Project 2

Uno Game

Course

CIS-17A

Section

47466

Due Date

December 21, 2025

Author

Matthew Madrigal

Table of Contents

Project 2 Proposal

Introduction

The primary goal of this project was to show our ability to use classes. Inheritance, operator overloading, polymorphism, and templates in C++. I decided to extend off my Project 1 code for an Uno game to fulfil the requirements. The version of the Uno project I decided to extend off used primarily structures to hold all the data. When I first wrote it, I planned ahead by considering how I would convert it into classes and then writing it in a way that would be easy to convert. I kept everything in it modular and in their separate functions whenever possible. This preplanning made working on this project much easier. Changing project 1 to use classes would make expansion and implementation of more complex features much easier.

Gameplay and Rules

This will largely be a repeat of the first project proposal since the rules of the game have remained the same. The fundamental rules of uno are that a card is placed, you and an opponent have cards, and whoever goes next either must play a card with the same color, number, or draw from the draw pile. There are exceptions to this rule, such as if the player has a wild card, then it can be placed no matter the previous card. There are also special cards that may force the opponent to draw cards or even skip the opponents turn. These are called skip cards, reverse cards, draw 2 cards, and draw 4 cards. The skip card and reverse card skip the turn of the opponent in a two-player game. The draw 2 and wild draw 4 card skip the opponent's turn and forces them to draw 2 or 4 cards respectively. When the game starts, players have 7 random cards. The card to start it all off is drawn from the draw pile and can't be a wild card. During gameplay, a player is not supposed to play a draw 4 wild card unless they don't have a matching color. They are allowed to do it but that opens them up to being challenged by the opponent where they can see their hand and if the player has a matching color, they instead are forced to draw 4. When a player draws a card, they can only draw once. When they draw the card, they can play it if its playable during the same turn where they draw. The rules I referenced from can be found here (https://service.mattel.com/instruction_sheets/42001pr.pdf).

Development Summary

There are 82 lines in main.cpp, 499 lines in Game.cpp, 47 lines in Game.h, 27 lines in GameS.h, 268 lines in Player.cpp, 45 lines in Player.h, 180 lines in Vector.h, 65 lines in Computer.cpp, 19 lines in Computer.h, 70 lines in Card.cpp, and 23 lines in Card.h which adds up to 1,325 lines in the files excluding the start.txt which is just for game setup. This project makes use of classes, subclasses, structs, pointers, dynamically allocated memory, random reads and writes from files, reading and writing to the same file, reading from more than 1 file, strings, functions, and other things.

V1.1 (Not in github but is first commit)

I was able to write my own Vector classes which I would use in my program to make managing arrays much easier. I thought it had everything I needed and the class would work but as it turned out later on, there was some errors and missing things.

V1.2

I was able to convert the Card struct to a class and add the printCard function. I also started work on converting the Game struct to a class. This version cannot be compiled due to the active work on converting the Game struct to a class.

V1.3

The classes have had enough changes made to them that they can now be compiled. They don't do anything functionally as classes yet. I also moved everything for the vector into the header file since it can't compile as a template class without being in the same file.

V1.4

The game now uses the Game class logic to run. There are still missing classes that need to be added such as a player class and the game still doesn't use the vector class.

V1.4.1

A segfault crash was found when the game loads a save and then quits

V1.5

I believe I finished the player and computer class now. I also fixed an issue in the Vector class.

V1.6

Almost everything is using their proper classes now. The only thing left will be the score. The Game seems to be fairly stable with the limited testing I have done.

V1.7

This will likely be V2.0 after some final testing and review. It should have everything that is on the checklist for the project. I added the += operator to add to the score in the player object. I also made the Game object use the player object for keeping score instead of it's own variables and made the player object a friend to the Game object to fulfill the requirement of making an object have a friend (unlike myself).

V2.0

After more testing 2 critical bugs were found and patched. These bugs included an issue where the memory would become corrupted when the game tried to calculate the points earned and a bug that allowed the player to keep catching a missed uno on the computer and the bot's hand count would not update when the bot drew cards from the uno catch. There should hopefully be no bugs now and I believe this project is done code wise. I was also able to clean up some of the Game class to remove unused functions. I also removed unused variables in the main class.

Specification

The specifications have not changed much from the first project and the output is expected to remain the same so most of the examples will be the same as Project 1.

