colSel, int numSel, int &plyrCnt, int &npcCnt, int ROW, bool &turn)
bool vldPly = false;
while (!vldPly)
usrInt(plyrHnd, npcHnd, srtdHnd, actvArr, actvDsp, plyrCnt, npcCnt, ROW, turn)
cout << "Choose a card color";
cin.clear(); cin.ignore(1000, '\n'); cout << "Invalid input! Try again.\n";
if (cin.fail())
cin >> colSel;
cout << "Choose a card color";
cin.clear(); cin.ignore(1000, '\n'); cout << "Invalid input! Try again.\n";
if (cin.fail())
cin >> colSel;
if (colSel == 'D' || colSel == 'd')
cout << "You've drawn a card!" << endl;
draw(plyrHnd, ROW)
player turn end
int colIdx = getCol(colSel);
while (colIdx == -1)
cin >> colSel;
cout << "Invalid color selection! Choose again: ";
if (colSel == 'W' || colSel == 'w')
if (plyrHnd[numSel][colIdx] > 0)
wldPlay(plyrHnd, actvArr, actvDsp, colIdx, numSel, ROW, turn);
plyrCnt--; plyrHnd[numSel][colIdx]--; vldPly = true;
cout << "You don't have any Wild cards!";
if (turn)
if (plyrHnd[numSel][colIdx] > 0)
if (numSel >= 0 && numSel <= 11)
play(plyrHnd, npcHnd, actvArr, actvDsp, plyrCnt, npcCnt, colIdx, numSel, ROW, turn);
vldPly = true
cout << "The active card is now" crdCnv(actvArr, ROW)
cout << "Invalid card number! Choose again: ";
cout << "You don't have that card! Choose again: ";
cin >> colSel >> numSel;
cin >> colSel >> numSel;

The diagrams below show the intricate NPC logic that exists when making decisions based off several tiered list of priorities. The npc begins by searching both the 2D array of *active card* that store a single card as an index reference for what card is currently at play, and searches through their hand to see if there is a card that matches EITHER the number OR the color of the card. If such a card exists, the NPC will prioritize on playing a card of the same color, then number. If a car does not exist, it resorts to searching through its hand for a wild card and proceeding to swap the card with the color suit that has a predominant presence in its hand. If no such wild card exists, then the NPC will draw and start the logic loop all over again until a valid play has been made.

NPC Turn Logic

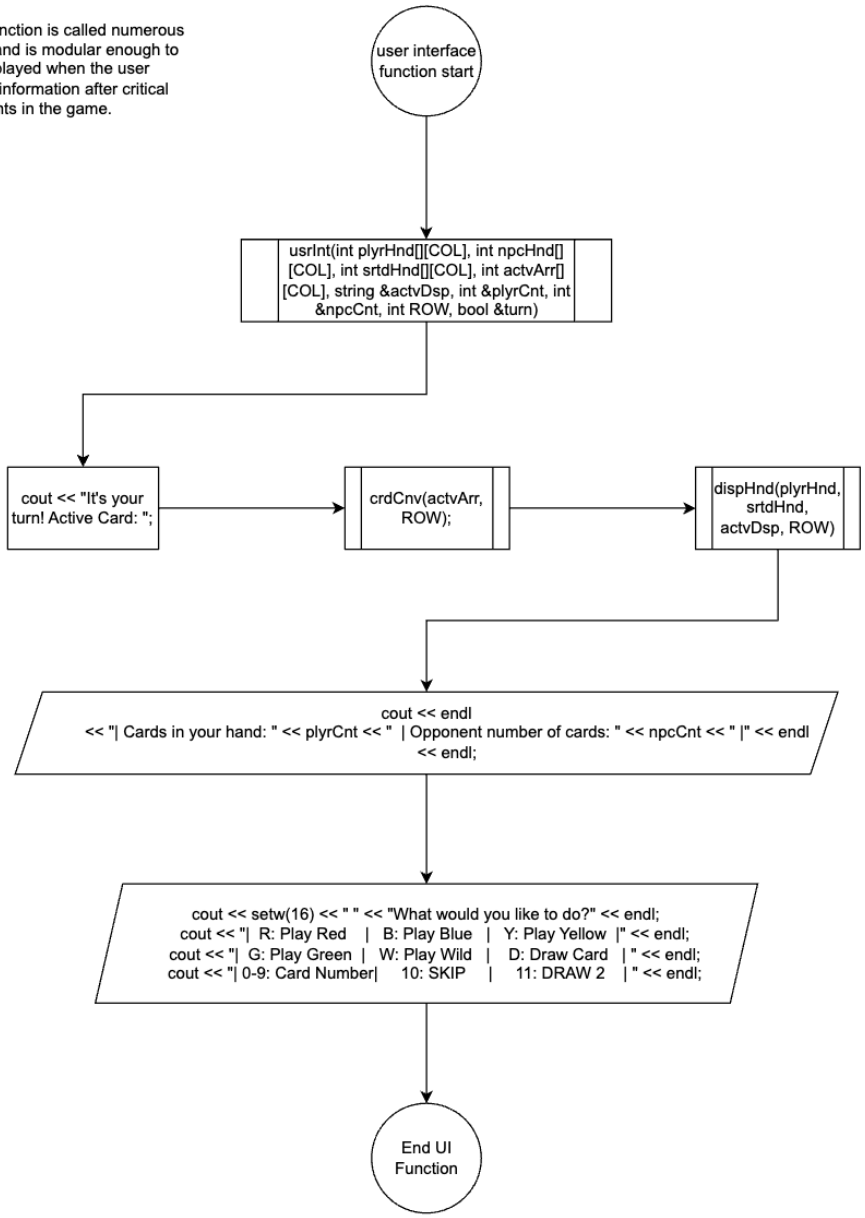
This the the visual map of all the decisions the NPC can make during their turn.



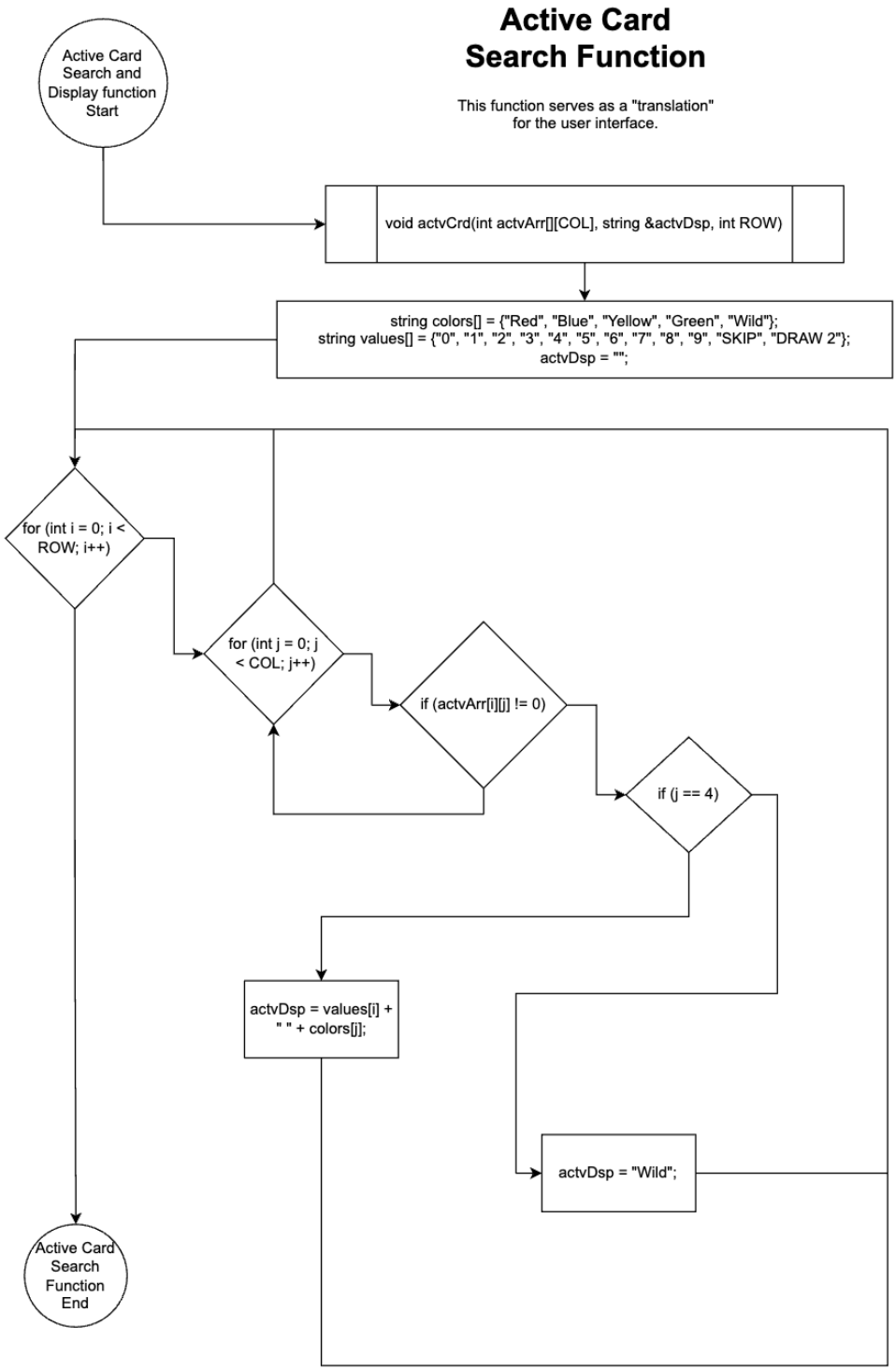
One of the most important pieces of the puzzle aside from the draw function is the User Interface function. This one was pivotal in my development priority because it enables me to have a modular function which I can call a numerous amount of times within my program. The UI is displayed every step of the way and being able to have a template to call on and display is monumental to the overall user experience of interacting with the game.

User Interface Function

This function is called numerous times and is modular enough to be displayed when the user needs information after critical moments in the game.



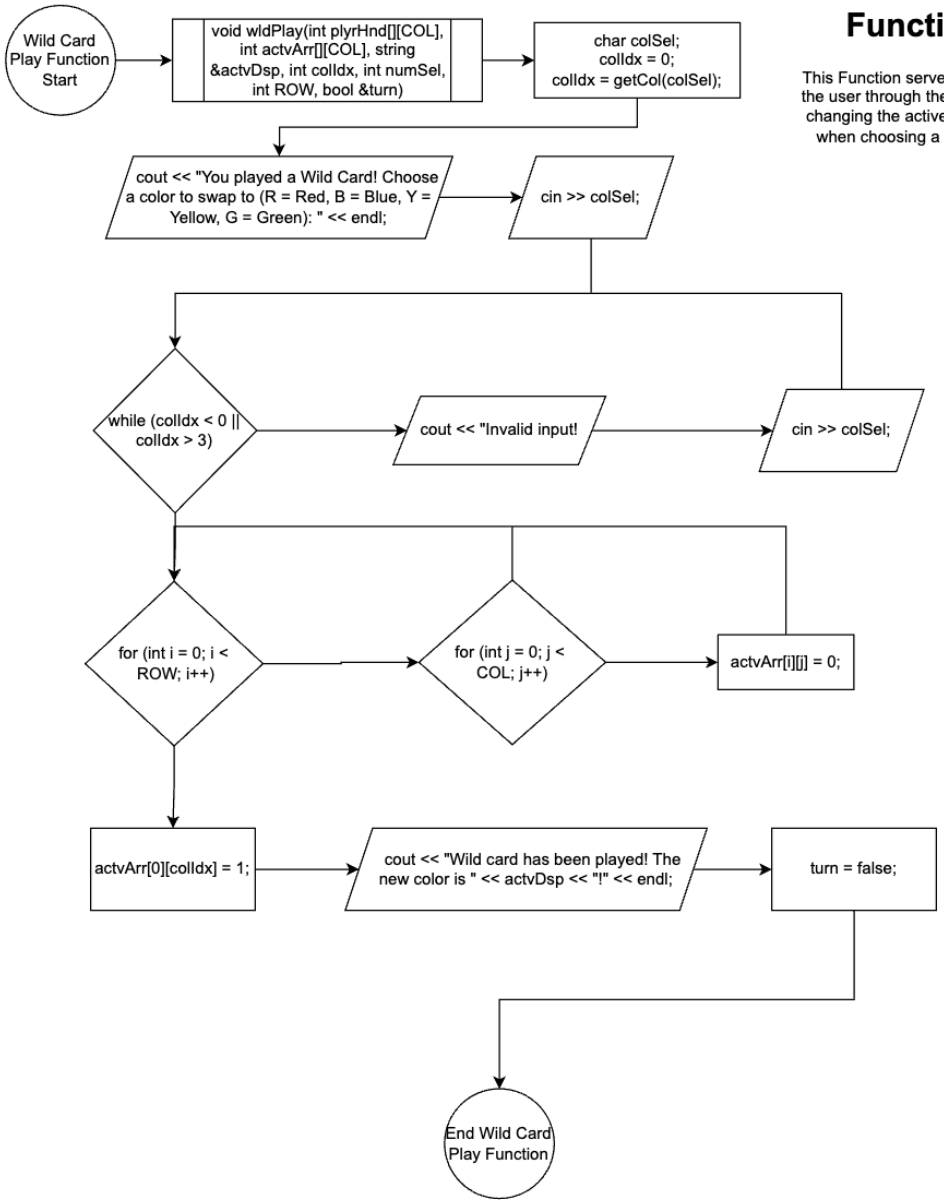
The flowchart below shows a modular function which we also call multiple times throughout various steps in our program. This function serves to be a translation for the user interface, such that it changes the index information of the stored active card array and translates it for the user to be able to read in a string format. Every time we must display cards put in play or drawn for the user, we call on this function in order to translate integer coordinates into a readable string.


Active Card Search Function
This function serves as a "translation" for the user interface.
Active Card Search and Display function Start
void actvCrd(int actvArr[][COL], string &actvDsp, int ROW)
string colors[] = {"Red", "Blue", "Yellow", "Green", "Wild"};
string values[] = {"0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "SKIP", "DRAW 2"};
actvDsp = "";
for (int i = 0; i < ROW; i++)
for (int j = 0; j < COL; j++)
if (actvArr[i][j] != 0)
if (j == 4)
actvDsp = values[i] + " " + colors[j];
actvDsp = "Wild";
Active Card Search Function End

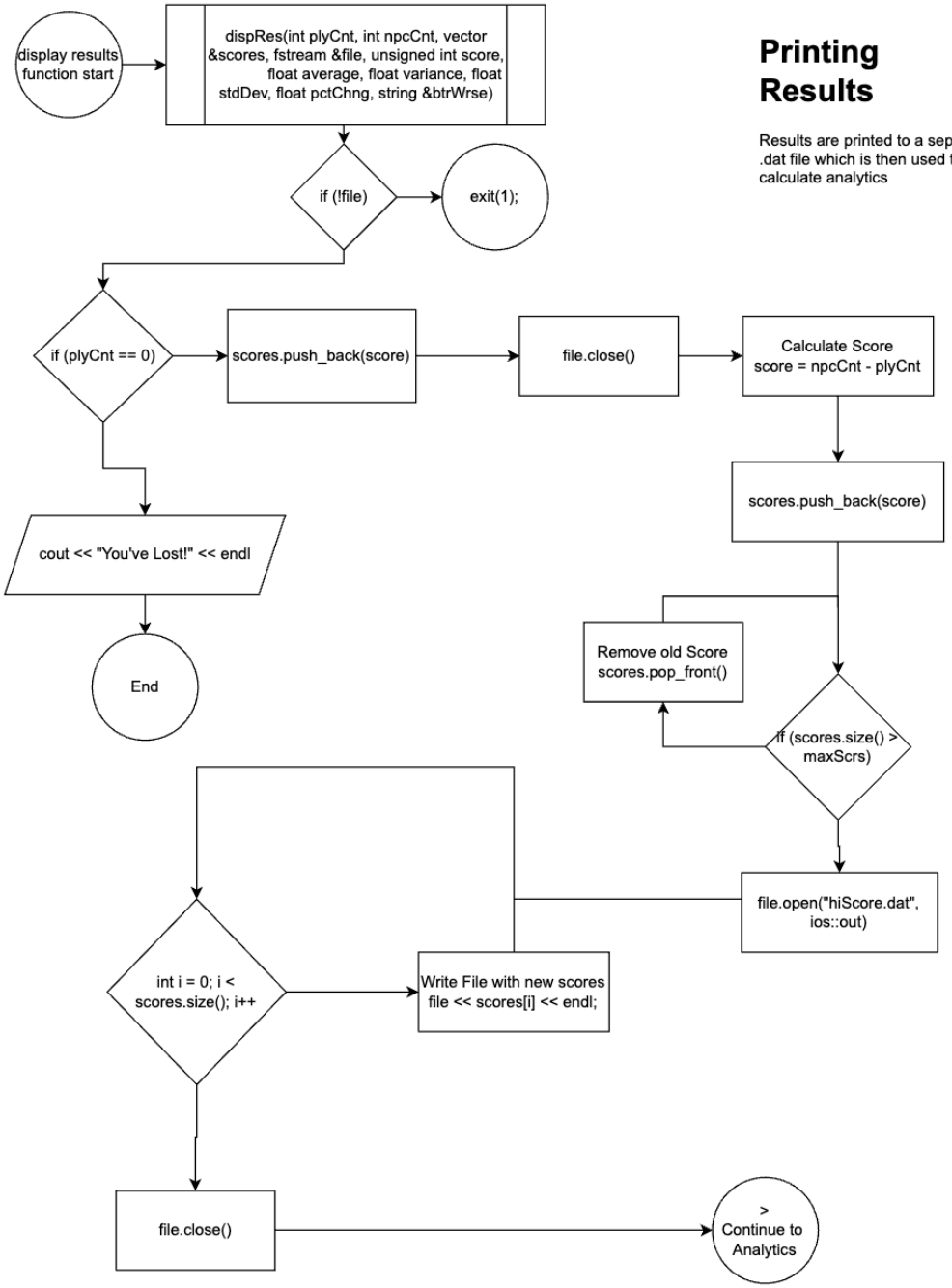
The Wild Card Play Function is one that is called by the player during their turn in order to prompt the user in the color swapping process. First we prompt the user to input a color selection and also have a failsafe check to ensure that the user does not input anything that may cause bugs. Once the user inputs a valid character we then clear the active card 2D array and replace it with coordinates according to the color the user selects.

Wild Card Play Function

This Function serves to prompt the user through the process of changing the active card color when choosing a wild card.



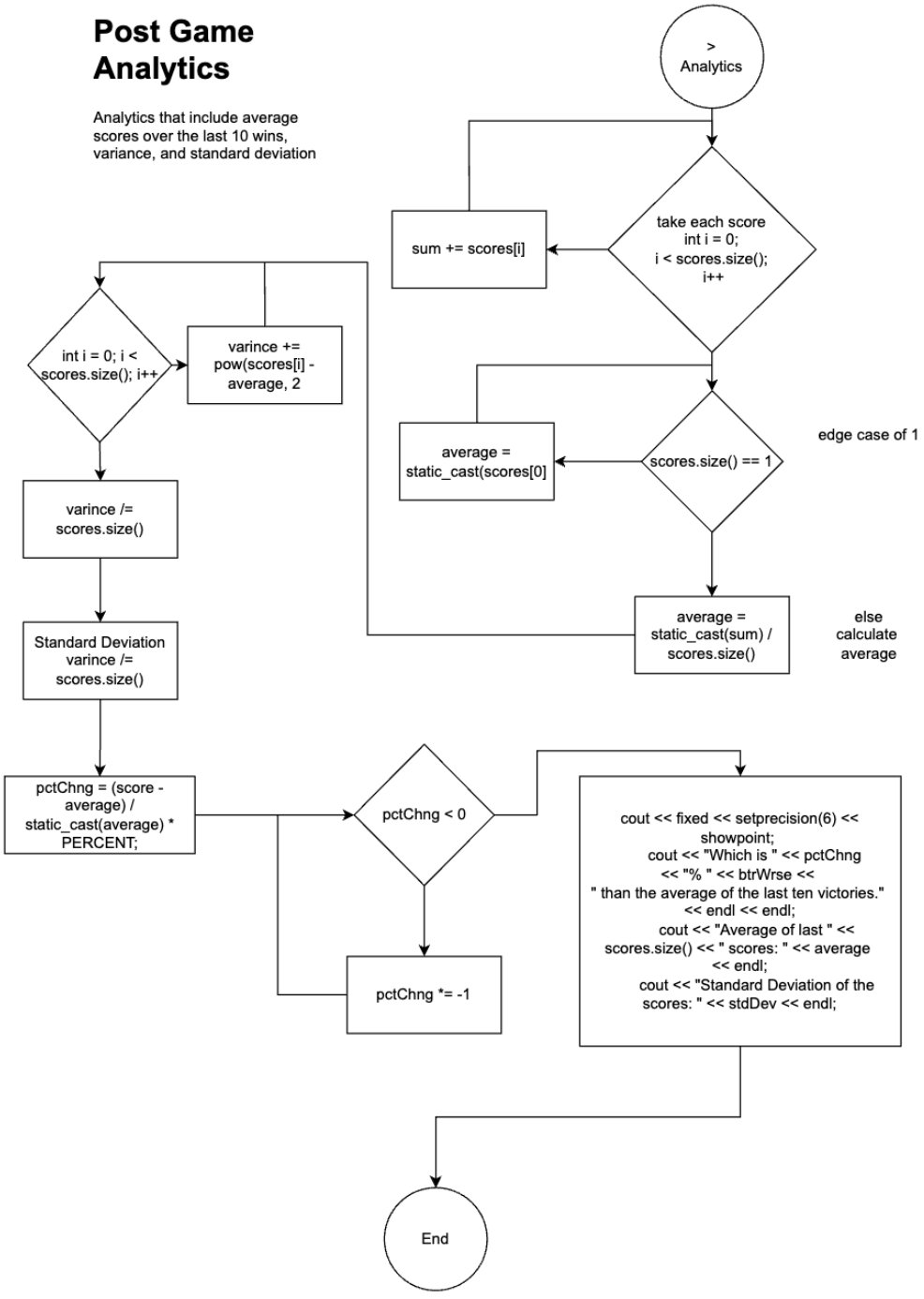
The display results function serves to print the initial results of the game and initialize the process of accessing the high scores sheet. If no such files exist, we include an exit function to immediately end the program. If the sheet exists we push the new entry to the back of the que of arrays which is limited to 10. If the sheet exceeds the limit of 10, we delete the entry at the very front of the que, which translates to a first-in-first out system.



Printing Results

Results are printed to a seperate .dat file which is then used to calculate analytics

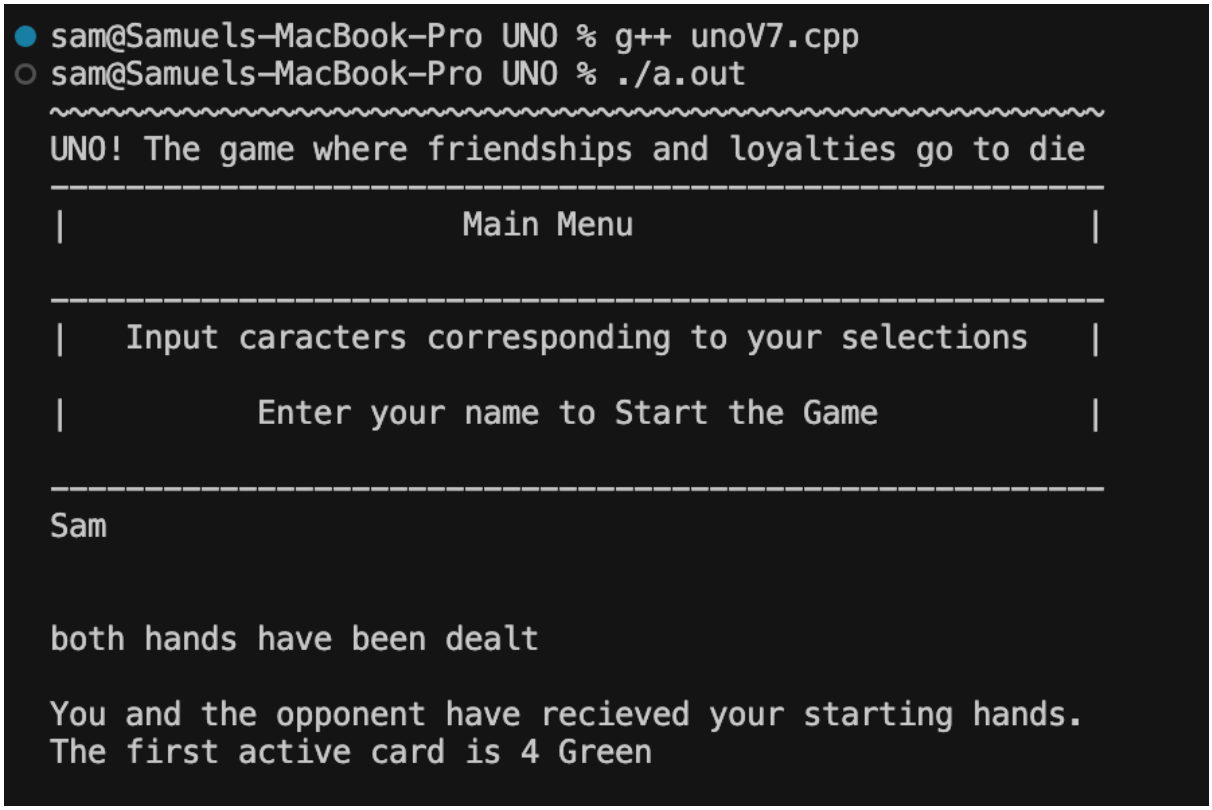
We then read in the information on the separate page in order to calculate various analytics, including percent change improvement (or not) of the submitted victories over the last 10 stored on the sheet, the average of the scores, and the standard deviation.


Post Game Analytics
Analytics that include average scores over the last 10 wins, variance, and standard deviation
> Analytics
take each score int i = 0; i < scores.size(); i++
sum += scores[i]
scores.size() == 1
edge case of 1
average = static_cast(scores[0]
average = static_cast(sum) / scores.size()
else calculate average
int i = 0; i < scores.size(); i++
varince += pow(scores[i] - average, 2
varince /= scores.size()
Standard Deviation varince /= scores.size()
pctChng = (score - average) / static_cast(average) * PERCENT;
pctChng < 0
pctChng *= -1
cout << fixed << setprecision(6) << showpoint;
cout << "Which is " << pctChng << "% " << btrWrse << " than the average of the last ten victories." << endl << endl;
cout << "Average of last " << scores.size() << " scores: " << average << endl;
cout << "Standard Deviation of the scores: " << stdDev << endl;
End

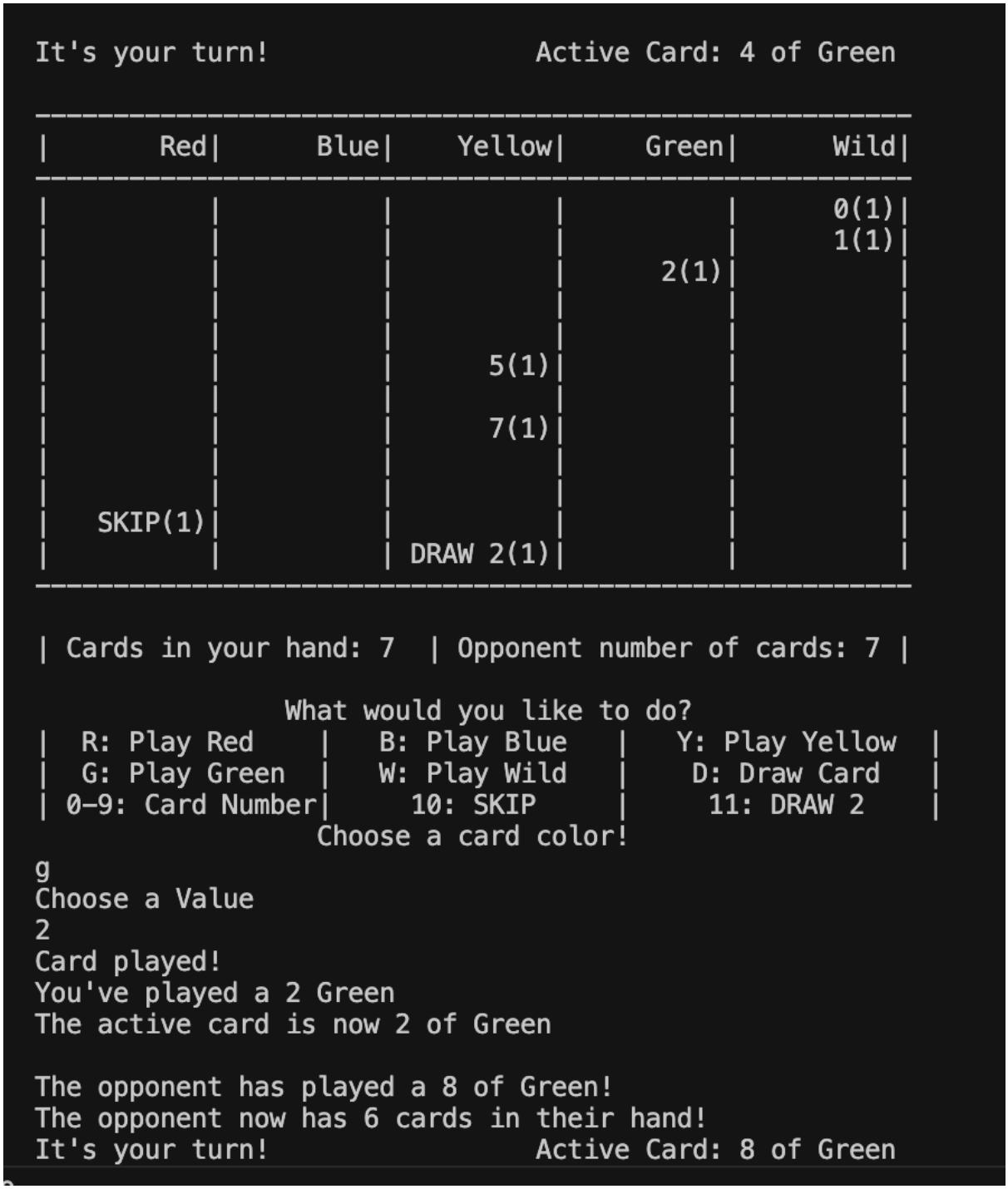
Proof of Concept for Phase 1 and 2

Here is a collection of screenshots that contain a full gameplay loop and show all the working functions of the UNO game.

> Sequence below shows the user shown the main menu upon execution of the program, and upon name input the game is initialized and hands are dealt to both the user and the NPC.



> Sequence below shows the function created to show an informational and functional user interface that shows everything in a compact and concise way. The displayed sequence also properly demonstrates the NPC AI capabilities to play according to the active card in play, and when a valid play is made, turns are passed between user and NPC.



> A continuation of Last Photo, the card draw feature adds to the hand count and hand display when opting to draw a card, and loops back to the turn of the player that opts to draw.

```
| Cards in your hand: 6  | Opponent number of cards: 6 |

               What would you like to do?
|  R: Play Red    |   B: Play Blue   |   Y: Play Yellow  |
|  G: Play Green  |   W: Play Wild   |    D: Draw Card   |
| 0-9: Card Number|     10: SKIP     |     11: DRAW 2    |
                  Choose a card color!
d
Choose a Value
1
You've drawn a card!
It's your turn!                Active Card: 8 of Green

_____________________________________________________
|       Red|      Blue|    Yellow|     Green|      Wild|
-----------------------------------------------------
|          |          |          |          |      0(1)|
|          |          |          |          |      1(1)|
|          |          |          |          |          |
|          |          |          |          |          |
|          |      4(1)|          |          |          |
|          |          |      5(1)|          |          |
|          |          |          |          |          |
|          |          |      7(1)|          |          |
|          |          |          |          |          |
|          |          |          |          |          |
|   SKIP(1)|          |          |          |          |
|          |          | DRAW 2(1)|          |          |
-----------------------------------------------------

| Cards in your hand: 7  | Opponent number of cards: 6 |
```

> Sequence below shows that the program allows players to match EITHER color or number, which then the computer is able to continue to play off of.

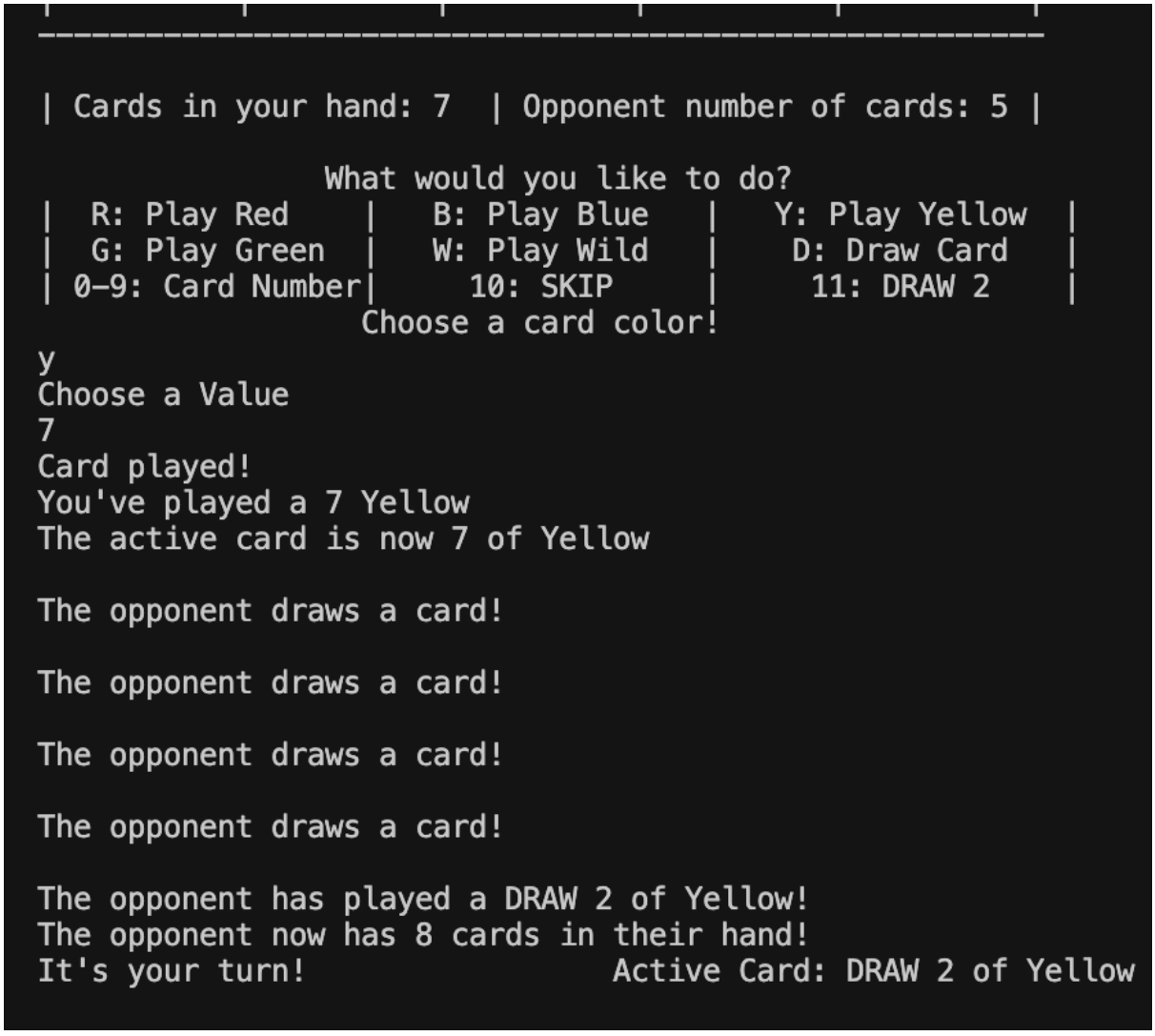
```
1
You've drawn a card!
It's your turn!                Active Card: 1 of Blue

-------------------------------------------------------
|       Red|      Blue|    Yellow|     Green|      Wild|
-------------------------------------------------------
|          |          |          |          |          |
|          |          |      1(1)|          |      1(1)|
|          |          |          |          |          |
|          |          |          |          |          |
|          |          |          |          |          |
|          |          |      5(1)|          |          |
|          |          |          |          |          |
|          |          |      7(1)|          |          |
|      8(1)|          |      8(1)|          |          |
|      9(1)|          |          |          |          |
|   SKIP(1)|          |          |          |          |
|          |          | DRAW 2(1)|          |          |
-------------------------------------------------------

| Cards in your hand: 9  | Opponent number of cards: 4 |

                What would you like to do?
|  R: Play Red    |   B: Play Blue   |   Y: Play Yellow  |
|  G: Play Green  |   W: Play Wild   |    D: Draw Card   |
| 0-9: Card Number|     10: SKIP     |     11: DRAW 2    |
                  Choose a card color!
y
Choose a Value
1
Card played!
You've played a 1 Yellow
The active card is now 1 of Yellow

The opponent has played a 9 of Yellow!
The opponent now has 3 cards in their hand!
It's your turn!                Active Card: 9 of Yellow
```

> Sequence below shows card DRAW 2 and SKIP functionality works. The number of the opponent's cards in hand increases by two when using a

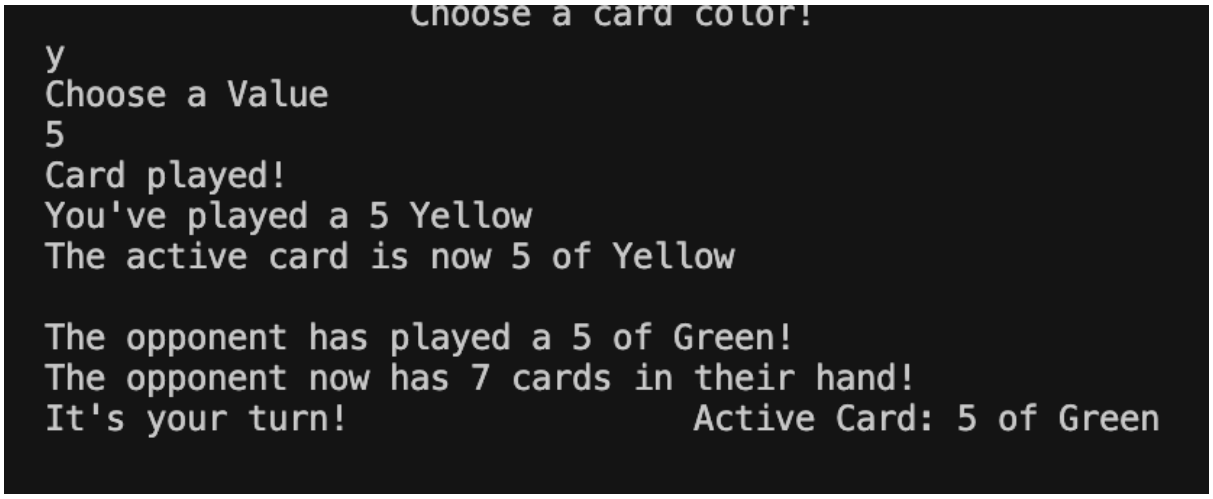
> DRAW 2 and both cases skips the NPC’s following turn to award player with another turn.

```
| Cards in your hand: 8  | Opponent number of cards: 3 |

                What would you like to do?
|  R: Play Red     |   B: Play Blue    |   Y: Play Yellow  |
|  G: Play Green   |   W: Play Wild    |    D: Draw Card   |
| 0-9: Card Number|      10: SKIP      |     11: DRAW 2    |
                  Choose a card color!
y
Choose a Value
11
Card played!
You've played a DRAW 2 Yellow
The opponent has to Draw Two Cards!

The active card is now DRAW 2 of Yellow

It's your turn!                Active Card: DRAW 2 of Yello

_____________________________________________________
|       Red|      Blue|    Yellow|     Green|      Wild|
-----------------------------------------------------
|          |          |          |          |          |
|          |          |          |          |      1(1)|
|          |          |          |          |          |
|          |          |          |          |          |
|          |          |          |          |          |
|          |          |      5(1)|          |          |
|          |          |          |          |          |
|          |          |      7(1)|          |          |
|      8(1)|          |      8(1)|          |          |
|      9(1)|          |          |          |          |
|   SKIP(1)|          |          |          |          |
|          |          |          |          |          |
-----------------------------------------------------

| Cards in your hand: 7  | Opponent number of cards: 5 |
```

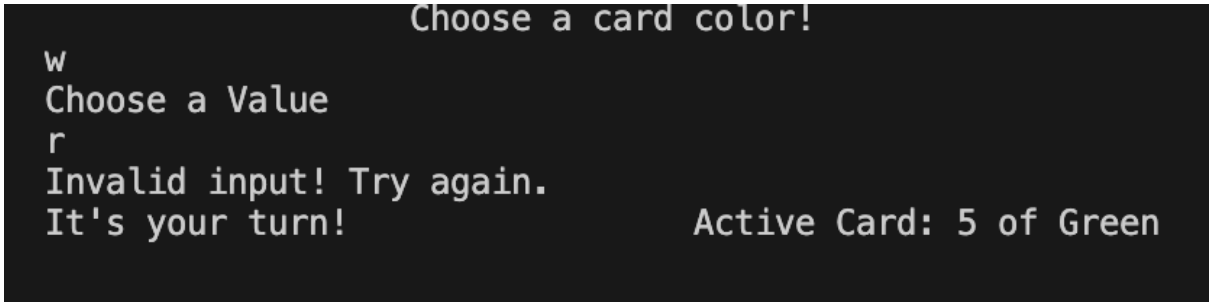
> Sequence below shows that when the opponent does not have any cards that either have matching colors or numbers, the opponent will continuously draw, add to the internal and external count of cards in hand, and stop when has reached a card that is able to be played.



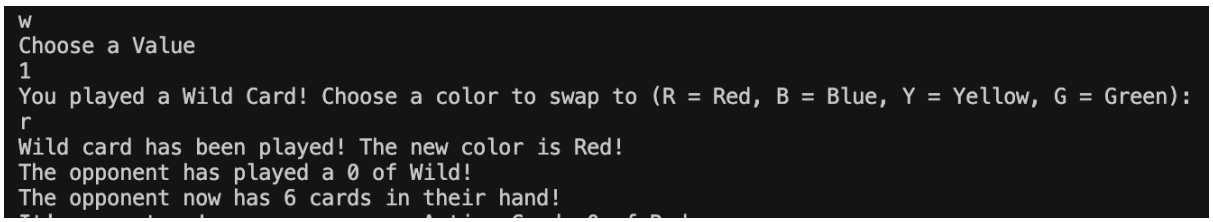
> Sequence below shows that the NPC is able to intelligently choose to play a matching number when appropriate to strategically swap to another color that they have available. The active color is converted from yellow to green by placing down a 5 of green to match the player's placement of a 5 of yellow in order to swap the active color to green.



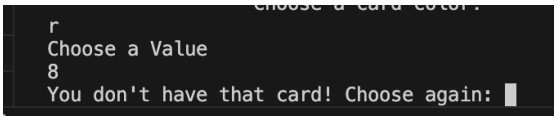
> Sequence below shows that there are carefully thought out debug prompts and loops to validate that card values played are appropriate to game rules.



> The sequence below shows a common high tempo play sequence that exists in UNO in which explosive color swapping can exist when players want to swap colors strategically in order to force draws by the opponent, or to swap to a favorable color that they have an abundance of. The player in this case swaps the active color to red, and the NPC can actively swap the active color again by checking their hand for available colors - and if none present, play a wild card for themselves.



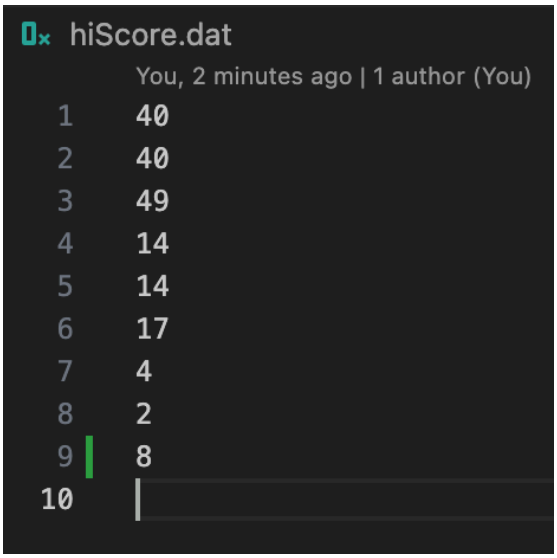
> Short sequence is shown as an external fail case check in which the player chooses a card they do not have in hand.



> Finally, the sequence below shows the end of a full gameplay loop in which the user wins the game, and is prompted that the game is finished. The user is notified of their score and shown analytics based off of the last 10 games WON and stored from an external *hiScores.dat* file.

```
The opponent has played a DRAW 2 of Yellow!
The opponent now has 8 cards in their hand!
It's your turn!                Active Card: DRAW 2 of Yellow

_____________________________________________________
|       Red|      Blue|    Yellow|     Green|      Wild|
_____________________________________________________
|          |          |          |          |          |
|          |          |          |          |          |
|          |          |          |          |          |
|          |          |          |          |          |
|          |          |          |          |          |
|          |          |          |          |          |
|          |          |          |          |          |
|          |          |          |          |          |
|          |          |      8(1)|          |          |
|          |          |          |          |          |
|          |          |          |          |          |
|          |          |          |          |          |
_____________________________________________________

| Cards in your hand: 1  | Opponent number of cards: 8 |

                What would you like to do?
|  R: Play Red    |   B: Play Blue   |   Y: Play Yellow  |
|  G: Play Green  |   W: Play Wild   |    D: Draw Card   |
| 0-9: Card Number|     10: SKIP     |     11: DRAW 2    |
                  Choose a card color!
y
Choose a Value
8
Card played!
You've played a 8 Yellow
The active card is now 8 of Yellow

Game Over!

Congratulations, you've won!

You've scored 8 points!
Which is 61.70% worse than the average of the last ten victories.

Average of last 9 scores: 20.89
Standard Deviation of the scores: 16.46
sam@Samuels-MacBook-Pro UNO %
```

> External file is displayed below which is confirmed to properly calculate percentage comparison to last 10 games played, the average of the previously won games, and the standard deviation of the scores.



Project Phase 1 Versions

Version 1.1

Creating Variables

The first step in conceptualizing my game is to create variables that can store the information needed for the card game. Since I'm unable to use arrays, we can only start with one characteristic that ties to each card, and in this case I'm choosing one of four colors for simplicity.

Another thing to consider as well is that since I can't use methods that will allow me to attribute an array of cards and suits to a player due to limitations of the project, I will have to think outside of the box in how I want to handle cards.

I've decided on listing out all of the potential cards that can be obtained and creating a stored variable in order to store the amount of player owned color cards and NPC owned color cards. This enables the base functionality that exists in UNO in which players have the option to hold on to cards for strategic purposes in order to have a wider array of answers for what is needed to win the game.

Concept Explanation in Dealing Color Attributed Cards

The simplest way I've devised to classify card colors is by assigning numbers to the different colors. We need four different colors in correlation to each color which in this program I've decided will be 0-3.

To initialize each variable, we will mass initialize each counter to 0.

```
// Declare Variables
char
    menuSel; // main menu selection

unsigned int
    card,    // card placeholder
    redCard, // user number of red cards
    bluCard, // user number of blue cards
    yelCard, // user number of yellow cards
    grnCard, // user number of green cards
    npcRed,  // npc number of red cards
    npcBlu,  // npc number of blue cards
    npcYel,  // npc number of yellow cards
    npcGrn;  // npc number of green cards

// Initialize Variables
redCard = bluCard = yelCard = grnCard = npcRed = npcBlu = npcYel = npcGrn = 0; // mass initialization of base card value to 0
// CREATE MENU
```

Creating a User Interface

The next step is to create a simple user interface that is easy to read and navigate

We begin by creating a banner for the UNO game, and a general start and exit option. We'll strive to implement the exit option throughout every game menu.