Classes and layout

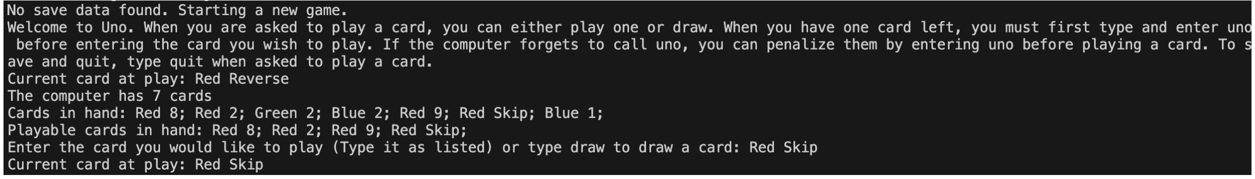
I programmed this game by using a main class which stored the current state of the game and handled the primary logic. Most of the code is split up into smaller functions in their respective to make reusing code and debugging easier. The main class is first created in the main function and is given the save structure or starting file as needed. Main function then calles the playGame function which is where the program will spend most of its time. I also have a class for a card which contains the color (I consider wild a color in this program) and the type which can be either a number or a special card such as draw 2. The card also has a printCard function to quickly translate the class into human readable text. There are also a class for the player and subclass for the computer which allow them to play the game. Finally, I have a structure for saving the game which is a more basic version of the main game structure. Instead of having dynamic arrays, it has static arrays of the unique cards, and how many are in each dynamic array from the main game structure. This makes restoring the program state after a save and load much easier

Program start

When the program is ran, the game first checks if a save file exists. If it does, it tries to read from it. If it notices the save file is of a game that already ended, it will ignore it and create a new game. If its not, it will recreate the game with a helper function and then resume it. If there was no save file it would then continue to create a new game. Finally, before running the main loop, the program explains what the program is and some important things that might not be obvious at first.

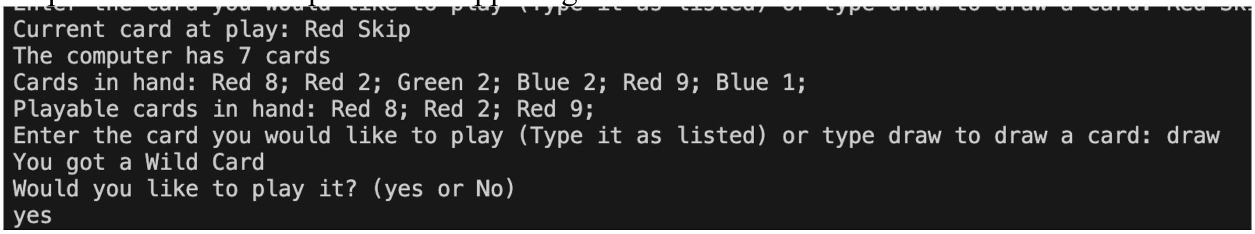
Game Loop

The first thing the program then asks from the user is what card they want to play. The cards the player has and the cards the player can play, and the number of cards the computer has are printed for the player to see before this. The player is then expected to type out one of the playable cards, the word draw, quit, or uno. The program then performs the actions required for the card played and then continues. Below is an example of the player playing a red skip card.



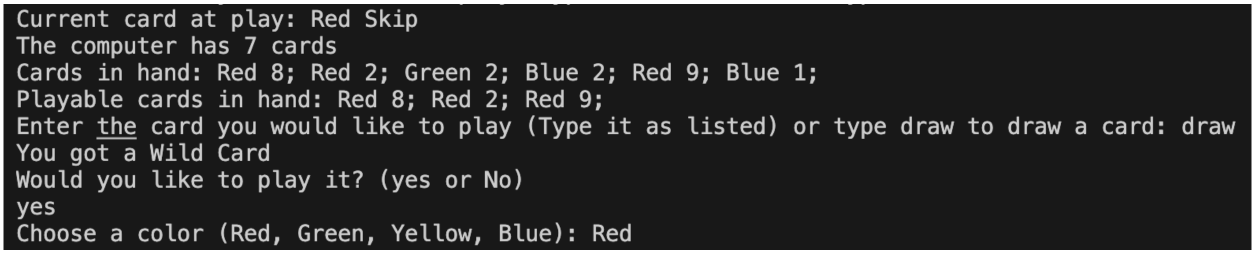
```
No save data found. Starting a new game.
Welcome to Uno. When you are asked to play a card, you can either play one or draw. When you have one card left, you must first type and enter uno
 before entering the card you wish to play. If the computer forgets to call uno, you can penalize them by entering uno before playing a card. To s
ave and quit, type quit when asked to play a card.
Current card at play: Red Reverse
The computer has 7 cards
Cards in hand: Red 8; Red 2; Green 2; Blue 2; Red 9; Red Skip; Blue 1;
Playable cards in hand: Red 8; Red 2; Red 9; Red Skip;
Enter the card you would like to play (Type it as listed) or type draw to draw a card: Red Skip
Current card at play: Red Skip
```