We will then create a user counter for the amount of cards available of each color.

Random Card Draw Functionality

To Create a random card draw, we will introduce the *ctime* and *stdlib* libraries in order to randomize our card draw variable.

We will then create a for loop that draws up to the number of cards in the initial hand which will be 5 for now. Within each iteration of the loop, we will set the variable *card* to be equivalent to the randomly generated number between 0-3, which will then be translated to the appropriate color-number correlation as follows:

Red - 0
Blue - 1
Yellow - 2
Green - 3

```
// CREATE MENU
cout << "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~" << endl;
cout << "UNO! The game where friendships and loyalties go to die" << endl;
cout << "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~" << endl
     << endl;
cout << "Main Menu" << endl
     << endl;
cout << "Type the character that corresponds to your selections within this game" << endl
     << endl;
cout << "| S : Start Game | E : Exit Game |";
cin >> menuSel;
cout << endl;

// Map the inputs and outputs - Process
// INITIAL DRAW of 5 CARDS
for (int i = 0; i < 5; i++)
{
    card = rand() % 4;                          // random 0-3 representing four colors - red blue yellow green
    card == 0 ? redCard++ : card == 1 ? bluCard++ // ternary operator to translate random card draw
                          : card == 2   ? yelCard++
                                        : grnCard++;
}

cout << "Red Cards: " << redCard << " Blue Cards : " << bluCard << " Yellow Cards: " << yelCard << " Green Cards: " << grnCard << endl;

// Display and output the results
```

The loop will then increment the color counters and we then print the number of cards within each color category that exists within the player's hand.

The program we've created now looks like this within the command prompt:

```
sam@Samuels-MacBook-Pro UNO % ./a.out
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
UNO! The game where friendships and loyalties go to die
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Main Menu

Type the character that corresponds to your selections within this game

| S : Start Game | E : Exit Game |S

Red Cards: 0 Blue Cards : 0 Yellow Cards: 3 Green Cards: 2
sam@Samuels-MacBook-Pro UNO %
0
```

In summary, we've taken our first steps to create base functionality in our UNO game. We've created the foundation of how our cards will be dealt and brainstormed through how we want to attribute the player's and NPC hands without the use of arrays and functions.

The next version's goal will be to create the framework for the NPC to be dealt a hand and create the base game logic of being able to put an appropriate card down in play and have the game simultaneously prompt the user to draw if no cards match the current color, or if the user so chooses to draw for strategy's sake.

> UNO Version 1.0 has now been deployed to GitHub at
>
> https://github.com/Sam-T-G/UNO/releases/tag/v1

https://github.com/Sam-T-G/UNO/releases/tag/v1

Version 1.2

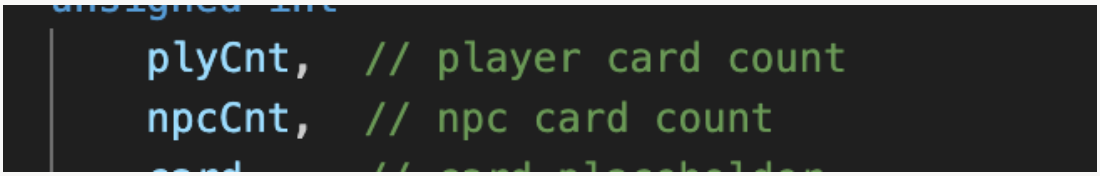
Win or Lose Case

One challenge that's taken a significant amount of time for my newbie programmer brain to conceptualize is how the game should be structured overall since I'm unable to use functions to create modular segments of my game. This poses an issue in which we need to structure our entire program through carefully though nested loops, and the best way I can tackle such challenge is to work backwards.

We need to structure our biggest check to be around the win or lose case, and in the case of UNO, the game ends when a player no longer has cards in their hand.

Therefore, our largest check or wrapper loop should be to check if either player reaches zero cards.

We begin by initializing two variables to keep track of card count amount and also logic to add up the sum of the cards currently in each player's hands.



Within our loops we also add an increment to both the NPC and player's card count variables whenever either is dealt a card.

Active Card Play Functionality

The next step is to implement a randomly generated active card in play once both the player and the NPC are dealt cards in their hand. To make amends with original allotted 5 cards, we will now adjust our program to deal 7 cards as the actual rules in UNO dictate.

We should now create another variable to store a random card value, and also generate a random card in order to have an active card in play. When complete, display card to player.

```
// if - else if statements to translate rng number into colors
if (actCrd == 0)
{
    actCol = "Red";
}
else if (actCrd == 1)
{
    actCol = "Blue";
}
else if (actCrd == 2)
{
    actCol = "Yellow";
}
else if (actCrd == 3)
{
    actCol = "Green";
}
cout << "CARD IN PLAY: " << actCol; // display card in play
```

We then create a conversion using if-else if statements (to meet our quota) to translate the RNG number to strings readable by the user and display it to our UI.

```
CARD IN PLAY: Red
| Red Cards: 1 | Blue Cards : 3 | Yellow Cards: 1 | Green Cards: 2 | Total number of cards in hand: 7 |
```

User Play and Draw Operations

The final step in Version 2 that I wanted to accomplish was the program's capability of taking user input and changing the stored values in the program.

For example:
When the program asks me what I want to do, when pressing draw, the program must assign me another card, increment the card count in the UI for the appropriate color, and also increment the overall card count.

When the USER selects to play a specific card, the program will opt to compare the selected card with the card in play. If the selected card matches the card in play AND the player has one in their hand, put the card in play, and decrement the count in the hand and overall card count.

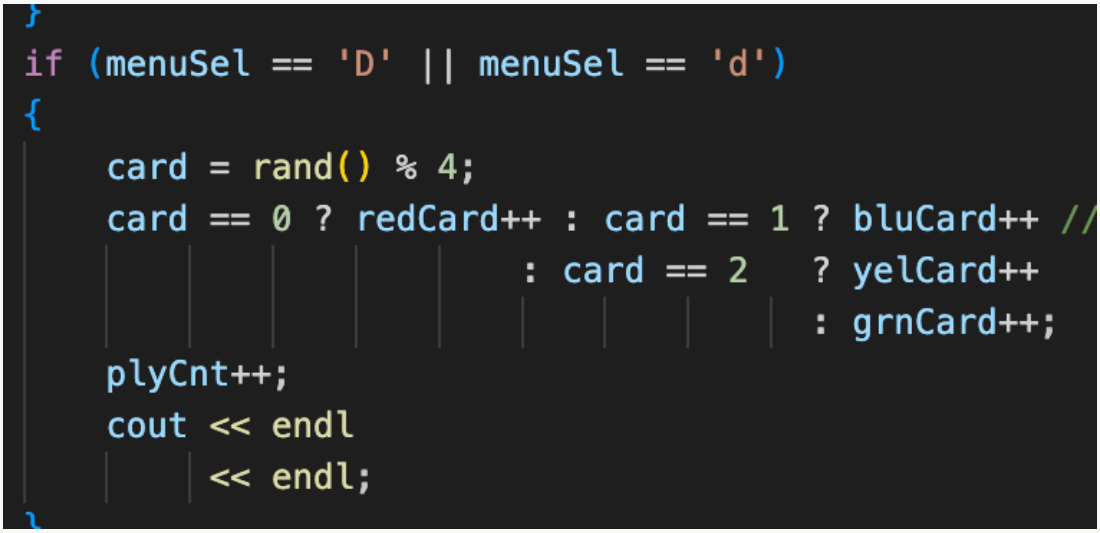
To achieve this, I used five
if statements for each of the possible player responses, four color plays, and a draw.

The color plays are copy and paste versions of the following.

```
if (menuSel == 'R' || menuSel == 'r')
{
    if (redCard > 0 && actCrd == 0)
    {
        redCard--;
        plyCnt--;
        cout << "You play a red card!" << endl
             << endl;
    }
    else if (actCrd != 0)
    {
        cout << "The colors don't match!" << endl
             << endl;
    }
    else
    {
        cout << "You don't have any RED cards!" << endl
             << endl;
    }
}
```

I included catch conditions in the case that the player does not have any of the cards selected to play and also if the selected card does not match the active card.

In the case of the draw functionality, I wrote the following.



```
}
if (menuSel == 'D' || menuSel == 'd')
{
    card = rand() % 4;
    card == 0 ? redCard++ : card == 1 ? bluCard++ //
                          : card == 2   ? yelCard++
                                        : grnCard++;
    plyCnt++;
    cout << endl
         << endl;
}
```

I prompted the program to set the card temp value to a random number again and increment the hand and color value accordingly.

Command Line Test and Finishing touches to V2

Adding some final finishing touches with spacing, formatting, and covering various small bugs when it comes to interpreting user input, the program now functions as intended.

When choosing the correct match card, the program prompts the user that they've played an appropriate card, the color card count is decremented, as well as the total number of cards in hand.

```
CARD IN PLAY: Red
| Red Cards: 1 | Blue Cards : 1 | Yellow Cards: 3 | Green Cards: 2 | Total number of cards in hand: 7 |

What would you like to do?
| R: Play Red | B: Play Blue | Y: Play Yellow | G: Play Green | D: Draw Card |

r

You play a red card!

CARD IN PLAY: Red
| Red Cards: 0 | Blue Cards : 1 | Yellow Cards: 3 | Green Cards: 2 | Total number of cards in hand: 6 |
                                                                    You, 7 minutes ago   Not Committed Yet
```

When choosing the wrong colored card, the program appropriately tells the user that the color is not of the correct color.

```
What would you like to do?
| R: Play Red | B: Play Blue | Y: Play Yellow | G: Play Green | D: Draw Card |

b

The colors don't match!

CARD IN PLAY: Red
| Red Cards: 0 | Blue Cards : 1 | Yellow Cards: 3 | Green Cards: 2 | Total number of cards in hand: 6 |

What would you like to do?
| R: Play Red | B: Play Blue | Y: Play Yellow | G: Play Green | D: Draw Card |
                                                                    You, 7 minutes ago   Not Committed Yet
```

Finally, when the user prompts the program to draw a card, the program will randomly draw a card and increment the proper card color as well as the total number of cards in hand.

```
CARD IN PLAY: Red
| Red Cards: 0 | Blue Cards : 1 | Yellow Cards: 3 | Green Cards: 2 | Total number of cards in hand: 6 |

What would you like to do?
| R: Play Red | B: Play Blue | Y: Play Yellow | G: Play Green | D: Draw Card |

d



CARD IN PLAY: Red
| Red Cards: 0 | Blue Cards : 1 | Yellow Cards: 3 | Green Cards: 3 | Total number of cards in hand: 7 |
```

You, 7 minutes ago Not Committed Yet

> UNO Version 2.0 has now been deployed to GitHub at
>
> https://github.com/Sam-T-G/UNO/releases/tag/v2

https://github.com/Sam-T-G/UNO/releases/tag/v2

Version 1.3

Goals and Implementations for Version 3

Now that we have our base functionality of dealing cards, playing cards, and base UNO game functions to play appropriate colored cards and also opt to draw cards for strategic purposes, the next goals in version three are as follows:

- Create computer auto playing functionality in which the computer defaults to playing the appropriate colored cards. If no cards exist, the computer will automatically draw a card.
- Restructure program so that the base condition of winning and losing will be when the card count of either the player or npc reaches zero.
- Clean up user interface further: Add NPC card count to UI and various other formatting cleanup.

Once we achieve those three goals, version three should leave a relatively thorough beginning to end user experience in playing the UNO game.

First and foremost, I wrapped the base function of the program using a do-while loop so that the program will properly go through one cycle of base functionality, and after both players' turns end, the program will end.

MPORTANT CONSIDERATION!
There is an edge case in which the player AND the npc could end their turns within the same cycle of the same game. To save time and prioritize the rest of the functionality we'll table that topic for later, either towards the end of the development of project 1 or save it for when we are able to compartmentalize functions within project 2.

Stop and Think:
I'm unsure at this point as to how I want to organize player turn handling and checks. Thinking further of the consideration listed above, some problems arise when I have both player turn and NPC turns handled within a single sweep of the do-while loop. Although i think the handling of the single pass and checking for loop functionality makes sense, future implementation of cards such as the skip or draw two cards will pose a problem.

Now that I'm taking the time to consider the possibilities, I think handling turns would be best handles with a boolean of if its the player or computer's turn. I can implement this with the current code and this makes it easy to then perform a check when swapping between turns to check whether or not the win conditions have been met.

I dread implementing this without the use of functions but here we go.

While developing this program, I've come to the realization that with the new boolean check to see whose turn it is, I have to fundamentally restructure how the entire program loops through turns and moves executed during the turn. I also realized that I have to rethink the way the program will process specific moves done during both the NPC and the player's turn. For example, when the player draws a card, it still needs to be the player's turn which allows them to continuously draw cards or choose to play one within the same turn.

```
// if it's not the player's turn AND if the player has not won yet
if (plyrTrn == false && plyCnt != 0)
{
    // program displaying opponents turn and game status
    cout << "It's the opponent's turn!" << endl
        << endl;      You, 4 minutes ago • Uncommitted changes
    cout << "The opponent has " << npcCnt << " cards in their hand." << endl
        << endl;
    cout << "The active card is " << actCol << "." << endl // call the active color string to display active color
        << endl;
```

To start out the NPC functionality, I started with the base wrapper and condition in which the player turn boolean is set to false AND the human player’s card count has not reached zero.

I then scripted out some general responses to explain the status of the game which includes feedback that it’s the opponent’s turn, how many cards the opponent has, and what the current active card is.

I then wrote out the logic for the NPC's turn.

Admittedly, this took a while for me to brainstorm, edit and processed but I ended up with the following:

```
// have npc check active card
if (actCrd == 0) // if active card is red
{
    if (plyrTrn == false) // if it's still the opponent's turn
    {
        if (npcRed > 0) // nested case if npc also has red cards
        {
            cout << "Opponent plays a red card!" << endl
                 << endl;
            npcRed--;
            npcCnt--;
            plyrTrn = true; // npc has a red card and has played one
        }
        else // if it's red and npc does not have a red card
        {
            card = rand() % 4;                          // draw a random card value
            card == 0 ? npcRed++ : card == 1 ? npcBlu++ // ternary operator to translate random card draw
                                : card == 2   ? npcYel++
                                              : npcGrn++;
            npcCnt++; // increment opponent hand count
            cout << endl
                 << endl;
        }
    }
}
```

I created four versions of this snippet for each color case possible for the active card color, and in the example above we take the instance of case 0 which is when the active color is red.

```
    You, 1 hour ago • v3 initialization
if (plyrTrn == false) // if it's still the opponent's turn
{
```

I first created the first wrap if statement to check if it's still the opponent's turn. This is a *dial* that I can turn on and off in order to allow more complex turn sequences in the following and in future implementations to happen.

```
if (npcRed > 0) // nested case if npc also has red cards
{
    cout << "Opponent plays a red card!" << endl
         << endl;
    npcRed--;
    npcCnt--;
    plyrTrn = true; // npc has a red card and has played one
}
```

The next nested if statement is a check to see if the NPC also has a card in hand that matches the current color in play. If so, the NPC then plays the card in hand, and the card color is decremented from their hand as well as the overall opponent card cont. Finally we set the player turn boolean set to true in order to pass the turn back to the human player.

```
else // if it's red and npc does not have a red card
{
    card = rand() % 4;                          // draw a random card value
    card == 0 ? npcRed++ : card == 1 ? npcBlu++ // ternary operator to translate random card draw
                         : card == 2   ? npcYel++
                                       : npcGrn++;
    npcCnt++; // increment opponent hand count
    cout << endl
         << endl;
}
```

If the NPC DOES NOT have a card in hand that is of the same color in play, the NPC then draws a random card.

This is replicated four times for all color cases.

We then must implement the above changes to the player turn logic sequences. The condition in which the player's turn ends is when the correct card color is played.

To do this, we implement the changes as follows:

```
if (menuSel == 'R' || menuSel == 'r') // if user selects either uppercase or lowercase r
{
    if (plyrTrn == true)
    {
        if (redCard > 0 && actCrd == 0) // logic to verify user input and return appropriate responses
        {
            redCard--;
            plyCnt--;
            cout << "You play a red card!" << endl
                 << endl;
            plyrTrn = false;
        }
        else if (actCrd != 0) // if the card selected from menu option does not equal card in play
        {
            cout << "The colors don't match!" << endl
                 << endl;
        }
        else // else user does not own any of the cards selected for play
        {
            cout << "You don't have any RED cards!" << endl
                 << endl;
        }
    }
}
```

First we wrap the turn choices with the base case in which the player turn is still true.

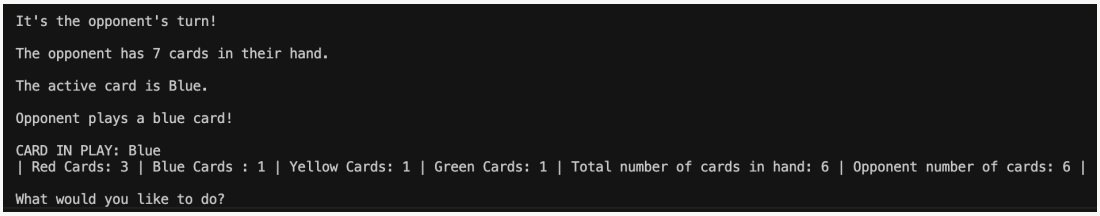
```
if (redCard > 0 && actCrd == 0) // logic to
{
    redCard--;
    plyCnt--;
    cout << "You play a red card!" << endl
         << endl;
    plyrTrn = false;
}
```

The only other change needed is to add the base case of the turn ending which is when the player plays a card that matches the current color in play. Once this happens, set the boolean to false and the next pass will exit the loop.

Command Line Test and Finishing Touches to V3

There were some bugs that I've run into when testing out the V3 program in the command line that were cleaned up first and foremost.

The next step is to properly test the features I've set out to accomplish for this game's newest version.



Above we observe that after our base functions that were displayed in V2, we need to check how the program handles turn switching, and as shown above, this new function properly works.

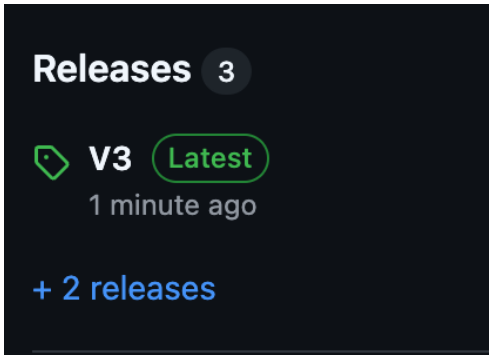
After the player's first turn, the program gives a prompt to list out the opponent's steps. First it lets the user know that the opponent's turn has begun, followed by a display of how many cards the opponent currently has in their hand. The active card color is shown, and afterward the opponent takes a turn according to the card color shown.

The program then loops back to the player's turn and properly finishes the cycle of finally having a fully contained gameplay loop.

Something to note and consider is that the game in the current state does not have any card values and options to change the card color, which in the game of UNO is usually handled by either playing a similar number to the one at play regardless of color suit, OR by using a wild card in order to manually change the color at play.

In the next version we will finally implement the final piece of the puzzle which is handling being able to change the color of the card a play.

> UNO Version 3.0 has now been deployed to GitHub at
>
> https://github.com/Sam-T-G/UNO/releases/tag/v3



https://github.com/Sam-T-G/UNO/releases/tag/v3

Version 1.4

Goals for UNO V4

- Create a wild card in order to bring color changing functionality.
- Fix any bugs in logic that the game may have.
- Clean up UI and potentially reorganize display.
- Add more descriptive steps and options.

Once those goals are reached, we should have a fully functioning beginning to end (simplified) version of UNO.

First Consideration for implementation of wild cards

The idea of adding another card to the mix also brings the idea of card draw probablity. One thing to consider is if we want the likelihood of drawing a wild card to be the same as the other colors.

Upon thorough consideration, we should keep the odds the same as the other cards. To create a minimum viable product, having the odds the same as other cards is imperative for the flow of the game so that card values don't reach astronomical numbers in any one given color if there is a case in which one of the players is very unlucky. Having the player option to swap colors accordingly will smooth out the goal gameplay loop of the game for this first project's iteration.

That being said, an important thing to note is that we also need to program the NPC to behave properly with the introduction of the wild card. Upon careful consideration and for the sake of programming a computer opponent that is designed to be predictable and beatable, I've decided that the program logic should go as follows:

If the computer program does not have any of the current suit colors, it prioritizes playing any wild cards they may have in hand.

> Wild card priority will compare the cards we currently have and assess the quantity in hand. Program will use this information to find the color in which they have the most amount of colors and swap the card at play to that color.

If the computer program does not have any cards that can be played, proceed to drawing a card until they can play one that's viable.

Refer to base logic of playing proper color card → refer to wild card color priority.

We kick off UNO V4 by introducing two new variables to keep track of our fourth card type - the wild card.

```
    wildCrd, // player wild card
    npcWld;  // npc wild card
```

The next step is to change all the logic throughout the current program so that it reflects a random draw between five cards and will add a draw counter to the wild card if it is drawn. We then apply this too all card draw functionality of both player operations as well as NPC.

```
{
    card = rand() % 5;                              // rando
    card == 0 ? redCard++ : card == 1 ? bluCard++ // terna
                          : card == 2   ? yelCard++
                          : card == 3   ? grnCard++
                                        : wildCrd++;
    plyCnt++;
}
```

Implementing the Wild Card Logic

The wild card logic is was pretty straightforward to implement. We begin by creating a menu pathway for the user to select the Wild Card to be put into play.

```
if (menuSel == 'W' || menuSel == 'w') // if user selects either uppercase or lowercase w
{
    if (plyrTrn == true)
    {
        if (wildCrd > 0) // logic to verify user input and return appropriate responses
        {
            wildCrd--; // decrement wild card when used
            plyCnt--;  // decrement player card total when used

            cout << "You play a wild card! What color would you like to swap to?" << endl
                 << endl;
            cout << "| R: Swap Red | B: Swap Blue | Y: Swap Yellow | G: Swap Green | " << endl
                 << endl;
            cin >> menuSel;
            cout << endl;
```

As shown above, if the player has a wild card available, the program will decrement the amount of wild cards and total number of cards from the player's hand then verify the player that they have chose to play a wild card. The program prompts the user to then input what color they would like to swap to.

```
if (menuSel == 'R' || menuSel == 'r')
{
    actCrd = 0;
    actCol = "Red";
}
if (menuSel == 'B' || menuSel == 'b')
{
    actCrd = 1;
    actCol = "Blue";
}
if (menuSel == 'Y' || menuSel == 'y')
{
    actCrd = 2;
    actCol = "Yellow";
}
if (menuSel == 'G' || menuSel == 'g')
{
    actCrd = 3;
    actCol = "Green";
}

cout << "You have used your wild card to change the color to " << actCol << "!" << endl // display the new color
     << endl;

// END TURN ONCE WILD HAS BEEN PLAYED
plyrTrn = false;
```

The program then checks the user's input and appropriately changes the active card value and the active color string content so that it may be relayed to the user in the confirmation text below.

When conversion is complete, end player's turn.

NPC Wild Card Logic

This step is an expansion on the idea used for the player's turn. We need to add in the computer logic in which it should choose to use a wild card when it doesn't have a proper card to play and which color would be beneficial for it to play.

```
}
else if (npcWld > 0) // logic to verify user input and return appropriate responses
{
    npcWld--; // decrement wild card when used
    npcCnt--; // decrement player card total when used

    if (npcRed > npcBlu && npcRed > npcYel && npcRed > npcGrn) // if NPC has the most amount of red cards
    {
        actCrd = 0;
        actCol = "Red"; // npc will then opt to choose red
        plyrTrn = true; // end npc turn
    }
    if (npcBlu > npcRed && npcBlu > npcYel && npcBlu > npcGrn) // if NPC has the most amount of blue cards
    {
        actCrd = 1;
        actCol = "Blue"; // npc will then opt to choose blue
        plyrTrn = true;  // end npc turn
    }
    if (npcYel > npcRed && npcYel > npcBlu && npcYel > npcGrn) // if NPC has the most amount of red cards
    {
        actCrd = 2;
        actCol = "Yellow"; // npc will then opt to choose yellow
        plyrTrn = true;    // end npc turn
    }
    if (npcGrn > npcRed && npcGrn > npcBlu && npcGrn > npcYel) // if NPC has the most amount of red cards
    {
        actCrd = 3;
        actCol = "Green"; // npc will then opt to choose green
        plyrTrn = true;   // end npc turn
    }

    cout << "The opponent plays a Wild Card! The new color is " << actCol << "!" << endl
         << endl;
}
```

The code above is only one instance of the four instances that can exist within the computer's decision making logic. I created multiple logic conditions for every possible color combination and then proceeded to check what the highest card count color is for the computer to determine what color the program should swap to.

In this case, the program checks to see if they have available wild cards, and if so, decrements the available number, then proceeds to change the color to whatever it has most of.

Now that this functionality exists, the game's full base feature and gameplay is complete. The game can now be played from start to finish, and when one player reaches zero cards in hand, the game prompts who the winner is!

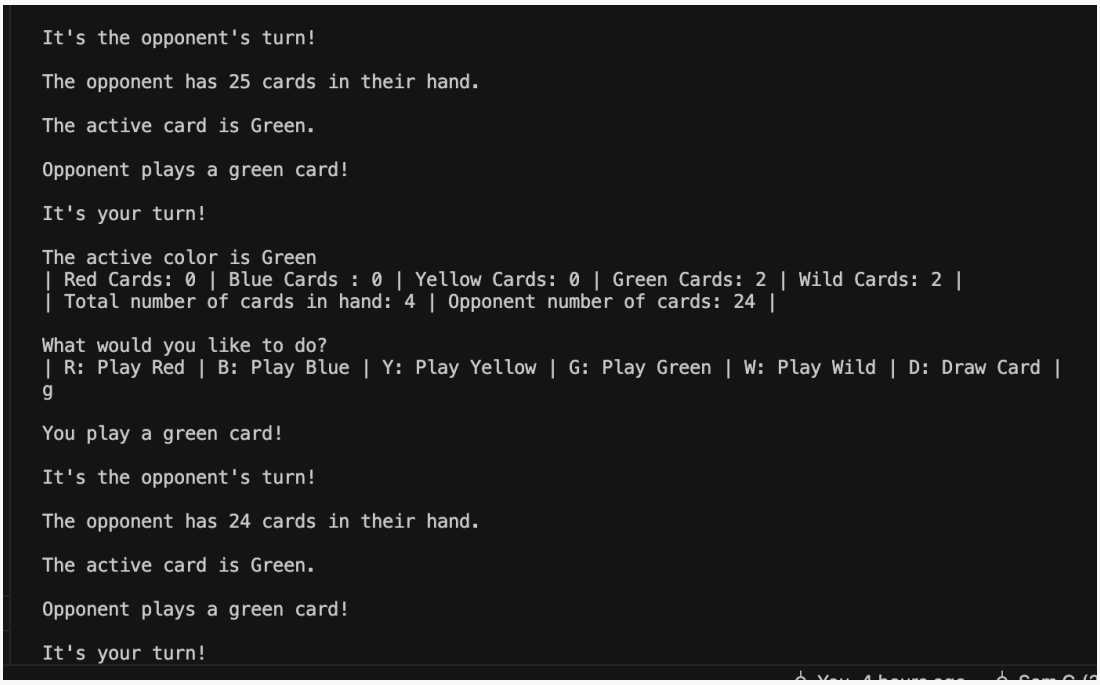
Cleanup

Last but not least is to clean up spacing errors, and adding the proper info when something is done on either the NPC or player side. Being as most concise with every single step and piece of information is important for the human player to make decisions as to what they should do strategically to win.

Final V4 Tests

In the final test, we check to see if the game properly displays every step of the process on both players' behalf.

The game now properly displays all the relevant information that the player needs to make informed decisions.



As we can observe, there are indications as to whose turn it is, a reminder of how many cards the opponent has when beginning their turn as well as the active color, and the decision the opponent makes.

We are then prompted that it's our turn, and the UI properly shows the user's available cards in hand and a tally of how many cards are in each player's hand.

Another example of turn passing is shown towards the end to ensure proper play interaction.

```
y

You have used your wild card to change the color to Yellow!

It's the opponent's turn!

The opponent has 22 cards in their hand.

The active card is Yellow.

Opponent plays a yellow card!

It's your turn!

The active color is Yellow
| Red Cards: 0 | Blue Cards : 0 | Yellow Cards: 0 | Green Cards: 0 | Wild Cards: 1 |
| Total number of cards in hand: 1 | Opponent number of cards: 21 |

What would you like to do?
| R: Play Red | B: Play Blue | Y: Play Yellow | G: Play Green | W: Play Wild | D: Draw Card |
w

You play a wild card! What color would you like to swap to?

| R: Swap Red | B: Swap Blue | Y: Swap Yellow | G: Swap Green |

y

You have used your wild card to change the color to Yellow!

Congratulations, you've won!
```

The above shows the final steps leading to a the end of a full gameplay loop. As we can observe, the player uses a wild card and inputs *y* to choose to swap to a yellow color. During the computer's turn, they respond by putting down the proper card as the active color deems, then passes the turn back to the player.

The player then is shown their hand and only has a single wild card left and plays it by typing
w.

The player then chooses the color yellow to swap to, gets prompted that the active color is now yellow, but it doesn't matter because the hand count reaches zero and the player is quickly prompted that they've won!

> UNO Version 4.0 has now been deployed to GitHub at
>
> https://github.com/Sam-T-G/UNO/releases/tag/v4

https://github.com/Sam-T-G/UNO/releases/tag/v4

Version 2.1

Goals for UNO Version 5

- Implement remaining project library requirements which include: cstring, cmath, fstream, and iomanip.
- Named constant - likely a conversion for percentage.
- Float - likely also for calculating percentage.
- Type Casting - likely to utilize float percentage conversion with an integer value.
- Formatting output - likely in conjunction with iomanip features.
- Math Library
- File input/output using fstream - likely to export a list of scores
- Further UI improvements.
- (Maybe Skip Card)

One implementation I will focus on in Version 5 is to create High Scores list to store the top 5 high scores of any locally run game instance. Implementing this also implies that I must come up with a scoring system. Some ideas include:

- Number of turns it takes to finish a game
- Card number difference when the game is won

Upon thorough consideration, we will go with the latter and score based off of the card difference between the player and NPC. The player will not be able to submit their scores if they fail to succeed.

Things to adjust to allow this implementation:

- The player must input their name when starting the game
- The program needs to only take in a highscores input if and only if the player wins.

Goals, Considerations, and Steps to Create New Functionality

- Create a feature in which the player inputs their name at the very beginning of the game's launch.
- Have the program check to see if the player wins in order to allow the following steps of actually storing a winning score to the highscores page.
- We will use the existing card count variables in order to calculate the card difference between player and computer.
- Using existing information calculate the mean of the top 10 scores, and the standard deviation.
- Display to user the average of all the top 10 scores, then display standard deviation.
- Have a fun improvement metric that displays how far off the player was from the average, if the player is lower than the standard deviation, tell them that there is improvement to be made regardless of the fact that they've reached the top 10, if they are above the standard deviation, tell the player that they are exceptional.
- IMPORTANT: After going over course material, using arrays to store name and score are not within the project 1's bounds so I will just have the program give the player information on the last 10 scores and rank where they stand compared to the last 10 scores regardless of player name submissions.
- Upon more research I will implement the deque library in order to accomplish my program without the use of arrays. The library allows me to manipulate the entries in my document so that I can avoid using arrays or vectors in order to manage my scores list which are locked features I can only implement in project 2.

Create a Player Name Input

```
cout << "Main Menu" << endl
     << endl;
cout << "Type the character that corresponds to your selections within this game" << endl
     << endl;
cout << "| Enter your name to Start the Game |" << endl
     << endl;
cin >> name;
cout << endl;
```

This implementation was very simple, we just create a variable to store the user's name. This makes a modular variable which is set by the player at the beginning of the game.

Check for Player Win and Export Into High Scores Page

This step was relatively straightforward but took a little bit of research for me to set up and implement. The challenge came with the project parameters of not being able to use arrays to store and search through the previous scores documented in our *hiScores.dat* file within the same directory.

To accomplish the functionality we're looking for, we utilize the *deque* library in order to manage the information that we are handling.

```
fstream file("hiScore.dat", ios::in); // initialization of high scores storage
deque<int> scores;                    // Using deque to store scores instead of using arrays

const int maxScrs = 10; // Create an unedtiable ceiling of 10 scores
```

As shown above, we set a constant integer variable to dictate the max amount of scores the document can handle. This allows our *deque* library to create a first-in-first out score handling system so we can create statistical feedback on the last 10 victories against the computer.

```
while (file >> score) // read in scores stored in hiScores.dat
{
    scores.push_back(score); // use push_back function from deque library
}
file.close(); // close file

cout << "Congratulations, you've won!" << endl // display player name if player wins the game.
     << endl;
score = npcCnt - plyCnt; // if player wins calculate score by finding the difference between the opponent card count and player card count
cout << "You've scored " << score << " points!" << endl
     << endl;

scores.push_back(score); // add score to the que

if (scores.size() > maxScrs) // checks to see if the list of scores exceeds the constant limit of 10
{
    scores.pop_front(); // removes the oldest score
}

file.open("hiScore.dat", ios::out); // clear file and write new scores

// write updated scores back to the file
for (int i = 0; i < scores.size(); i++) // loops for the length of size of scores
{
    file << scores[i] << endl;
}

file.close();
```

The code section above shows the process in which the scores are read from the existing *hiScores.dat* file and further processed in order to store the most recent winning score.

We first calculate the score by finding the difference between the opponent card count and the player count (which is zero since the player wins), and relays the information back to the player.

The program then takes the score and pushes it to the back of the que since it's the most recent in a first-in-first-out model. If the number of scores exceeds the constant set at the beginning of the program at a maximum score hold of 10 individual wins, we use the *scores.pop_front()* feature from the *deque* library in order to remove the oldest score inserted.

```
score = npcCnt - plyCnt; // if player wins calculate sco
cout << "You've scored " << score << " points!" << endl
     << endl;

scores.push_back(score); // add score to the que

if (scores.size() > maxScrs) // checks to see if the lis
{
    scores.pop_front(); // removes the oldest score
}

file.open("hiScore.dat", ios::out); // clear file and wr

// write updated scores back to the file
for (int i = 0; i < scores.size(); i++) // loops for the
{
    file << scores[i] << endl;
}

file.close();
```

We then open the *highscore.dat* file in the directory and we clear the previous values stored in order to write the updated scores back into the file using a loop that iterates for the length of the number of scores.

When completed, we close the file.

Calculating the Average of the Past Ten Victories

The process in this next part is very simple, we simply set our sum variable equal to zero, then loop through each stored score and add it back into the sum. We then create an edge case response in which we return the only score on the list if there's only one that exists.

Next we meet of showing type casting competency by changing the the integer value of sum into a float so that we can return a more precise average value in which we store in the float variable of average.

```
// Calculate the average of the stored scores
sum = 0;                                // set the sum equal to 0
for (int i = 0; i < scores.size(); i++) // take each score in the deque
{
    sum += scores[i]; // set sum equal to all of the scores in the deque
}
// Calculate the average
if (scores.size() == 1)
{
    average = static_cast<float>(scores[0]); // If there is only one score, set average to that score
}
else
{
    average = static_cast<float>(sum) / scores.size(); // Calculate average for more than one score
}
```

Various Statistical Features

- Finding Variance and Standard Deviation
  We finally introduce the
  cmath library to aid us in finding both the variance and standard deviation of our scores as shown below

```
// initialize and calculate float variance
varince = 0;
for (int i = 0; i < scores.size(); i++)
{
    varince += pow(scores[i] - average, 2); // squared difference from the mean
}
varince /= scores.size(); // average squared difference

// calculate the standard deviation
stdDev = sqrt(varince); // standard deviation is calculated from taking the square root of variance
```

We calculate the variance by taking the difference between the individual scores and the average and square which then we find the average variance of all of our scores.

We then simply use the
sqrt feature from the cmath library to find the square root of the variance which gives us out standard deviation.

- Percent Change Calculation

We then create a percent change calculation to find the percent difference from what the average of the last scores is. To do this we simply do the following.

```
// Display the Statistics
if (average < score)
{
    btrWrse = "better";
}
else
{
    btrWrse = "worse";
}

// Calculate the percent change from the average
pctChng = (score - average) / static_cast<float>(average) * PERCENT;
if (pctChng < 0)
{
    pctChng *= -1; // if percent change is negative, change to positive
}
```

We create a condition that brings a little bit of user clarity for the information we are trying to provide, which is if the change is better or worse in comparison to the average.

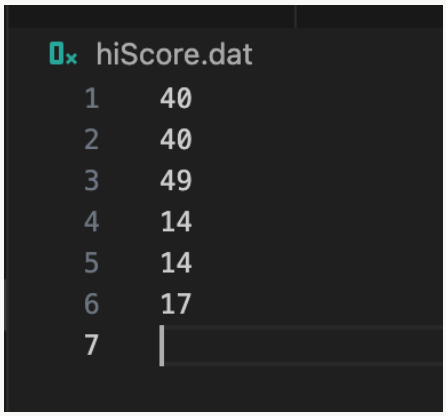
We then calculate the percent change and ensure that it is always a positive value by multiplying -1 if it's less than zero. This brings clarity to the game UI such that the player will be notified if its a positive or negative comparison using the better or worse variable.

- Statistical Features Presented

We wrap our game up by displaying all the features we've implemented. We calculate the percent change and indicate whether or not the player has improved, we show the average of the scores within an adjustable indicator of how many scores were submitted using *scores.size()* and we display the standard deviation of all the previous scores.

```
cout << "Which is " << pctChng << "% " << btrWrse << " than the average of the last ten victories." << endl
     << endl;
cout << "Average of last " << scores.size() << " scores: " << average << endl;      You, 3 minutes ago • Uncommi
cout << "Standard Deviation of the scores: " << stdDev << endl;
```

Here's what the values look like in out *hiScores.dat* and what a victory sequence looks like with all of the math confirmed to calculate accordingly.




```
You play a wild card! What color would you like to swap to?

| R: Swap Red | B: Swap Blue | Y: Swap Yellow | G: Swap Green |

y

You have used your wild card to change the color to Yellow!

Congratulations, you've won!

You've scored 17 points!
Which is 41.3793 % worse than the average of the last ten victories.

Average of last 6 scores: 29
Standard Deviation of the scores: 14.3527
sam@Samuels-MacBook-Pro UNO %
```

> UNO Version 5.0 has now been deployed to GitHub at
>
> https://github.com/Sam-T-G/UNO/releases/tag/v5

https://github.com/Sam-T-G/UNO/releases/tag/v5

Future Implementations for Phase 2

There are several improvements and feature implementations to look forward to in project two which include but are not limited to:

- More organized and compartmentalized code due to capability of using functions and calling the functions multiple times
- Richer sets of data with the capability of using arrays, namely multi-dimensional arrays to store even more sets of information tied to each player submission
- The capability to traverse through the database of player submissions and history of scores of each player using sorts and searches
- Having more card types to increase the gameplay complexity and breadth of strategies that can be implemented
- Introduce numbers alongside color types to enable color switching based off of standard cards and not just wild cards
- Create a richer, more immersive, and succinctly informational user interface to improve the overall gameplay experience

Project Phase 2 Development

Implementing Functions, Arrays, Sort and Search

The goal in phase two of development is to introduce features that will further replicate the mechanics of the UNO game - namely utilizing arrays as a structure in order to enable full intended gameplay functionality.

A major feature we are looking to implement include the introduction of numbers as another dimension of the game's matching progression loop. In the current state of the game, we have four colors that enable card dump. The only way to be able to put down a card currently is to have a card of the same color. Introducing numbers enables an alternative and game altering condition in that a player can change the active color using number matching instead of relying on wild cards.

There are some considerations in the way we want to implement this feature. The current method in tracking cards within each player's hand is to have ten individual variables that keep track of the cards between the player and the NPC. The question becomes -
How do we want to approach adding a second characteristic to each card?

I think the first step is to create a visual table to indicate which cards will be introduced and how the permutations of cards will exist within our game.

Card Variation Table	Red	Blue	Yellow	Green	Wild
	0	0	0	0	Standard
	1	1	1	1	DRAW 2
	2	2	2	2	DRAW 4
	3	3	3	3	
	4	4	4	4	
	5	5	5	5	
	6	6	6	6	

	7	7	7	7	
	8	8	8	8	
	9	9	9	9	
	SKIP	SKIP	SKIP	SKIP	
	DRAW 2	DRAW 2	DRAW 2	DRAW 2	
*future imp.	Reverse*	Reverse*	Reverse*	Reverse*	

We can observe from the table above that there really are only two variables, the types of cards, and the value or function that they hold. From this visual, my first intuition is to create a 2-Dimensional Array, in which we can allocate the same card draw values within the arrays index of i.

- 0 = red
- 1 = blue
- 2 = yellow
- 3 = green
- 4 = wild

We then assign a number in the array's index of j starting from 0 in which:

- 0 = 0
- 1 = 1
- 2 = 2
- 3 = 3
- 4 = 4
- 5 = 5
- 6 = 6
- 7 = 7
- 8 = 8
- 9 = 9
- 10 = SKIP
- 11 = DRAW 2
- 12 = DRAW 4*

We then can add extra logic in which the program can check either color or number in order to enable card play and color swap fluidly.

The question then evolves to:

> *How do we want to display the cards available in the player's hand?*

The current method of hand display is to have four columns that display card colors and how many of each are available. Now that we have two dimensions to quantify a card's value, we may have to critically think about how we would like to convey the necessary information of card color and number value.

> *How can we display cards available with permutations but not bloat the user interface?*

In order to avoid UI bloat, we should only display values of cards that exist in the user's hand, if not, omit the values.

Solution:

The solution I've devised is to display information as follows. We will populate a 2 dimensional array according to the cards available in the user's hand so that cards are organized in columns horizontally that specify the card color, and will populate downwards if a card exists in hand of whatever value the colored card may hold.

Red	Blue	Yellow	Green	Wild
5	DRAW 2	8	6	Standard
SKIP	7	2	9	DRAW 4
	9		4	STANDARD

As cards are either drawn or dealt to the player, they will populate downwards in the UI as follows, which brings us to the next step of creating a functional UI.

Sorting the Cards

We will now use this opportunity to meet our sorting and searching quotas. In order to create a good functioning UI, we should implement the following priorities in order.

1. Have Playable Number Card be Highest Priority
2. Have Playable Function Card be Second Priority
3. Have rest of the cards be sorted in descending numbers from low to high

An example of the above can be visualized as such:

ACTIVE CARD = RED 9

Red	Blue	Yellow	Green	Wild
5	9	2	9	Standard
SKIP	7	8	4	Standard
	DRAW 2		6	DRAW 4

In this example we observe that when the while the active card has a RED and 9 value, the 9 cards in the blue and green column are shown at the top even though they are not the active color which indicates functionality in which the active color can be swapped if played those cards.

LOGIC AND OTHER CONSIDERATIONS

- If a number is placed in play, set the active color equal to card's color.
- Added draw functions will also skip the opponent's turn.
- WILD DRAW 4 will simultaneously change active color, make the opponent draw four cards, then skip the opponent's turn
    - Sort logic will be as follows

      - If active card value is equal to a card value within the j index, sort so that they have a lower vertical index

      - for loop to swap index of cards around depending on active card in play
- Properly compartmentalize existing logic to individual functions to be called:
    - Card Draw
    - Logic Check of Active card
    - NPC Logic compartmentalization

Version 2.1

Goals for UNO V2

- Create functions to streamline existing functionality

First and foremost, the first goal in version 6 of our game is to rewrite ALL of the code so that it can now finally utilize functions. These will be MONUMENTAL in creating modularity and essentially cutting down all of our currently existing code in half since a lot of the logic can be compartmentalized to specific functions that we can call when needed.

Streamlining Existing Repetitive Code Into Functions

Starting off, we will create various functions in order to handle card draw, base logic in determining active card color, and user menu selection.

```
// Function Protypes
void draw();    // function to draw a card and place into hand
void actvCrd(); // function to handle active card logic
void menuSel(); // handles menu selection
```

Re-Imagination of Card Draw System

The implementation of a new card draw system was very challenging to mentally tackle and construct. Referring to the pseudo table I had before was very important on taking on this challenge in which the goal is to represent the following table in a two dimensional array:

Red	Blue	Yellow	Green	Wild
5	DRAW 2	8	6	Standard
SKIP	7	2	9	DRAW 4
	9		4	STANDARD

In order to achieve the above I had to create a draw function that was modular and can be used and called in multiple nuanced cases such as when the user voluntarily opts to draw, when the opponent is resorted to having to draw, and when either player plays a forced draw card. The goal is to use this modular function in order to streamline multiple scenarios and have other independent functions to call on this draw feature to return a randomized card.

To accomplish what has been stated above, we start with an entirely new program to wipe our palette clean of code bloat to create the following:

```
// draw function - pass in copy of pyrTrn, pass by reference hand count of player and npc
void draw(int drwHnd[][MAXCARDS])
{
    int drwVal;              // initialize a value of card - i index
    int drwCol = rand() % 5; // randomize a color drawn - j index
    if (drwCol == 4)
    {
        drwVal = rand() % 3; // if wild card, choose from only three options - standard, dr
    }
    else
    {
        drwVal = rand() % 13; // if not wild card, choose from 12 options
    }
    if (drwHnd[drwVal][drwCol] < MAXCARDS) // pass in index value of color in first array,
    {
        drwHnd[drwVal][drwCol]++; // increment the value found at given coordinate
    }

    // VISUAL MATRIX DEBUG
    for (int i = 0; i < 13; i++) // loop i for max length of card values = 13
    {
        for (int j = 0; j < 5; j++) // loop j for max height of color types = 5
        {
            cout << drwHnd[i][j]; // display value at given matrix coordinate
        }
        cout << endl; // move to next line when row is filled
    }
}
```

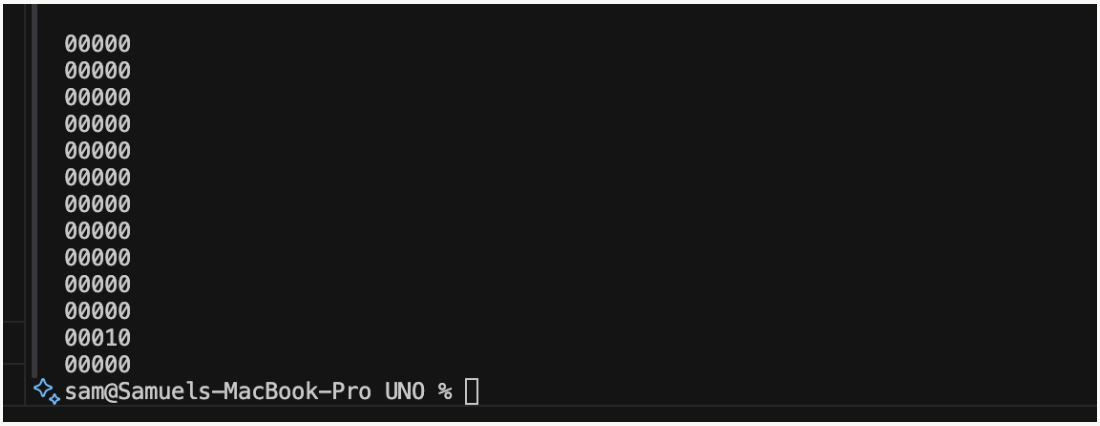
We initialize two variables - the color and value of a single card that is drawn.
We first start with determining the color drawn which is necessary to cover the edge case of drawing a wild card since there are only three values attached to the wild card being *standard, DRAW 2,* and *Draw 4.*

```
if (drwCol == 4)
{
    drwVal = rand() % 3; // if wild card, choose from only three options - sta
}
else
{
    drwVal = rand() % 13; // if not wild card, choose from 12 options
}

drwHnd[drwVal][drwCol]++; // increment the value found at given coordinate
```

We process the edge case of a wild being drawn, which we will only assign between three varying functions. If the card drawn is a standard color, we will assign the standard value range.