Normally, the turn will alternate every time a card is played, however in some situations, such as the example input/output, this is not the case and the turn doesn't change. Other cards that can cause this include a skip, reverse, and draw 4 card. In the situation that the player decides to draw, the game gives them a card from the draw pile which if playable they can choose to play or keep. Below is an example of this happening.



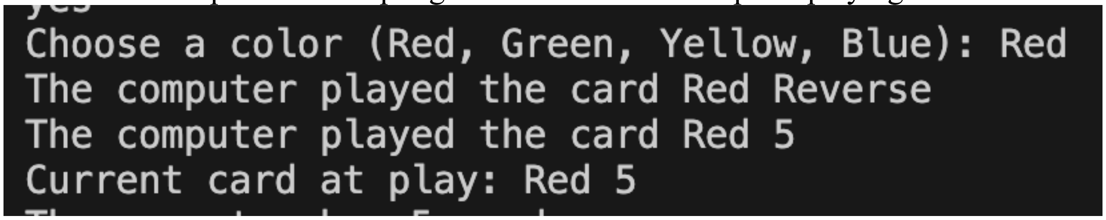
```
Enter the card you would like to play (Type it as listed) or type draw to draw a card: Red Sk
Current card at play: Red Skip
The computer has 7 cards
Cards in hand: Red 8; Red 2; Green 2; Blue 2; Red 9; Blue 1;
Playable cards in hand: Red 8; Red 2; Red 9;
Enter the card you would like to play (Type it as listed) or type draw to draw a card: draw
You got a Wild Card
Would you like to play it? (yes or No)
yes
```

When the player plays any wild card, they will be asked what color they want. This is shown in the same example input which is extended below.



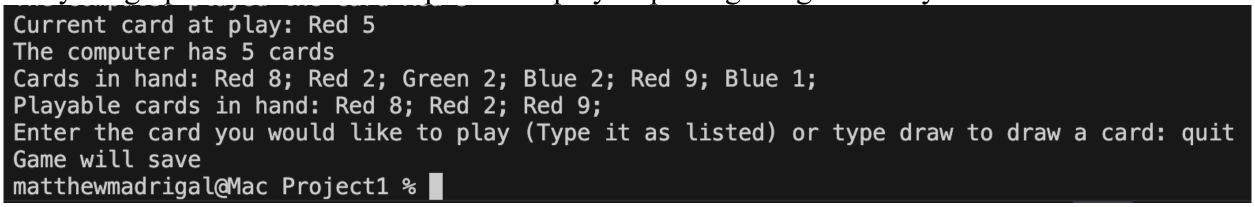
```
Current card at play: Red Skip
The computer has 7 cards
Cards in hand: Red 8; Red 2; Green 2; Blue 2; Red 9; Blue 1;
Playable cards in hand: Red 8; Red 2; Red 9;
Enter the card you would like to play (Type it as listed) or type draw to draw a card: draw
You got a Wild Card
Would you like to play it? (yes or No)
yes
Choose a color (Red, Green, Yellow, Blue): Red
```

After this, the computer will get to play with most of the same things happening. The primary difference is that the choices are made by a random number generator instead of the player and some of the stuff shown to the player such as the hand isn't shown when the computer plays. Here is an example of the output generated from the computer playing two cards in a row.