One important immediate change I made was the decision to move the overflow check of MAXCARDS elsewhere instead of having the function check if there is an overflow within the draw function itself.

As a check, we now will display our matrix to confirm our matrix works and move on the the next phase of development. The image below shows an incremented value at the coordinates of our matrix. In translation, this represents that the card drawn is a green DRAW 2.

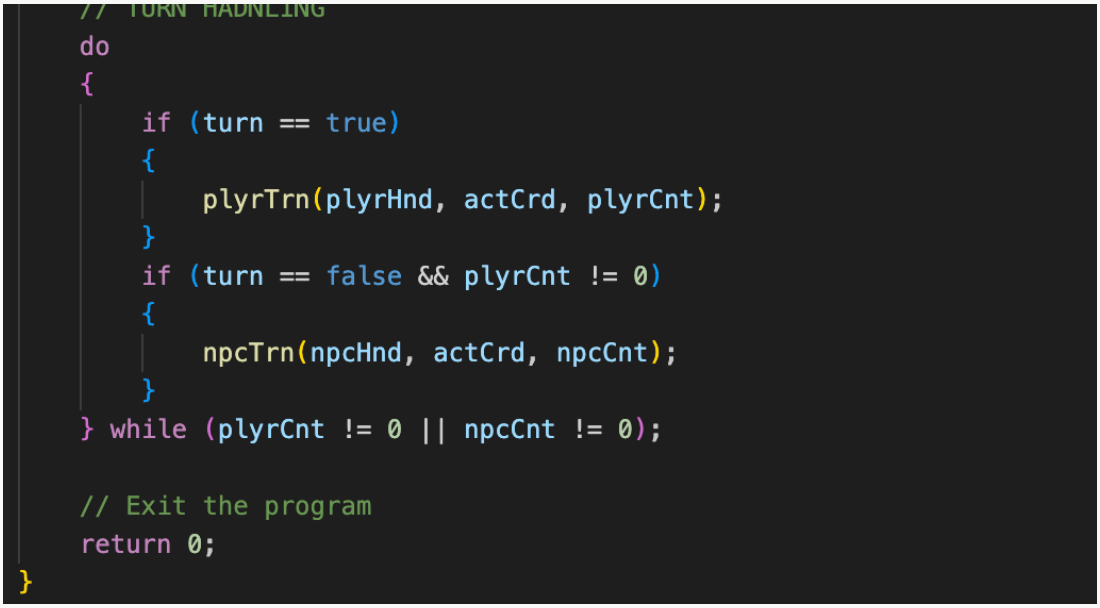


I've realized that the maximum variations for the first implementation will only cover 11 values, so the ranges written up to this have been adjusted accordingly.

Reconstructing Turn Handling

The next step in rewriting our program is to think about the way we want to compartmentalize and handle player and NPC turns within our program.

The current method in having a boolean of *plyrTrn* is still fine but we will change the name to now simply *turn.* The next step is to have our programs branch out properly and efficiently for readability and modularity when considering future development.



```
    // TURN HADNLING
    do
    {
        if (turn == true)
        {
            plyrTrn(plyrHnd, actCrd, plyrCnt);
        }
        if (turn == false && plyrCnt != 0)
        {
            npcTrn(npcHnd, actCrd, npcCnt);
        }
    } while (plyrCnt != 0 || npcCnt != 0);

    // Exit the program
    return 0;
}
```

This new framework allows us to compartmentalize our functions properly and have decisions made remotely within the new player and npc turn functions.

Initial Deal of Cards on Game Start

In spirit of cleaning up and compartmentalizing main, we can move the function of dealing the initial hand of cards and determining the starter card into it’s own compartmentalized function and call the draw function to properly initialize the game.

This next segment actually took me about 4 hours to fully realize and implement, but the general structure has now been created in order to properly have a modular draw function that can be called. This process took up lots of time because I had to keep reinventing the way in which I wanted to compartmentalize snippets of logic.

For example, I began with having two separate functions for handling the player card draw and the NPC card draw. I realized that having a singular function whilst being modular was likely better practice, but with that adjustment, I had to nearly start from square one because the handful of variables that I passed into each function would change. This then became a perpetual issue when I arrive at a crossroads and think about a more optimal way to organize and manipulate data, but these are all just valuable lessons and live practice on how I should process the information that’s important. Nevertheless here is what I’ve engineered thus far.

Draw Function

The first major modular piece I had to create was the draw function. This function needed to be modular in a way that enabled me to call within other functions such as player or NPC opted draws, the initial dealing of cards, and forced card draws when either the player or NPC play a DRAW 2 or DRAW 4 card.

The main feature in this draw function was trying to navigate around chronological priorities when assigning the randomization of a normal card vs a wild card. Normal colored cards draw from an index of 0-11 to determine whether it has a number, a SKIP, or a DRAW 2 attribute. A wild card can only draw from an index of 0-2 because the attributes it can draw from are a regular WILD CARD, DRAW 2 WILD CARD, and DRAW 4 WILD CARD.

```
// draw function - pass in copy of pyrTrn, pass by reference hand count of player
void draw(int drwHnd[][ROW])
{
    int drwVal;              // initialize a value of card - i index
    int drwCol = rand() % 5; // randomize a color drawn - j index
    if (drwCol == 4)
    {
        drwVal = rand() % 3; // if wild card, choose from only three options - sta
    }
    else
    {
        drwVal = rand() % 12; // if not wild card, choose from 11 options
    }

    drwHnd[drwVal][drwCol]++; // increment the value found at given coordinate

    // VISUAL MATRIX DEBUG
    for (int i = 0; i < 12; i++) // loop i for max length of  = 5
    {
        for (int j = 0; j < 5; j++) // loop j for max height of color types = 12
        {
            cout << drwHnd[i][j]; // display value at given matrix coordinate
        }
        cout << endl; // move to next line when row is filled
    }
    cout << endl;
}
```

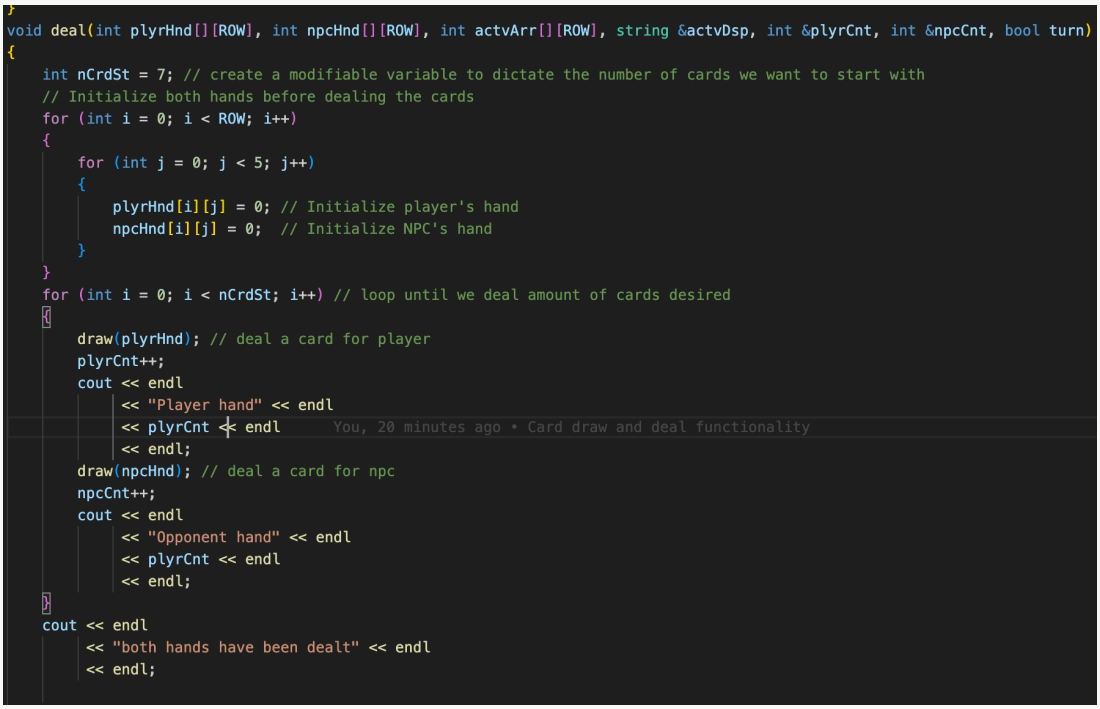
Although the code above is simple and straightforward, it took me quite a while to engineer it - having to think of all the possibilities and nuances of how the game should draw a card,

what variables it should manipulate, and what functions and how a function would pass in variables to manipulate.

Deal Function

The deal function and the draw function go hand in hand in creatively thinking on how I wanted the program to handle manipulated variables, as the deal function was actually very valuable in terms of testing the modularity of the draw function.

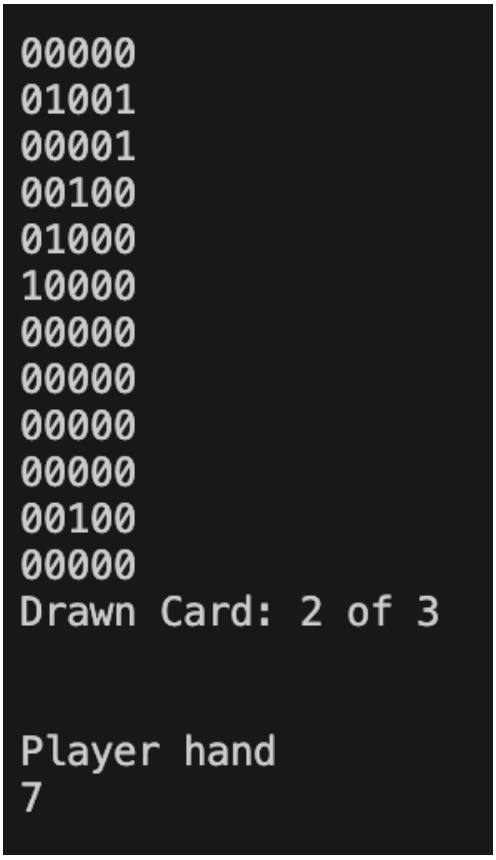
For example, when calling the draw function, I initially had the draw function read in whether or not it was the player's turn, and have branching *if* conditions to process the card draw, but had to completely rethink my approach when thinking through the deal function.

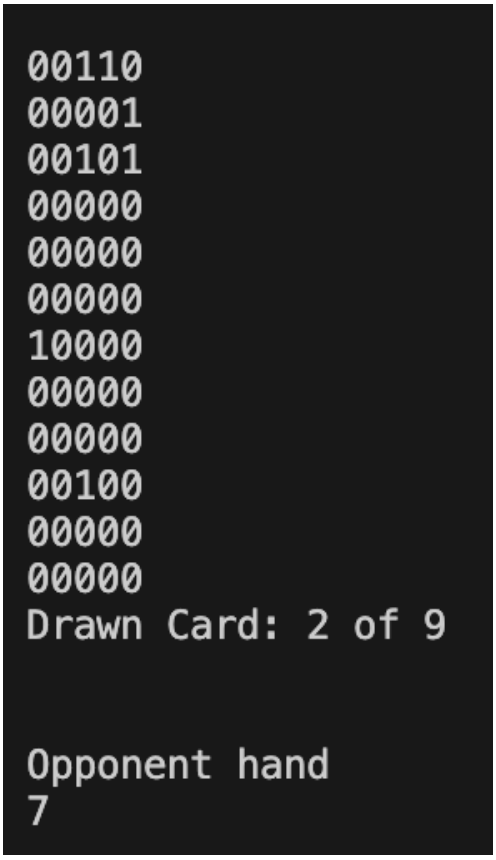


```
}
void deal(int plyrHnd[][ROW], int npcHnd[][ROW], int actvArr[][ROW], string &actvDsp, int &plyrCnt, int &npcCnt, bool turn)
{
    int nCrdSt = 7; // create a modifiable variable to dictate the number of cards we want to start with
    // Initialize both hands before dealing the cards
    for (int i = 0; i < ROW; i++)
    {
        for (int j = 0; j < 5; j++)
        {
            plyrHnd[i][j] = 0; // Initialize player's hand
            npcHnd[i][j] = 0;  // Initialize NPC's hand
        }
    }
    for (int i = 0; i < nCrdSt; i++) // loop until we deal amount of cards desired
    {
        draw(plyrHnd); // deal a card for player
        plyrCnt++;
        cout << endl
             << "Player hand" << endl
             << plyrCnt << endl      You, 20 minutes ago • Card draw and deal functionality
             << endl;
        draw(npcHnd); // deal a card for npc
        npcCnt++;
        cout << endl
             << "Opponent hand" << endl
             << plyrCnt << endl
             << endl;
    }
    cout << endl
         << "both hands have been dealt" << endl
         << endl;
```

As we can observe above, the values that the single deal function are passed in and actively utilize include the arrays of the player and npc's hands as well as the active card, a string to translate the active card array into a readable string output to the user, the counts of both the player and npc hands, and a boolean to manipulate whose turn it is.

We create a modifiable variable within the function that enables us to change the starting dealt amount of cards if we desire, initialize both hands, then loop and deal the cards according to how many cards are dictated to be in the first hand. We call the draw function and pass in the array of both players' hands to keep track of how many and what cards exist in each 2D array, we then increment the hand count of both the player and the NPC for UI informational purposes.





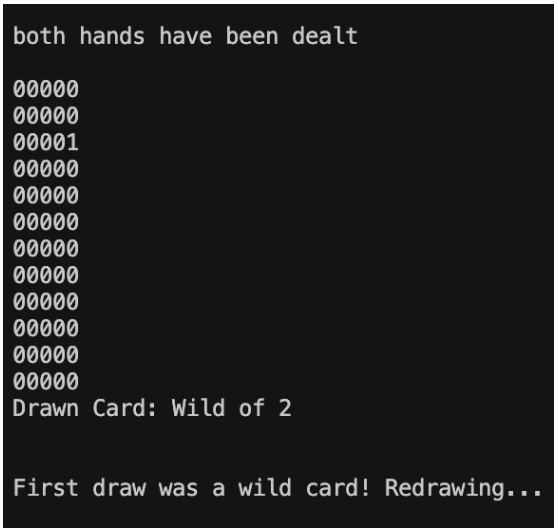
> Above we show the debug lines that calculate all of the underlying logic and information being stored. The table shows a two dimensional array with columns indicating color, and rows indicating what card value. The numbers are incremented whenever a card is drawn at a given coordinate. We also track how many cards are in each player's hand.

Wild Card on First Deal Edge Case

Within the deal function, we address an edge case in which the first card that the program draws is a wild card. In UNO, the first card that is put into play must have a card value, so we approach this by creating a boolean do while loop to make sure that the first card in play is appropriate. We set the boolean to false so that future loops will reset the check, manually reset the active card 2D array, then pass the empty array into the draw function. We check only the fifth column to see if the card is a wild card, and if so, we have a prompt to show that the initial card is a wild card and we redraw the card until we get a color card.

```
    bool wldFrst = false; // initialize a boolean to signify if the first active card is a wild
    do
    {
        wldFrst = false; // initialize a boolean to signify if the first active card is a wild
        // reset active card value
        for (int i = 0; i < 12; i++)
        {
            for (int j = 0; j < COL; j++)
            {
                actvArr[i][j] = 0;
                // cout << endl
                //      << actvArr[i][j] << endl // debug line
                //      << endl;
            }      You, now • Uncommitted changes
        }
        draw(actvArr); // draw a card and set it to the active card
        // check to see if the active card is a wild card
        for (int i = 0; i < 12; i++)
        {
            if (actvArr[i][4] != 0)
            { // Only check the wild row
                cout << endl
                     << "First draw was a wild card! Redrawing..." << endl
                     << endl;
                wldFrst = true;
            }
        }
    } while (wldFrst == true);

    // call active card function to interpret the active card
    actvCrd(actvArr, actvDsp);

    cout << "You and the opponent have recieved your starting hands." << endl
         << "The first active card is " << actvDsp << endl;
}
```

We then create and use an active card function to translate the card into an appropriate string but first we test our program to see if the command line unveils the proper functionality.



> Above we observe that once both hands are dealt, the active card function draws the first card which happens to be a wild of two, which translates into a DRAW TWO wild card. The program then loops to find an appropriate card to set as the initial active card.

Active Card Function

Another vital modular function we need to create is one that translates the matrix of the 2D arrays into two string arrays of colors and values for UI to display. We simply create two separate arrays of colors and value which we pass in every possible it's at this point where I've made the decision to not include WILD variations of DRAW 2 and DRAW 4, as having those introduced at this point will extend games unnecessarily. We will keep the random drawn values of 0-2 wild because it will aid in the overall odds of drawing a wild card to 3 / 11 chances. We return once the card has been properly found.

```
void actvCrd(int actvArr[][COL], string &actvDsp)
{
    // create string array to describe the columns - displaying card colors
    string colors[] = {"Red", "Blue", "Yellow", "Green", "Wild"};
    // create string array to describe the row values
    string values[] = {"0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "SKIP", "DRAW 2"};
    actvDsp = ""; // clear the active display
    // determine active card
    for (int i = 0; i < 12; i++)      // Iterate through Card Types
        for (int j = 0; j < COL; j++) // iterate through card colors
        {
            if (actvArr[i][j] != 0) // iff this slot has a card
            {
                if (actvArr[i][j] != 0) // If this slot has a card
                {
                    // Handle wild cards separately
                    if (j == 4)
                        actvDsp = "Wild";
                    else
                        actvDsp = values[i] + " " + colors[j];

                    return; // Stop searching after finding the first card
                }
            }
        }
}
```

Creating a Player Hand UI

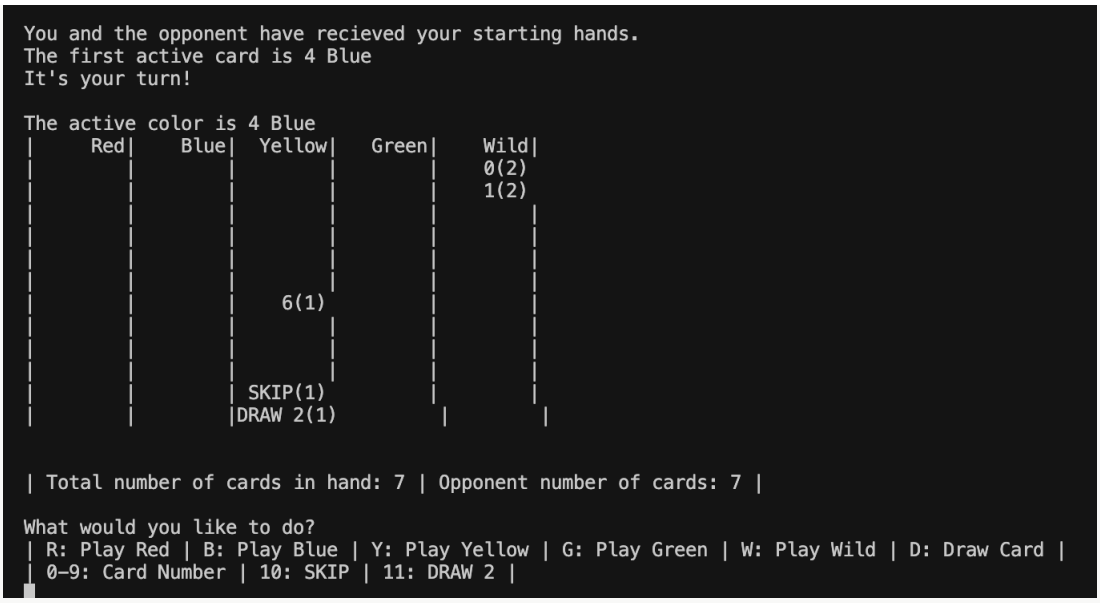
This will be similar to the active card function in which we create two arrays as sort of a key to navigate and translate our 2D arrays. We first create a proper UI using a simple for loop to populate the top into columns to organize our cards.

```
void dispHnd(int plyrHnd[][COL], string &actvDsp, int ROW)
{
    // Create an array of strings for the card colors
    string colors[] = {"Red", "Blue", "Yellow", "Green", "Wild"};
    // Create an array of strings for the card values
    string values[] = {"0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "SKIP", "DRAW 2"};

    cout << fixed << "|";
    for (int j = 0; j < COL; j++)
    {
        cout << setw(8) << colors[j] << "|"; // Print color columns
    }
    cout << endl;
```

Next we loop through our 2D array so that we can print the card values that exist within the proper columns.

```
for (int i = 0; i < ROW; i++)
{
    cout << "|"; // Start a new row      You, 11 minutes ago • Uncommitted changes

    for (int j = 0; j < COL; j++)
    {
        if (plyrHnd[i][j] != 0) // If there are cards in this slot
        {
            cout << setw(5) << values[i] << "(" << plyrHnd[i][j] << ") "; // Print the value and how many of that card exist
        }
        else
        {
            cout << setw(8) << " " << "|"; // No card in this slot, leave it blank
        }
    }
    cout << endl; // New line after each row of cards
}

cout << endl; // Additional space after displaying the hand
```

Our updated UI finally looks like a proper UNO game.
All the information now exists for the player to properly view all the elements within the game: Whose turn it is, the active color, what their hand consists of, and a live reading on how many cards exist in both hands.



Here's where i get very picky with my work. I want to create a better display that:

- Shows multiple of a card instead of showing the amount within the index of each coordinate
- Have it populate from the top
- eliminate unnecessary space

But perhaps after trying to tackle this for about an hour now, this is outside the scope of what I need to accomplish within the current stage of this project. We shall table this implementation to future development.

For now, I'll just clean up spacing with the UI so that it's prettier to look at.

```
The first active card is 0 Green
It's your turn!

The active color is 0 Green
_____________________________________________
|     Red|    Blue|  Yellow|   Green|    Wild|
_____________________________________________
|        |        |        |        |   0(2)|
|        |        |        |        |       |
|        |        |        |        |   2(2)|
|        |        |        |    3(1)|       |
|    4(1)|        |        |        |       |
|        |        |        |        |       |
|        |        |        |        |       |
|        |        |        |        |       |
|        |        |        |        |       |
|        |    9(1)|        |        |       |
|        |        |        |        |       |
|        |        |        |        |       |
_____________________________________________

| Total number of cards in hand: 7 | Opponent number of cards: 7 |

What would you like to do?
| R: Play Red | B: Play Blue | Y: Play Yellow | G: Play Green | W: Play Wild | D: Draw Card |
| 0-9: Card Number | 10: SKIP | 11: DRAW 2 |
```

Handling Player turn choices

The next step is to create logic that enables the player to interact based off of the card in play. Admittedly, this next segment took an incredible amount of a planning, self research, and time investment in order to properly execute.

I've decided to create a base UI function to handle all the re-printing of the displays so that I can just call on the function and have it retrieve all the necessary real-time information for the user to see and play off of.

```
void usrInt(int plyrHnd[][COL], int npcHnd[][COL], int actvArr[][COL], string &actvDsp, int &plyrCnt, int &npcCnt, int ROW, bool &turn)
{
    cout << "It's your turn!" << endl
         << endl;
    cout << "The active color is " << actvDsp;
    cout
        << endl;
    dispHnd(plyrHnd, actvDsp, ROW);
    cout << endl
         << "| Total number of cards in hand: " << plyrCnt << " | Opponent number of cards: " << npcCnt << " |" << endl
         << endl;

    // Prompt for how player wants to proceed
    cout << "What would you like to do?" << endl;
    cout << "| R: Play Red | B: Play Blue | Y: Play Yellow | G: Play Green | W: Play Wild | D: Draw Card | " << endl;
    cout << "| 0-9: Card Number | 10: SKIP | 11: DRAW 2 | " << endl;
}
```

To accomplish this, I put all of the necessary elements of the UI in its own function of *userInt* which stands for user interface.

Then began development of what turns out to be the longest I've spent on any single feature thus far - handling the player input logic and having it interact with both the card and the color indexes. In retrospect, this was the most valuable programming and logic development experience I've had so far, as it tested me on my logical planning prioritization.

```
}

void plyrTrn(int plyrHnd[][COL], int npcHnd[][COL], int actvArr[][COL], string &actvDsp, char colSel, int numSel, int &plyrCnt, int &npcCnt
{
    bool vldPly = false;

    while (!vldPly) // Loop until a valid play is made
    {
        usrInt(plyrHnd, npcHnd, actvArr, actvDsp, plyrCnt, npcCnt, ROW, turn);
        cout << "Choose a card color!" << endl;
        cin >> colSel;
        if (cin.fail())
        {
            cin.clear();
            cin.ignore(1000, '\n');
            cout << "Invalid input! Try again.\n";
            continue;
        }       You, 1 second ago • Uncommitted changes
        cout << "Choose a Value" << endl;
        cin >> numSel;
        if (cin.fail())
        {
            cin.clear();
            cin.ignore(1000, '\n');
            cout << "Invalid input! Try again.\n";
            continue;
        }
        // Handle drawing a card
        if (colSel == 'D' || colSel == 'd')
        {
            cout << "You've drawn a card!" << endl;
            draw(plyrHnd, ROW); // draw randomized card value
            plyrCnt++;          // increment player hand count
            colSel = 0;         // empty value
            numSel = 0;         // empty value
        }
```

This took many iterations and restarts, but I've come to realize that prioritizing the planning of overarching win conditions and immediately addressing special cases is something I should do in all my future developments. I ended up creating a boolean to keep track of whether or not the selection permutations during the player's turn were valid. I call the user interface to start, then check the user inputs to see if the color and value chosen are valid, if now we begin the while loop again.

```
else
{
    int colIdx = getCol(colSel); // create a colindex int variable to retrieve if color selected is viable
    while (colIdx == -1)         // if not viable
    {
        dispHnd(plyrHnd, actvDsp, ROW);                    // display hand UI
        cout << "Invalid color selection! Choose again: "; // prompt user selection is invalid
        cin >> colSel;                                     // input a new color selection
        colIdx = getCol(colSel);                           // re-index and check loop again
    }

    // Handle wild card logic
    if (colSel == 'W' || colSel == 'w')      You, 47 seconds ago • Uncommitted changes
    {
        if (plyrHnd[numSel][colIdx] > 0)
        {
            wldPlay(plyrHnd, actvArr, actvDsp, colIdx, numSel, ROW, turn);
            plyrCnt--;     // decrement hand count
            vldPly = true; // confirm as valud play and continue
        }
        else
        {
            cout << "You don't have any Wild cards!";
        }
    }

    // Regular card play logic
    if (numSel >= 0 && numSel <= 11) // Check if number is within range
    {
        if (plyrHnd[numSel][colIdx] > 0)
        {
            play(plyrHnd, actvArr, actvDsp, colIdx, numSel, ROW, turn);
            vldPly = true; // Exit loop after a successful play
        }
        else
        {
            cout << "You don't have that card! Choose again: ";
            cin >> colSel >> numSel; // Re-prompt
        }
    }
    else
    {
        cout << "Invalid card number! Choose again: ";
        cin >> colSel >> numSel; // Re-prompt
    }
}
```

This next segment of the program is where a lot of meticulous planning had to take part. I had to make sure that the information displayed to the user UI was in a string format for readability and ease of use/input, so i created a get color function that I can call every time a color needs to be translated from the index of 0-4 into their respective card values.

```
int getCol(char colSel)
{
    switch (colSel)
    {
    case 'R':
    case 'r':
        return 0; // Red
    case 'B':
    case 'b':
        return 1; // Blue
    case 'Y':
    case 'y':
        return 2; // Yellow
    case 'G':
    case 'g':
        return 3; // Green
    case 'W':
    case 'w':
        return 4; // Wild
    default:
        cout << "Invalid color!" << endl;
        return -1;
    }
}
```

Next we write logic in order to address our Wild Card handling, and to do this we write a separate function for for the sake of readability. To do this, we first start by creating a remote wild function for the sole purpose of translating a chosen wild card selection into readable string text to prompt the player.

```
// function to process WILD card color change
void wild(int actvArr[][COL], int ROW, int &j)
{
    actvArr[0][j] = 1; // Set wild card action to the active card
    string newCol;
    if (j == 0)
    {
        newCol = "RED";
    }
    else if (j == 1)
    {
        newCol = "BLUE";
    }
    else if (j == 2)
    {
        newCol = "YELLOW";
    }
    else if (j == 3)
    {
        newCol = "GREEN";      You, 15 minutes ago • Uncommitted changes
    }
    cout << "Wild card has been played! The new color is " << newCol << "!" << endl;
}
```

We then create a remote function for wild card prompting and selecting for the user under the function *wldPlay*. We initialize a color select and a color index in order to keep track of a char input from the user for what color they would like to select, and in index that we then translate the char value to integer for the program to be able to search through the 2D arrays.

```
void wldPlay(int plyrHnd[][COL], int actvArr[][COL], string &actvDsp, int colIdx, int numSel, int ROW, bool &turn)
{
    char colSel; // initialize user input of color char
    cout << "You played a Wild Card! Choose a color to swap to (R = Red, 2 = Blue, Y = Yellow, G = Green): " << endl;
    cin >> colSel; // choose color
    colIdx = 0;
    colIdx = getCol(colSel); // fetch color index

    cout << "Choose which wild card slot!" << endl; // validate correct wild index slot
    cin >> numSel;                                  // choose wildcard index slot
    while (plyrHnd[numSel][colIdx] == 0)            // Check if player has the card
    {
        cout << "Invalid input! Make sure you choose the right WILD index from your hand 0-2" << endl;
        cin >> numSel;
    }

    while (colIdx < 0 || colIdx > 3)
    {
        cout << "Invalid input! Choose a valid color (R = Red, B = Blue, Y = Yellow, G = Green): " << endl;
        cin >> colSel;
        colIdx = getCol(colSel); // fetch color index
    }

    // Remove the wild card from player's hand
    plyrHnd[numSel][colIdx]--;

    // Clear the active array2
    for (int i = 0; i < ROW; i++)
        for (int j = 0; j < COL; j++)
            actvArr[i][j] = 0; // Reset active card display

    // Set the chosen color as active
    actvArr[0][colIdx] = 1; // Assign the wild card to the new color
    actvDsp = (colIdx == 0) ? "Red" : (colIdx == 1) ? "Blue"
                                    : (colIdx == 2)   ? "Yellow"
                                                      : "Green";

    cout << "Wild card has been played! The new color is " << actvDsp << "!" << endl;

    turn = false; // End player's turn
}
```

We then create two separate while loops to validate whether or not the correct wild card index has been chosen(later useful for WILD DRAW implementations), and one to check if the target swap color input is also valid. If both conditions are met, we decrement the amount of cards that exist at the given coordinate of the 2D array, set the active card array to empty, then reset the active card to be the color chosen by the user.

```
void play(int plyrHnd[][COL], int actvArr[][COL], string &actvDsp, int colIdx, int numSel, int ROW, bool &turn)
{
    int color = 0, number = 0;

    // Find the currently active card
    for (int i = 0; i < ROW; i++) // Iterate over columns (card numbers)
    {
        for (int j = 0; j < COL; j++) // Iterate over rows (colors)
        {
            if (actvArr[i][j] > 0) // Active card found
            {
                color = j;  // Store the active card's color
                number = i; // Store the active card's number
            }
        }
    }

    // Verify if the card can be played
    if (colIdx == color || numSel == number) // Matches either color or number
    {
        if (plyrHnd[numSel][colIdx] > 0) // Check if player has the card
        {
            plyrHnd[numSel][colIdx]--; // Remove card from player's hand
            cout << "Card played!" << endl;

            // Clear the active array
            for (int i = 0; i < ROW; i++)
                for (int j = 0; j < COL; j++)
                    actvArr[i][j] = 0;

            // Set the active card
            actvArr[numSel][colIdx] = 1;
            actvCrd(actvArr, actvDsp, ROW); // Update the active card display

            cout << "You've played a " << actvDsp << endl;
            turn = false; // Switch to NPC turn
        }
        else
        {
            cout << "You don't have that card!" << endl;
        }
    }
    else
    {
        cout << "Neither the color nor the number matches!" << endl;
    }
}
```

The next function we've created remotely is the play validation. This function ensures that the card chosen by the user is valid before proceeding with the rest of the turn. The function creates a color and number int variable which then it searches through the 2D array in order to find the currently active card. We then store the coordinates within the variables and later checks to see if EITHER our selected color index and number selected match with the active card. If so, we proceed with the checks, and if failed, we prompt the user that neither color or number matches.

THIS IS IMPORTANT for UNO because the game fundamentally relies on the user being able to play matching numbers in order to change the color variable in play.

We then have another check in the internal if statement that checks if the player also

actually has the card within their hand, and if so we decrement the value within the 2D array at the given coordinate, clear the active array, and set the card played as the active card.

NPC AI Logic

This next segment has mostly been scripted in the the first phase of development, the only difference now is that we need to adapt the logic so that it also has a priority of checking the values as well in order to play the appropriate card.

Luckily we've written and thought out a lot of the logic needed during our phase 1 of development, but back then we were limited to not being able to use more advanced techniques.

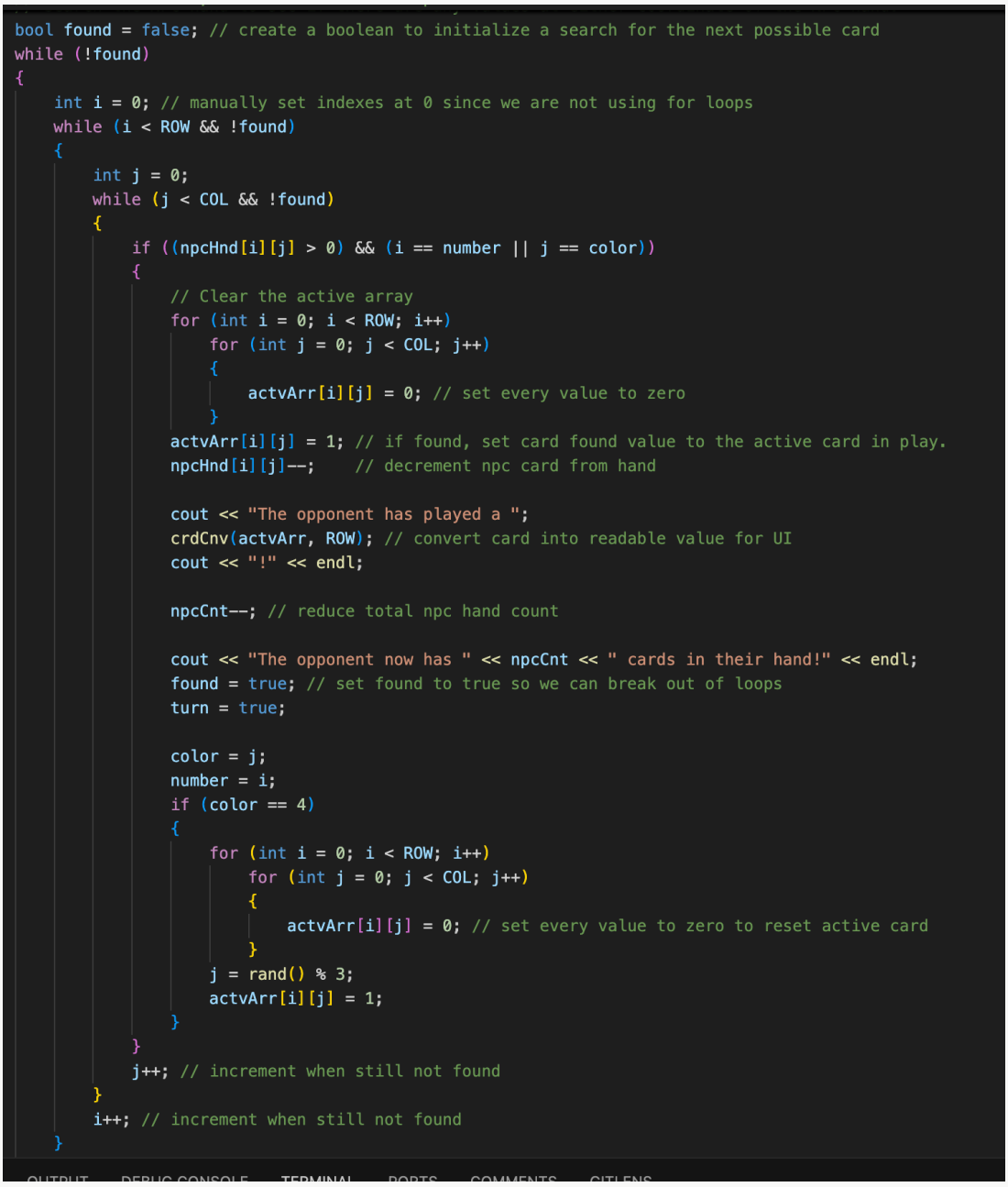
We first use a simple linear search to look through our stored active card array and set them to the variables of color and number to have two separate indexes that create a set of coordinates.

```
// npc AI logic
void npcTrn(int plyrHnd[][COL], int npcHnd[][COL], int actvArr[][COL], strin
{
    int color = 0, number = 0; // initialize color and number to 0

    // Find the currently active card
    for (int i = 0; i < ROW; i++) // Iterate over columns (card numbers)
    {
        for (int j = 0; j < COL; j++) // Iterate over rows (colors)
        {
            if (actvArr[i][j] > 0) // Active card found
            {
                color = j;  // Store the active card's color
                number = i; // Store the active card's number
            }
        }
    }
```

We then wrap the rest of the NPC logic around a boolean so that the NPC will continue to attempt and make a valid play after they draw cards.

I've decided to use while loops so that I can validate whether or not a card has been found at every step of the way. This next check actually took a lot of mental gymnastics for me to come up with but the inner if loop properly checks to see if either index of the NPC hand matches the number OR color of the active card.

The npc needs to be able to know that they can actively change the color of the active card by matching the number and manually changing the color themselves.



```
bool found = false; // create a boolean to initialize a search for the next possible card
while (!found)
{
    int i = 0; // manually set indexes at 0 since we are not using for loops
    while (i < ROW && !found)
    {
        int j = 0;
        while (j < COL && !found)
        {
            if ((npcHnd[i][j] > 0) && (i == number || j == color))
            {
                // Clear the active array
                for (int i = 0; i < ROW; i++)
                    for (int j = 0; j < COL; j++)
                    {
                        actvArr[i][j] = 0; // set every value to zero
                    }
                actvArr[i][j] = 1; // if found, set card found value to the active card in play.
                npcHnd[i][j]--;    // decrement npc card from hand

                cout << "The opponent has played a ";
                crdCnv(actvArr, ROW); // convert card into readable value for UI
                cout << "!" << endl;

                npcCnt--; // reduce total npc hand count

                cout << "The opponent now has " << npcCnt << " cards in their hand!" << endl;
                found = true; // set found to true so we can break out of loops
                turn = true;

                color = j;
                number = i;
                if (color == 4)
                {
                    for (int i = 0; i < ROW; i++)
                        for (int j = 0; j < COL; j++)
                        {
                            actvArr[i][j] = 0; // set every value to zero to reset active card
                        }
                    j = rand() % 3;
                    actvArr[i][j] = 1;
                }
            }
            j++; // increment when still not found
        }
        i++; // increment when still not found
    }
```

The rest of the checks and actions are pretty standard past this. We continue to display all of the proper checks of cards being played, turn the switches on all the proper booleans, and increment the cards properly when drawn.

One difference above though is that we decide to implement the wild card action by checking if the card played after it has been selected by the NPC. This ensures that the NPC prioritizes playing a valid card if available first, then chooses the Wild Card if none available. We also randomize the color chosen and set the new color.

```
        // while STILL not found any matches, prompt the enemy to draw cards
        if (!found)
        {
            draw(npcHnd, ROW); // draw a card if no values found
            npcCnt++;          // increment npc hand count
            cout << "The opponent draws a card!" << endl
                 << endl;
        }
    }
}
```

Above is the code snippet that shows the simple draw function at the very bottom of the while loop.

High Scores Export

The final step in the long journey to translate our program over to the new format which uses functions is to create a function in order to calculate the stats.

Most of the logic is the same, except that we do not use the *deque* library since we now can use a vector to save the scores. We also split the calculate stats function with separate functions that individually read, save, calculate and display the high scores stored on the separate *.dat* file.

```
// Function to calculate the stats (average, variance, stdDev, pctChng)
void clcSts(vector<int> &scores, unsigned int score, float &average, float &variance, float &stdDev, flo
{
    unsigned int sum = 0;

    for (int i = 0; i < scores.size(); i++)
    {
        sum += scores[i];
    }

    // Calculate average
    if (scores.size() == 1)
    {
        average = static_cast<float>(scores[0]);
    }
    else
    {
        average = static_cast<float>(sum) / scores.size();
    }

    // Calculate variance
    variance = 0;
    for (int i = 0; i < scores.size(); i++)
    {
        variance += pow(scores[i] - average, 2);
    }
    variance /= scores.size();

    // Calculate standard deviation
    stdDev = sqrt(variance);

    // Check if the score is better or worse
    if (average < score)
    {
        btrWrse = "better";
    }
    else
    {
        btrWrse = "worse";
    }
```

The image below shows the passed vector in between the calculate stats page and the load and save functions.

```cpp
// Function to load scores from file
void loadScr(vector<int> &scores, fstream &file)      You, 1 hour ago
{
    int score;
    while (file >> score)
    {
        scores.push_back(score);
    }
    file.close();
}

// Function to save updated scores to file
void saveScr(vector<int> &scores, fstream &file)
{
    file.open("hiScore.dat", ios::out);
    for (int i = 0; i < scores.size(); i++)
    {
        file << scores[i] << endl;
    }
    file.close();
}
```

The last segment is to create a display results as we did in phase 1, except this time we choose to create another copy in which we overload the display results function.

```
// Function to display results
// Original function
void dispRes(int plyCnt, int npcCnt, vector<int> &scores, fstream &file, unsigned int score,
             float average, float variance, float stdDev, float pctChng, string &btrWrse)
{
    if (!file)
    {
        cout << "Error: High score file not found!" << endl;
        exit(1);
    }

    if (plyCnt == 0)
    {
        loadScr(scores, file); // call load scores file

        cout << "Congratulations, you've won!" << endl
             << endl;
        score = npcCnt - plyCnt;
        cout << "You've scored " << score << " points!" << endl;

        scores.push_back(score);

        if (scores.size() > 10)
        {
            scores.erase(scores.begin()); // Remove the oldest score (front of the vector)
        }

        saveScr(scores, file);

        clcSts(scores, score, average, variance, stdDev, pctChng, btrWrse);

        cout << fixed << setprecision(2) << showpoint;
        cout << "Which is " << pctChng << "% " << btrWrse << " than the average of the last ten victories." << endl
             << endl;
        cout << "Average of last " << scores.size() << " scores: " << average << endl;
        cout << "Standard Deviation of the scores: " << stdDev << endl;
    }
    else if (npcCnt == 0)
    {
        cout << "You've Lost!" << endl
             << endl;
    }
}

// Overloaded version without file and statistical details
```

```
// Overloaded version without file and statistical details
void dispRes(int plyCnt, int npcCnt, vector<int> &scores, unsigned int score)
{
    if (plyCnt == 0)
    {
        cout << "Congratulations, you've won!" << endl
             << endl;
        score = npcCnt - plyCnt;
        cout << "You've scored " << score << " points!" << endl;

        scores.push_back(score);

        if (scores.size() > 10)
        {
            scores.erase(scores.begin()); // Remove the oldest score (front of the vector)
        }
    }
    else if (npcCnt == 0)
    {
        cout << "You've Lost!" << endl
             << endl;
    }
}
```

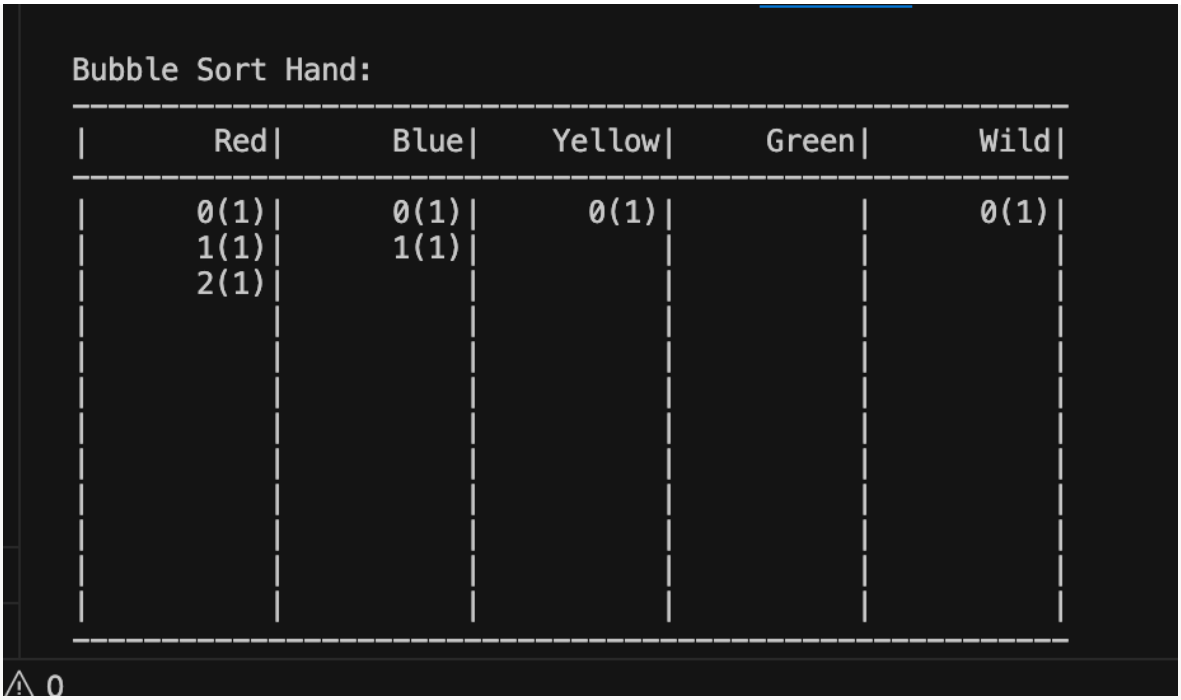
Version 2.2

The goal for this final iteration is to clean up a lot of the UI elements, fix a few bugs, and also implement framework for future implementation of the game.