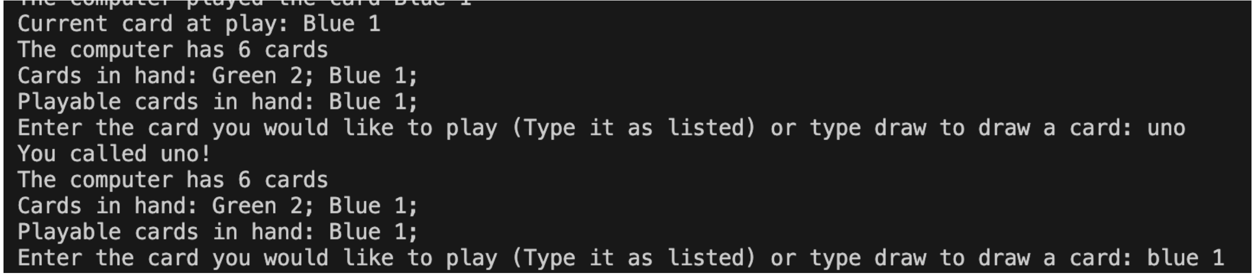
```
yes
Choose a color (Red, Green, Yellow, Blue): Red
The computer played the card Red Reverse
The computer played the card Red 5
Current card at play: Red 5
```

While the player continues to play, the game will keep track of the number of cards they have to handle the uno feature and end the round or game when needed. When the round ends, the game calculates the score of the winning player by examining the value of the cards the losing player had. Once a player gets over 500 points, the game ends and the program saves the game before cleaning up everything and exiting. The player is also able to quit the program early by typing quit when asked to play a card which will still result in the game saving before cleaning everything up. Below is an example of the player quitting the game early.



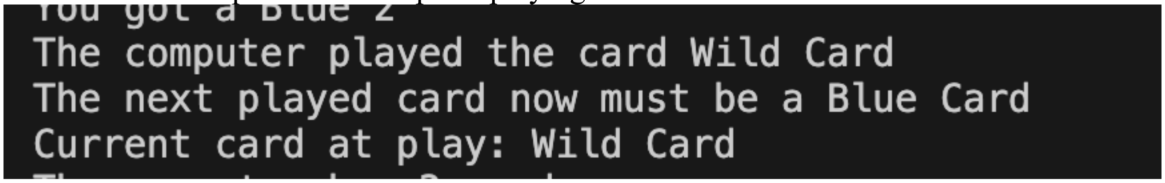
```
Current card at play: Red 5
The computer has 5 cards
Cards in hand: Red 8; Red 2; Green 2; Blue 2; Red 9; Blue 1;
Playable cards in hand: Red 8; Red 2; Red 9;
Enter the card you would like to play (Type it as listed) or type draw to draw a card: quit
Game will save
matthewmadrigal@Mac Project1 %
```

Below is an example of the player calling uno before having one card left



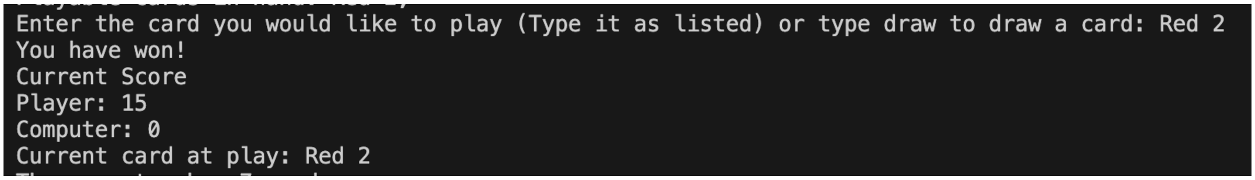
```
The computer played the card Blue 1
Current card at play: Blue 1
The computer has 6 cards
Cards in hand: Green 2; Blue 1;
Playable cards in hand: Blue 1;
Enter the card you would like to play (Type it as listed) or type draw to draw a card: uno
You called uno!
The computer has 6 cards
Cards in hand: Green 2; Blue 1;
Playable cards in hand: Blue 1;
Enter the card you would like to play (Type it as listed) or type draw to draw a card: blue 1
```

Below is an example of the computer playing a wild card



```
You got a Blue 2
The computer played the card Wild Card
The next played card now must be a Blue Card
Current card at play: Wild Card
```