Implementing Bubble and Selection Sort

As per the quota to meet for the project submission requirements, we implement options for both bubble and selection sort. Implementation took a little bit of thinking of how we can implement the sort properly but I've arrived at one of my initial goals of sorting the cards in hand towards the top and potentially having playable cards be listed up on top. I think we'll save the full functionality for future implementation but as of now, we have bubble sort and selection sort properly functioning in order to sort all of the cards towards the top of the UI.

```
// Bubble Sort Function (returns sorted array in srtdHnd)
void bubSrt(int plyrHnd[][COL], int ROW, string &actvDsp, int srtdHnd[][COL])
{
    // Initialize srtdHnd with zeroes
    for (int i = 0; i < ROW; i++)
    {
        for (int j = 0; j < COL; j++)
        {
            srtdHnd[i][j] = 0;
        }
    }

    for (int j = 0; j < COL; j++)
    {
        vector<int> colVal; // Store nonzero values for sorting

        // Collect nonzero values from the column
        for (int i = 0; i < ROW; i++)
        {
            if (plyrHnd[i][j] != 0)
                colVal.push_back(plyrHnd[i][j]);
        }

        // Bubble Sort
        for (size_t x = 0; x < colVal.size(); x++)
        {
            for (size_t y = 0; y < colVal.size() - x - 1; y++)
            {
                if (colVal[y] > colVal[y + 1])
                    swap(colVal[y], colVal[y + 1]);
            }
        }

        // Populate srtdHnd top-down
        for (size_t i = 0; i < colVal.size(); i++)
        {
            srtdHnd[i][j] = colVal[i];
        }
    }

    // Print the sorted hand
    // cout << "Bubble Sort Hand:" << endl;
    dispHnd(plyrHnd, srtdHnd, actvDsp, ROW);
}
```

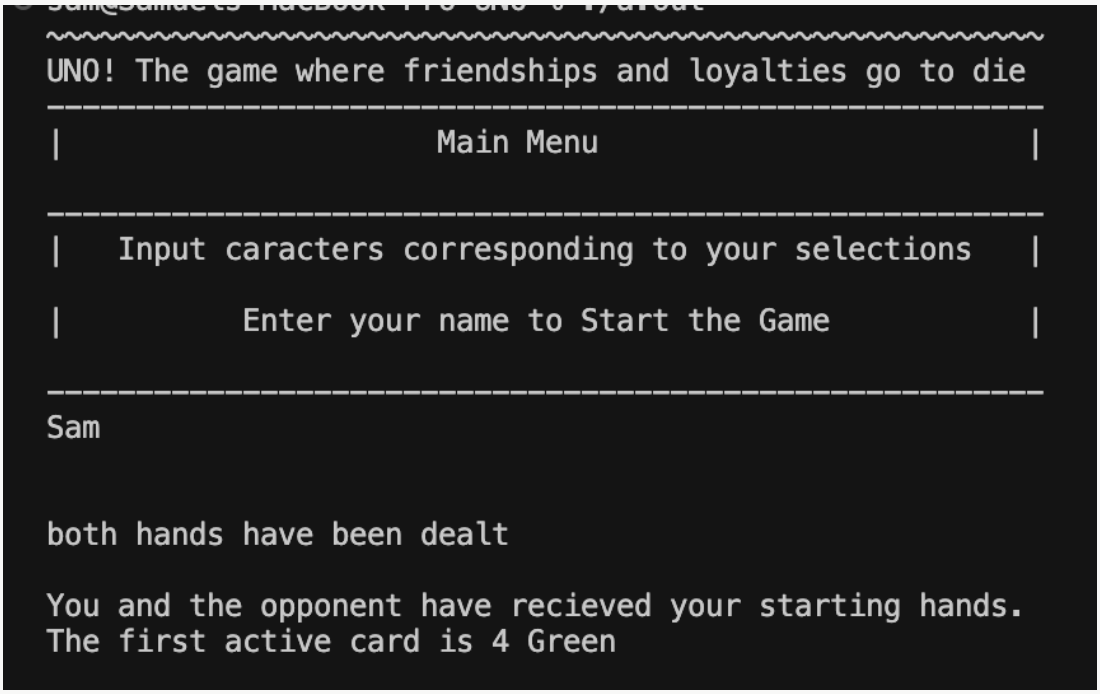


For the purpose of clarity and continuity with the work we've done thus far, we'll resort to the original implementation of hand display.

Cleaning up UI for readability and proofreading entire codebase.

This final segment was supposed to be the quickest final touches on the project, but I've ended up having to clean up a lot of the code I've written in order to eliminate unnecessary snippets across the entire project.

A few hours later, and with some very nit picky perfectionism, we've cleaned up our UI so that it is more aesthetically pleasing and more compact at conveying information.



> UNO Version 5.0 has now been deployed to GitHub at
>
> https://github.com/Sam-T-G/UNO/releases/tag/v7

https://github.com/Sam-T-G/UNO/releases/tag/v7

Project Phase 3 Development

> Implementing structures, memory allocation, and various other concepts

The goal in phase three of development is to further incorporate technologies that will streamline the effectiveness, functionality, and scalability of our UNO! game.

A major goal for this phase is to completely restructure the program's means of processing cards from using a two dimensional array to using structures. Expanding this goal will be a complete upheaval in the way our program stores and processes the game's information, but the functionality will largely be similar. This will completely re-imagine the current indexing system to be more dynamic, with the capability to process and display card quality and quantity much more efficiently and modularly.

Along the major program rewrite, we'll strive to also revamp the player UI to be more concise since we will not be printing out the entirety of the matrix included in the previous versions.

Version 3.1

Goals for UNO Version 3.1

For version 1, we will strive to create replicate the main game interface that exists within prior versions, and establish the new card and player indexing through structures.

Reorganizing Repository

Upon initial initialization of phase 3, I've come to the realization that he version indexing of the UNO! project is due for a revamp in order to more efficiently organize the development progress.

Past versions shall be renumbered to a main index to signify general version phase, and individual version iterations indexed in the following to indicate more minor release updates.

The indexing will reflect the following.

Phase 1:

- V 1.1
- V 1.2
- V 1.3
- V 1.4

Phase 2:

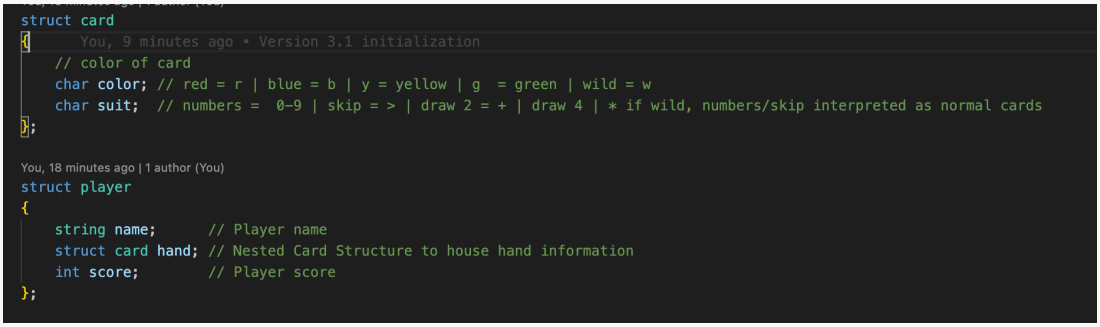
- V 2.1
- V 2.2
- V 2.3

Phase 3:

- V 3.1 (Current)

Pseudocode, Migration, and Restructuring

To start off version 3.1, we begin with organizing important features from prior iterations and re-imagining the structure of how the game should function.

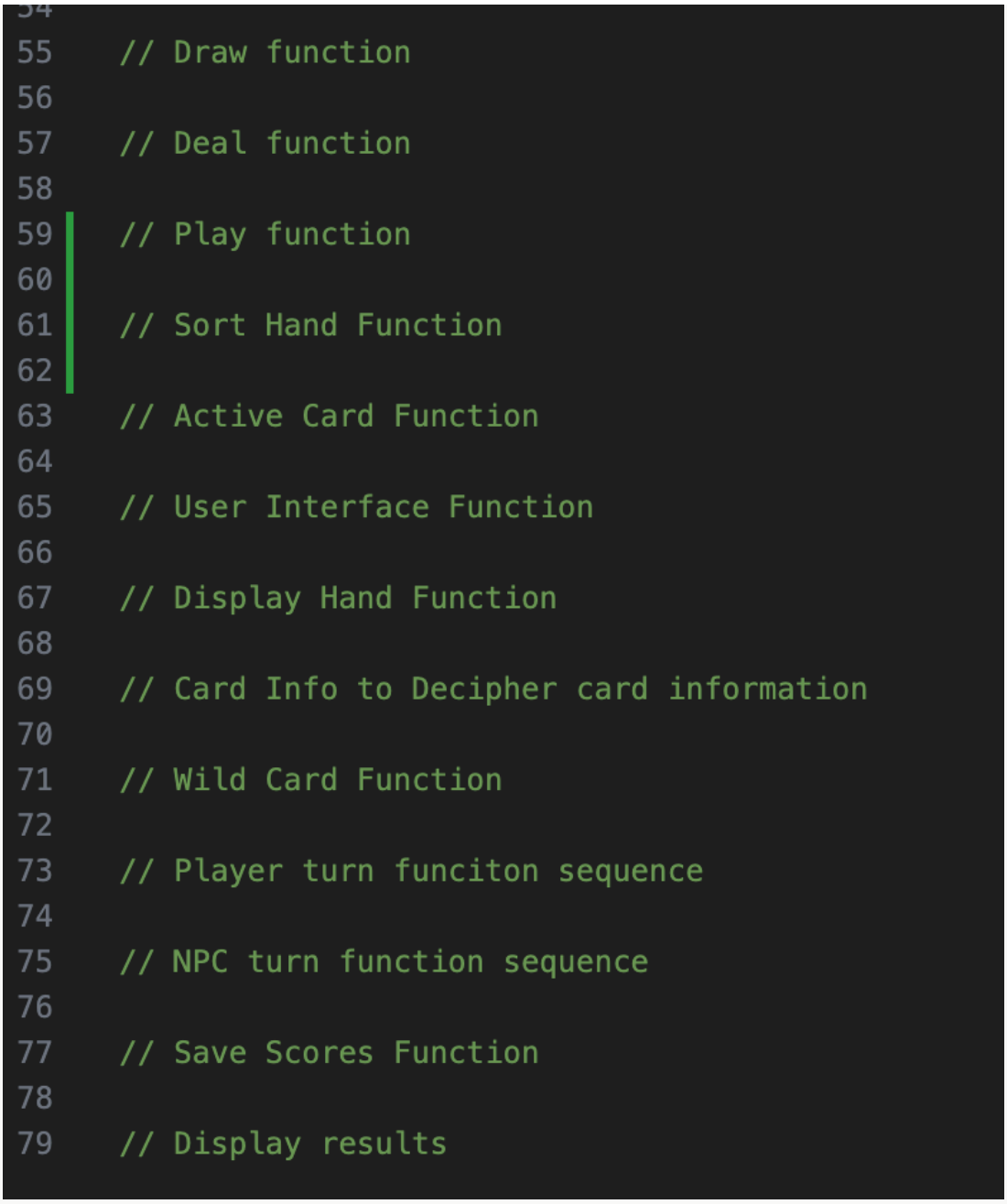


```
struct card
{      You, 9 minutes ago • Version 3.1 initialization
    // color of card
    char color; // red = r | blue = b | y = yellow | g  = green | wild = w
    char suit;  // numbers =  0-9 | skip = > | draw 2 = + | draw 4 | * if wild, numbers/skip interpreted as normal cards
};

You, 18 minutes ago | 1 author (You)
struct player
{
    string name;      // Player name
    struct card hand; // Nested Card Structure to house hand information
    int score;        // Player score
};
```

We begin with the first two structures needed for the transition for our game. Cards will now be created utilizing the new card structures as objects that we will then nest within the player structure we've created. This small iteration also enables us to instantiate multiple playable characters within our game.

Furthermore, we can use the player structure to hold information for both human and computer players.
This will be implemented in a future iteration to maintain our goals of reaching our minimum viable product.

Finally, we create pseudocode in order to outline the general functions we need in order to create a functioning UNO! game as previous versions have established.



These general functions have been migrated but need to be re-structured and re-written in order to function within our new data structuring utilizing structures for both card and player organization.

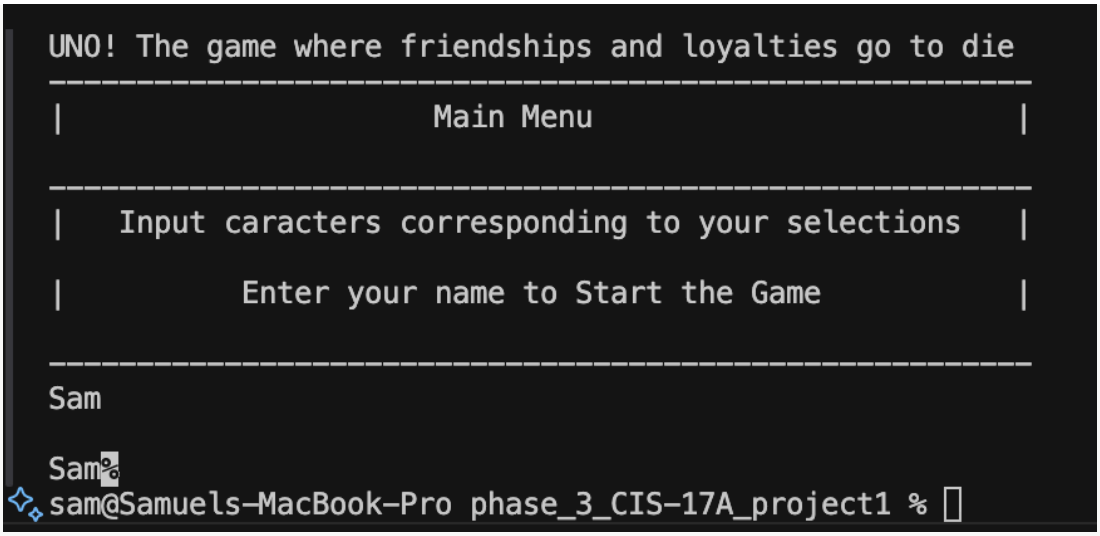
Structure Setup and Menu Migration

We'll start sequentially in our game restructure, starting with creating a main menu function which we modularly call the main menu for future purposes. We will then instantiate our first player object using the player structure and use the previously created main menu user name sequence in order to start our game off.

```
    // Map the inputs and outputs - Process
    Player player1; // Create a player 1 structure to hold player's information - later can be modularized
    menu(player1);  // pass player 1 structure into function
    cout << player1.name;
```

```
// Main menu function
void menu(Player &player1)
{
    for (int i = 0; i < 56; i++)
    {
        cout << "~";
    }
    cout << endl
         << "UNO! The game where friendships and loyalties go to die" << endl;
    for (int i = 0; i < 56; i++)
    {
        cout << "-";
    }
    cout << endl;
    cout << "|" << setw(30) << "Main Menu" << setw(25) << "|" << endl
         << endl;
    for (int i = 0; i < 56; i++)
    {
        cout << "-";
    }
    cout << endl
         << "|" << setw(3) << " " << "Input caracters corresponding to your selections" << setw(4) << "|" << endl
         << endl;
    cout << "|" << setw(10) << " " << "Enter your name to Start the Game" << setw(11) << " " << "|" << endl
         << endl;
    for (int i = 0; i < 56; i++)
    {
        cout << "-";
    }
    cout << endl;
    cin >> player1.name;
    cout << endl;
};      You, 57 seconds ago • Uncommitted changes
```

From this first step, we can test the output and ensure we are on the right track.



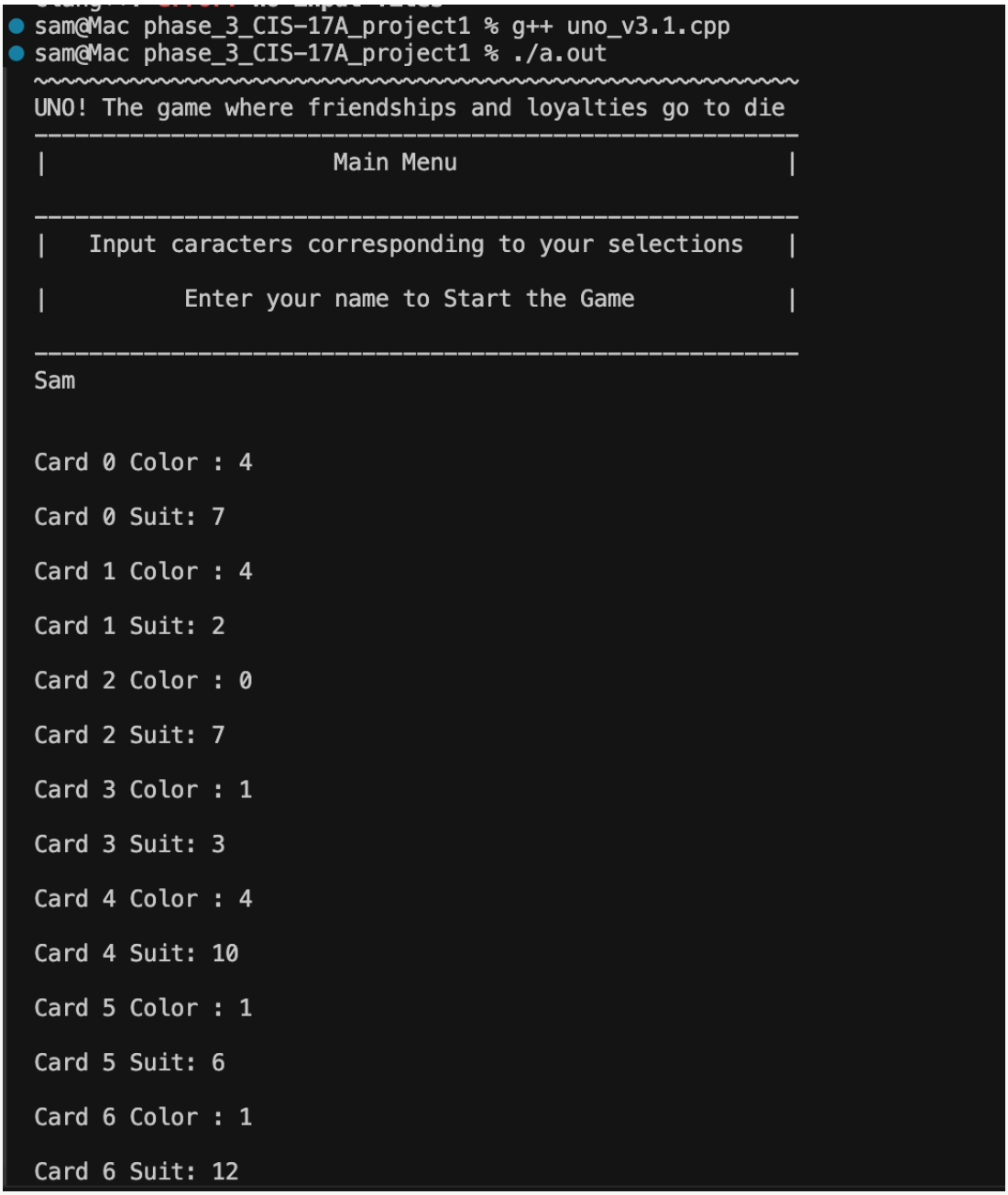
We now begin brainstorming how we want to organize the information that the game will utilize. To begin, we create a card structure that we assign a color and suit, then create an index to identify color and suit respectively. We'll also use enumerated data types to increase readability.

```cpp
enum CardColor
{
    RED,
    BLUE,
    YELLOW,
    GREEN,
    WILD
};

enum CardSuit
{
    ZERO,
    ONE,
    TWO,
    THREE,
    FOUR,
    FIVE,
    SIX,
    SEVEN,
    EIGHT,
    NINE,
    SKIP,
    DRAW_TWO,
    DRAW_FOUR
};
You, 53 seconds ago | 1 author (You)
struct Card
{
    // color of card
    CardColor color; // red = 0 | blue = 1 | 2 = yellow | 3  = green | wild = 4
    CardSuit suit;   // numbers =  0-9 | skip = 10 | draw 2 = 11 | draw 4 = 12 | * if wild, numbers/skip interpreted as normal cards
};

You, 53 seconds ago | 1 author (You)
struct Scores
{
    int numTrns; // Number of turns
    int hiCombo; // integer value to store highest number of combo
};

You, 54 seconds ago | 1 author (You)
struct Player
{
    string name;           // Player name
    vector<Card> hand;     // Nested Card Vector to house hand contents
    struct Scores plyrScr; // nested structure to store scores
};
```

Finally, we adjust our card drawing process such that a function will use the card structure to randomly generate a card and return the structure back.

```
// Draw function
Card draw()
{
    Card newCrd;
    newCrd.color = static_cast<CardColor>(rand() % 5);
    newCrd.suit = static_cast<CardSuit>(rand() % 13);
    return newCrd;
}

// Deal function
void deal(Player &p1, Player &npc)
{ // Deal 7 Starting cards for player and npc
    for (int i = 0; i < 7; i++)
    {
        p1.hand.push_back(draw());
        npc.hand.push_back(draw());
    }
};
```

We then create a simple deal function to handle the initial dealing of cards to both the player and the npc hands. When printing the contents, it should look like the following:


sam@Mac phase_3_CIS-17A_project1 % g++ uno_v3.1.cpp
sam@Mac phase_3_CIS-17A_project1 % ./a.out
UNO! The game where friendships and loyalties go to die
| Main Menu |
| Input caracters corresponding to your selections |
| Enter your name to Start the Game |
Sam
Card 0 Color : 4
Card 0 Suit: 7
Card 1 Color : 4
Card 1 Suit: 2
Card 2 Color : 0
Card 2 Suit: 7
Card 3 Color : 1
Card 3 Suit: 3
Card 4 Color : 4
Card 4 Suit: 10
Card 5 Color : 1
Card 5 Suit: 6
Card 6 Color : 1
Card 6 Suit: 12

Version 3.2

Goals for UNO Version 3.2

For version 2, we will strive to migrate the core of the game logic now that we've established the main game structure.

This next leg of development is the main chunk of progress to be made, albeit most of the logs has been created from the prior version 2.x models. I perceive the difficulty in this next segment to primarily lie in the re-imagining of the functions we've already written to interact with structures rather than a two dimensional array. In many of these cases, this method is a lot faster since the multiple steps of logic needed in order to read and manipulate cards in play for both the player and the NPC do not need to be processed through multiple nested loops and through the constant searching of a two dimensional array. We begin with migrating the basic functionality of card display, UI display and prompts, as well as turn handling.

To start out, we migrate the base logic for playing a wild card and handling color swapping via wild play.

```
// Wild Card Function
void wildCrd(Card &card)
{
    if (card.color == WILD)
    {
        int newClr;
        cout << "Choose a new color (0: RED, 1: BLUE, 2: YELLOW, 3: GREEN): ";
        cin >> newClr;
        card.color = static_cast<CardClr>(newClr);
    }
}
```

This was actually much simpler to execute than the original method since we simply change the color variable within our card structure to represent the new chosen color.

Next we create a modular function to display and translate our enum integer values into readable human strings by simply storing the potential values in arrays as a reference. We then create the edge case for wild cards and also a translation print for regular cards. Just to be safe, we have a failsafe in case the card information happens to be incorrect.

```
// Active Card Display Function
void dispCrd(Card &actvCrd)
{
    // String arrays for descriptive output
    string colors[] = {"Red", "Blue", "Yellow", "Green", "Wild"};
    string values[] = {"0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "SKIP", "DRAW 2", "DRAW 4"};

    if (actvCrd.color == WILD)
    {       You, 52 minutes ago • Logic Migration and Adaptation
        cout << "Wild" << endl;
    }
    else if (actvCrd.color >= 0 && actvCrd.color < 5 && actvCrd.suit >= 0 && actvCrd.suit < 13)
    {
        cout << values[actvCrd.suit] << " " << colors[actvCrd.color] << endl;
    }
    else
    {
        cout << "Invalid Card" << endl;
    }
}
```

Next we create the base overarching logic in main to handle the turn and win conditions. This was also much simpler to write because we can modularize a lot of the logic to take place in separate functions for the NPC and the Player.

```
bool turn = true; // Player starts first

// Display and output the results
while (!p1->hand.empty() && !npc->hand.empty())
{
    if (turn == true) // Player's turn
    {
        plyrTrn(*p1, *npc, actvCrd, turn);      You, 57 minu
    }
    else if (turn == false) // NPC's turn
    {
        npcTrn(*p1, *npc, actvCrd, turn);
    }
}

if (p1->hand.empty())
{
    cout << "You win!" << endl;
}
else
{
    cout << "NPC wins!" << endl;
}
```

The entirety of the main turn handling logic is shown above. We create a boolean to track whether or not it's the player's turn, then pass pointer variables to both npc and player turns so we can manipulate card draws for the opponents when playing draw cards. We then exit the while loop if either player's hands are empty and declare the winner.

Now we begin to modify our original turn handling functions, starting with the player.

```
// Player turn funciton sequence
void plyrTrn(Player &p1, Player &npc, Card &actvCrd, bool &turn)
{
    int choice;

    while (turn == true)
    {
        turn = false;             // Default set turn to false at the start of the loop
        usrInt(p1, npc, actvCrd); // Display the current game state
        cin >> choice;

        if (cin.fail())
        {
            cin.clear();
            cin.ignore(1000, '\n');
            cout << "Invalid input. Try again." << endl;
            turn = true;
        }
        else if (choice == -1)
        {
            cout << "You chose to draw a card." << endl;
            p1.hand.push_back(draw());
            turn = true; // Player goes again after drawing a card
        }
        else if (choice < 0 || choice >= static_cast<int>(p1.hand.size()))
        {
            cout << "Invalid choice. Pick a valid card index or -1 to draw." << endl;
            turn = true; // Allow another turn if the choice is invalid
        }
        else
        {
            play(p1, npc, choice, actvCrd, turn);
        }
    }
}
```

We migrate a majority of the logic, but adjust it so that we do not need to hold a variable to take inventory of the player hand count and also the process of drawing cards is simpler in that we utilize our draw function to generate a new card which we push to the end of the vector.

We also chose to compartmentalize the play function as follows:

```
// Play card function
void play(Player &p1, Player &npc, int choice, Card &actvCrd, bool &turn)
{
    // store selected card
    Card slctd = p1.hand[choice];

    // Error check for card range chosen
    if (choice < 0 || choice >= p1.hand.size())
    {
        cout << "Invalid choice!" << endl;
        return;
    }
```

Above, we've imported some variables created in the player function, as well as create a selected card copy in which we store the information that the player chooses.

We then create the main validation segment in which we compare the selected card color and suit such that playing a match of either will validate the card played. This segment is also vastly different than that of our 2-D array matrix indexing from the prior versions in that we simply erase the card once validated, then condense the array using a combination of the erase and begin functions.

```
// Validate play: same color, same suit (number/action), or wild
if (slctd.color == actvCrd.color || slctd.suit == actvCrd.suit || slctd.color == WILD)
{
    actvCrd = slctd;                          // Update active card
    p1.hand.erase(p1.hand.begin() + choice); // Remove played card
    cout << "You've played a ";
    dispCrd(actvCrd); // Display active card in human-readable format

    // Handle special cards
    if (slctd.suit == 10) // SKIP
    {
        cout << "SKIP played! It's your turn again!" << endl;
        turn = true; // Player goes again
    }
    else if (slctd.suit == 11) // DRAW 2
    {
        cout << "DRAW 2 played! Opponent draws 2 cards!" << endl;
        npc.hand.push_back(draw()); // draw two cards
        npc.hand.push_back(draw());
        cout << "Opponent now has " << npc.hand.size() << " cards!" << endl;
        turn = true; // Player goes again
    }
    else if (slctd.suit == 12) // DRAW 4 (if added)
    {
        cout << "DRAW 4 played! Opponent draws 4 cards!" << endl;
        for (int i = 0; i < 4; i++) // loop to process 4 card draw
        {
            npc.hand.push_back(draw());
        }
        cout << "Opponent now has " << npc.hand.size() << " cards!" << endl;
        turn = true; // Player goes again
    }

    // Handle color choice if Wild
    if (slctd.color == WILD)
    {
        wildCrd(actvCrd); // call wild card function
    }
}
else
{
    cout << "Invalid play: Card does not match active card by color or number!" << endl;
}      You, 1 hour ago • Logic Migration and Adaptation
}
```

As we can observe above, we also can simply create much more efficient logic than having to traverse through a 2-D matrix as was done previously and we simply draw cards and push to the end of the array of either the player or the NPC.

```
while (!valid) // Check if valid play has been made
{
    if (i < npc.hand.size())
    {
        Card c = npc.hand[i]; // Make a copy of card at given index

        if (c.color == actvCrd.color || c.suit == actvCrd.suit || c.color == WILD)
        {
            actvCrd = c;
            npc.hand.erase(npc.hand.begin() + i);

            cout << "NPC played: " << crdInfo(actvCrd) << "!" << endl;

            // Default: player's turn next
            turn = true;

            if (c.color == WILD)
            {
                CardClr newClr = static_cast<CardClr>(rand() % 4);
                actvCrd.color = newClr;
                string colors[] = {"Red", "Blue", "Yellow", "Green"};
                cout << "NPC plays a WILD and chooses " << colors[newClr] << "!" << endl;
            }

            // Handle special cards
            if (c.suit == SKIP)
            {
                cout << "NPC played SKIP! You lose a turn." << endl;
                turn = false;
            }
            else if (c.suit == DRAW_TWO)
            {
                cout << "NPC played DRAW 2! You draw 2 cards." << endl;
                for (int i = 0; i < 2; ++i)
                    p1.hand.push_back(draw());
                turn = false;
            }
            else if (c.suit == DRAW_FOUR)
            {
                cout << "NPC played DRAW 4! You draw 4 cards." << endl;
                for (int i = 0; i < 4; ++i)
                    p1.hand.push_back(draw());
                turn = false;
            }

            valid = true; // Set Valid to true if valid play has been made
        }
```

Something to note in the snippet above, we are able to massively cut back on the opponent AI logic in that we can reduce the external function into a single line in which the NPC can directly manipulate the active color in a single line of code.

```
else
{
    Card drawn = draw();
    npc.hand.push_back(drawn);
    cout << "NPC draws a card!" << endl;

    if (drawn.color == actvCrd.color || drawn.suit == actvCrd.suit || drawn.color == WILD)
    {
        actvCrd = drawn;
        npc.hand.pop_back(); // play the drawn card
        cout << "NPC plays the drawn card: " << crdInfo(actvCrd) << "!" << endl;

        if (drawn.color == WILD)
        {
            CardClr newClr = static_cast<CardClr>(rand() % 4);
            actvCrd.color = newClr;
            string colors[] = {"Red", "Blue", "Yellow", "Green"};
            cout << "NPC chooses " << colors[newClr] << "!" << endl;
        }

        // If it's a special card, handle turn
        if (drawn.suit == SKIP || drawn.suit == DRAW_TWO || drawn.suit == DRAW_FOUR)
        {
            if (drawn.suit == SKIP)
                cout << "NPC played SKIP! You lose a turn." << endl;
            else if (drawn.suit == DRAW_TWO)
            {
                cout << "NPC played DRAW 2! You draw 2 cards." << endl;
                for (int i = 0; i < 2; ++i)      You, 41 minutes ago • Base Functionality an
                    p1.hand.push_back(draw());
            }
            else if (drawn.suit == DRAW_FOUR)
            {
                cout << "NPC played DRAW 4! You draw 4 cards." << endl;
                for (int i = 0; i < 4; ++i)
                    p1.hand.push_back(draw());
            }

            turn = false; // NPC goes again
        }
        else
        {
            turn = true; // Player's turn
        }

        valid = true;
    }
```

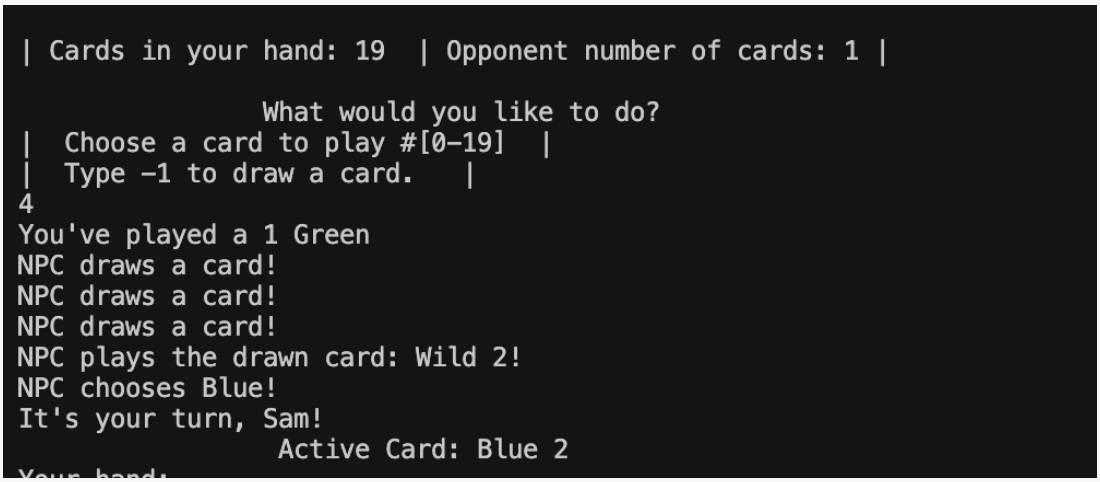
All other logic is similar to the player play verification and organization functions.

The last task to wrap up this version is to test a play-through of the game and the expected interactions.

Matching card color as well as number:

```
| Cards in your hand: 10  | Opponent number of cards: 5 |

                What would you like to do?
|  Choose a card to play #[0-10]  |
|  Type -1 to draw a card.   |
5
You've played a DRAW 4 Yellow
DRAW 4 played! Opponent draws 4 cards!
Opponent now has 9 cards!
It's your turn, Sam!
                 Active Card: Yellow Draw Four
Your hand:
 [0] DRAW 4 Green
 [1] DRAW 4 Blue
 [2] DRAW 4 Green
 [3] Wild
 [4] Wild
 [5] DRAW 2 Yellow
 [6] DRAW 2 Blue
 [7] 8 Green
 [8] DRAW 4 Yellow

| Cards in your hand: 9  | Opponent number of cards: 9 |

                What would you like to do?
|  Choose a card to play #[0-9]  |
|  Type -1 to draw a card.   |
0
You've played a DRAW 4 Green
DRAW 4 played! Opponent draws 4 cards!
Opponent now has 13 cards!
It's your turn, Sam!
                 Active Card: Green Draw Four
Your hand:
 [0] DRAW 4 Blue
 [1] DRAW 4 Green
 [2] Wild
 [3] Wild
 [4] DRAW 2 Yellow
 [5] DRAW 2 Blue
 [6] 8 Green
 [7] DRAW 4 Yellow

| Cards in your hand: 8  | Opponent number of cards: 13 |

                What would you like to do?
|  Choose a card to play #[0-8]  |
|  Type -1 to draw a card.   |
0
You've played a DRAW 4 Blue
```

Above we demonstrate turn handling pertaining to skipping opponent turns when playing skip cards, we demonstrate matching suit to adjust active color, and we demonstrate proper hand content number display.



Above we also demonstrate the NPC logic in action. When the NPC does not have a playable card, the NPC will opt to draw a card until acquiring a playable card. We also demonstrate the NPC's capability to play a wild card and choose a color to set as the new active color.

```
NPC plays a WILD and chooses Blue!
It's your turn, Sam!
                 Active Card: Blue 9
Your hand:
 [0] Wild
 [1] 8 Green

| Cards in your hand: 2  | Opponent number of cards: 28 |

               What would you like to do?
|  Choose a card to play #[0-2]  |
|  Type -1 to draw a card.   |
0
You've played a Wild
Choose a new color (0: RED, 1: BLUE, 2: YELLOW, 3: GREEN): 3
NPC played: Green 9!
It's your turn, Sam!
                 Active Card: Green 9
Your hand:
 [0] 8 Green

| Cards in your hand: 1  | Opponent number of cards: 27 |

               What would you like to do?
|  Choose a card to play #[0-1]  |
|  Type -1 to draw a card.   |
0
You've played a 8 Green
You win!
sam@Mac phase_3_CIS-17A_project1 %
```

Finally we observe the final sequence in a play-through in which the player wins and the game congratulates the player for the win.

> UNO Version 3.2 has now been deployed to GitHub at
>
> https://github.com/Sam-T-G/UNO/releases/tag/v3.2

https://github.com/Sam-T-G/UNO/releases/tag/v3.2

Version 3.3

Goals for UNO Version 3.3

For the final version in the 3.X series, we will focus on creating a new scoring system we will calculate within the scores structure.

For scores, we will track two different metrics - amount of turns taken to win, and also introduce the high combo metric, which will keep track of the highest amount of turns any player will sequentially perform.

We will then export the scores on to an external binary file to store high scores, and furthermore, make finishing touches to improve game UI.

Our first priority will be to implement a new scoring system that will keep track of three metrics:

- Difference of card amount between player and NPC hands
- Amount of turns needed to win
- Highest combo chain performed against opponent

```
if (p1->hand.empty())
{
    cout << "You win!" << endl;
    calcScrs(*p1, *npc); // Save only if player wins
}
else // NPC wins and score not saved
{
    cout << "NPC wins!" << endl;
}

readScrs(); // View history of past game scores
```

First, we'll implement a calculate scores function and a read scores function that will be called at the end of the main function once a winner has been declared. The calculate scores will only write to the binary scores sheet if the player wins, and simply prompts that the NPC is victorious otherwise.

We then include a universal read scores function to read the binary file that has been created to store high scores.

The victory prompt, the score writing process, and the reading of prior scores is displayed below:

```
You've played a 9 Yellow
You win!

--- SCORES ---
Sam wins in 11 turns
Highest combo: 3
Card diff: 8

=== SCORE HISTORY ===
Record 1:
  Player  : Sam
  Turns   : 11
  HiCombo : 3

sam@Samuels-MacBook-Pro phase_3_CIS-17A_project1 %
```

Implemented a feature to sort the player's hand for a better user experience.

```
// Sort Hand Function
void srtHnd(Player &p1)
{
    sort(p1.hand.begin(), p1.hand.end(), [](const Card &a, const Card &b)
        {
        if (a.color == b.color) {
            return a.suit < b.suit;
        }
        return a.color < b.color; });
}
```

In practice, the new user hand is organized as displayed below:

```
It's your turn, Sam!
                Active Card: Yellow 8
Your hand:
 [0] 4 Red
 [1] 5 Red
 [2] 6 Red
 [3] SKIP Red
 [4] 3 Blue
 [5] SKIP Blue
 [6] 9 Yellow
 [7] 9 Yellow
 [8] Wild

| Cards in your hand: 9  | Opponent number of cards: 2 |
```

The sort order organizes by card color, and card numbers within each respective color in ascending order of value.

UI Cleanup and Tweaks

Added a small dialogue to improve input clarity and spacing.

```
turn = false;             // Default set turn to false at the start of the loop
usrInt(p1, npc, actvCrd); // Display the current game state
cout << "Make a move: ";  // Prompt for visual Clarity
cin >> choice;            // Input player choice
```

Before:
The “0” line is the user input which is very hard to discern among the other text existing.

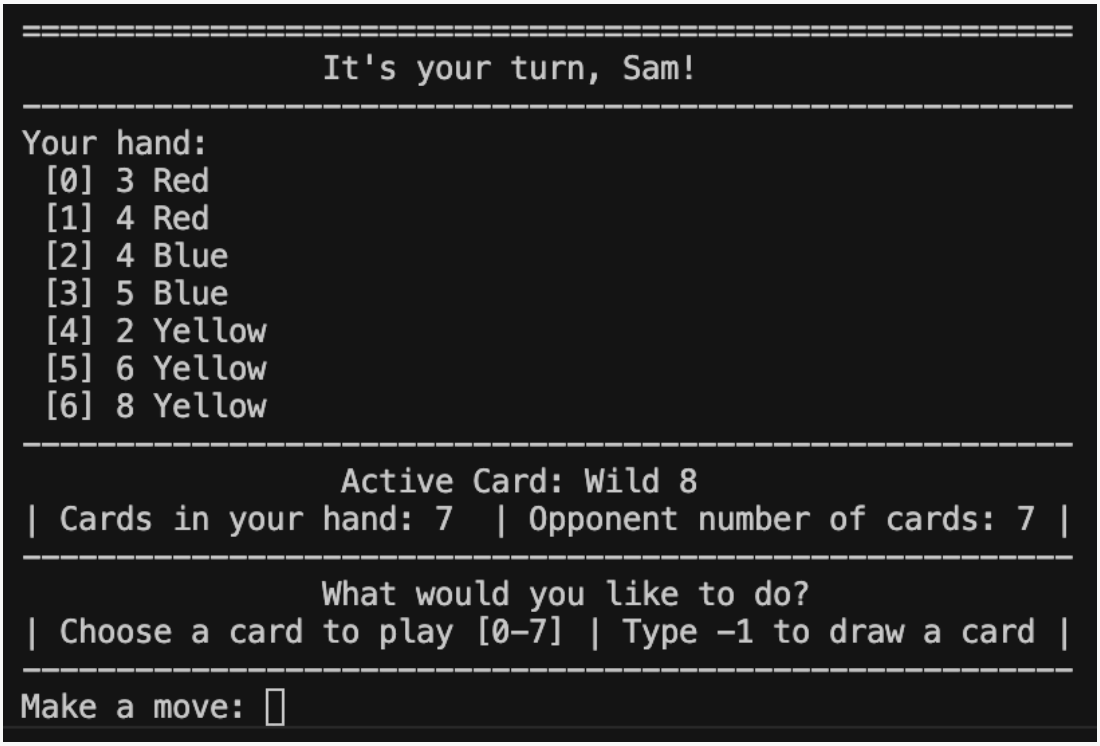
```
|  Choose a card to play #[0-7]  |
|  Type -1 to draw a card.   |
0
You've played a 5 Red
NPC played: Red 3!
It's your turn, Sam!
                    Active Card: Red 3
```

After:

```
| Cards in your hand: 7  | Opponent number of cards: 7 |

               What would you like to do?
|  Choose a card to play #[0-7]  |
|  Type -1 to draw a card.   |
Make a move: 2
You've played a 6 Blue
NPC played: Wild 4!
NPC plays a WILD and chooses Green!
It's your turn, Sam!
```

Further clarity improvements:

And even more UI formatting for a better user experience - featuring borders to compartmentalize data.



We also update the wild card play function so that it has better readability, and also now takes in either uppercase or lowercase character inputs to represent the color the user wants to swap to.

The program is structured as follows:

```
void wildCrd(Card &card)
{
    if (card.color == WILD)
    {
        char newClr;
        bool valid = false;

        while (!valid)
        {
            cout << "------------------------------------------------------" << endl
                 << setw(18) << " " << "Choose a new color!" << endl
                 << setw(6) << " " << "R = Red, B = Blue, Y = Yellow, G = Green" << endl
                 << "------------------------------------------------------" << endl
                 << "New Color: ";
            cin >>
                newClr;
            newClr = toupper(newClr); // Handle lowercase input
            cout << "------------------------------------------------------" << endl;

            switch (newClr)
            {
            case 'R':
                card.color = RED;
                cout << "You selected: Red" << endl;
                valid = true;
                break;
            case 'B':
                card.color = BLUE;
                cout << "You selected: Blue" << endl;
                valid = true;
                break;
            case 'Y':
                card.color = YELLOW;
                cout << "You selected: Yellow" << endl;
                valid = true;
                break;
            case 'G':
                card.color = GREEN;
                cout << "You selected: Green" << endl;
                valid = true;
                break;
            default:
                cout << "Invalid color selection. Please try again." << endl;
            }
            cout << "======================================================" << endl;
        }
    }
}
```

Note: There is an invalid color selection in which the user is prompted to make another selection. An example is shown in the prior screenshot above.

Fixed a bug in which when the user selects a mis-matched card or color, the player's turn ends.

```
else
{
    cout << "Invalid play: Card does not match active card by color or number!" << endl;
    turn = true; // Let the player try again
}
```

This is simply done above by flipping the user turn back to true in the final else statement.

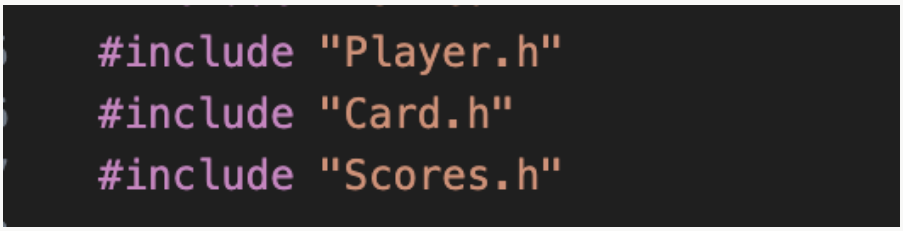
Fixed another bug where NPC wins off drawn card, and plays a special card to finish the game.

```
// Edge case for when NPC wins with last drawn card
if (npc.hand.empty())
{
    cout << "NPC plays the final drawn card and wins!" << endl;
    return;
}
```

Along with the above, also implemented edge case fix for a bug that occurs when the player wins off of playing a wild card.

```
actvCrd = slctd;                          // Update active card
p1.hand.erase(p1.hand.begin() + choice); // Remove played card
cout << setw(16) << " " << "You've played a ";
dispCrd(actvCrd); // Display active card in human-readable format

if (p1.hand.empty())
{
    cout << setw(16) << " " << "You've played your last card!" << endl;
    return; // Exit early — game ends after this play
}
```

The next step, we strive to clean up the main code by compartmentalizing various structures and player functions to external files. First, we've implemented external files to house structures with their respective file names.



Next, we've modularized the main player functionality to an external Player.cpp file.

```
#include "Player.h"
#include "Card.h"

extern Card draw();

void Player::rstCmb()
{
    cmb = 0;
}

void Player::updCmb()
{
    cmb++;
    if (cmb > cmbMx)
        cmbMx = cmb;
}

void Player::drwCrd()
{
    hand.push_back(draw());
}

int Player::hndSze() const
{
    return hand.size();
}
```

Here, we handle the combo counter and also implement a reset combo function, we handle the draw card function, and a function to return the current hand size.

We then fix a bug and simultaneously utilize our new player functions of drwCrd() and rstCmb (draw card and reset combo). The bug occurred in the way the program handled combo counter resets - it never reset when the player chose to draw a card.

```
else if (choice == -1)
{
    cout << "You chose to draw a card." << endl;
    p1.drwCrd();
    p1.rstCmb(); // Reset combo streak
    turn = true; // Player goes again after drawing a card
}
```

The scores now accurately reflect the proper metrics and combo resets.

Record 1 shows the bugged hi-combo play-through, and record 2 now shows the proper combo amounts.

```
You win!
Update your saved score? (y/n): y

--- SCORES ---
Sam wins in 32 turns
Highest combo: 4
Card diff: 13

=== SCORE HISTORY ===
Record 1:
  Player  : Sam
  Turns   : 22
  HiCombo : 37

Record 2:
  Player  : Sam
  Turns   : 32
  HiCombo : 4

sam@Samuels-MacBook-Pro phase_3_CIS-17A_project1 %
```

Next, we create a separate file to house the scores structure.

```
#ifndef SCORES_H
#define SCORES_H

struct Scores
{
    int trns;
    int cmbHi;
};

#endif
```

Following that, we implement a feature in which we can prompt the user whether or not they want to save the recent score to the high scores page.

```
if (p1->hand.empty())
{
    cout << "You win!" << endl;
    char upd;
    cout << "Update your saved score? (y/n): ";
    cin >> upd;
    if (tolower(upd) == 'y')
    {
        calcScrs(*p1, *npc);
    }
}
```

We then create a new SaveData structure and copy the data into our new structure in order to save the data into our binary file.

```
// Save Scores Function
void calcScrs(Player &p1, Player &npc)
{
    p1.scr.trns = p1.trns;
    p1.scr.cmbHi = p1.cmbMx;

    int diff = npc.hand.size(); // card difference

    cout << "\n--- SCORES ---\n";
    cout << p1.name << " wins in " << p1.scr.trns << " turns\n";
    cout << "Highest combo: " << p1.scr.cmbHi << '\n';
    cout << "Card diff: " << diff << "\n";

    // Prepare SaveData object
    SaveData data;
    strncpy(data.name, p1.name.c_str(), sizeof(data.name));
    data.name[sizeof(data.name) - 1] = '\0'; // ensure null-terminated
    data.scr = p1.scr;

    ofstream out("scores.dat", ios::binary | ios::app);
    if (out)
    {
        out.write(reinterpret_cast<char *>(&data), sizeof(SaveData));
        out.close();
    }
    else
    {
        cerr << "Failed to write to scores.dat\n";
    }
}
```