Below is an example of the player winning and the game starting a new round



```
Playable cards in hand: Red 2;
Enter the card you would like to play (Type it as listed) or type draw to draw a card: Red 2
You have won!
Current Score
Player: 15
Computer: 0
Current card at play: Red 2
The computer has 7 cards
```

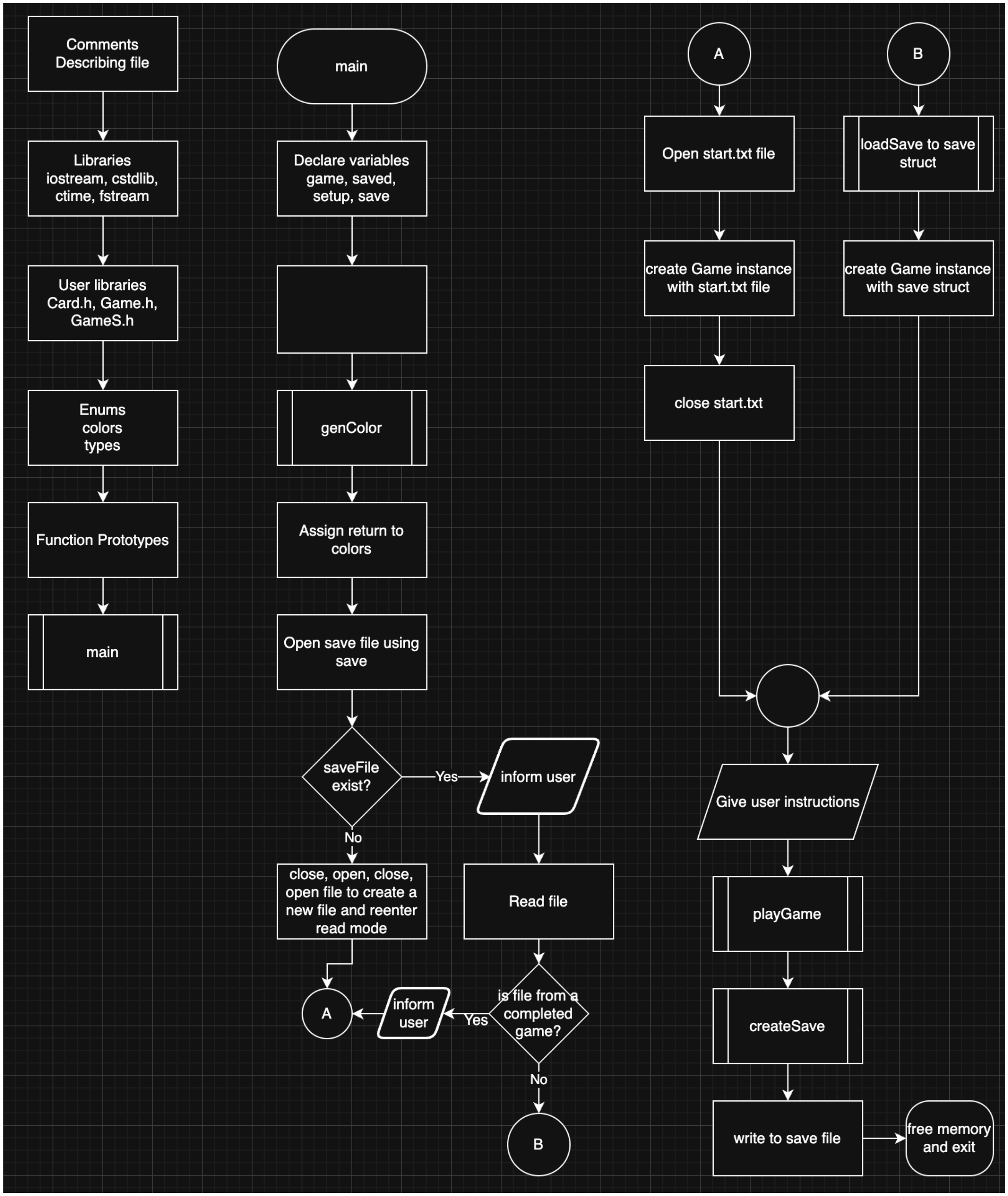
Below is an example of the computer somehow managing to do incredibly well and preventing the player from doing anything for multiple turns in a row (I couldn’t screen shot this one).

Enter the card you would like to play (Type it as listed) or type draw to draw a card: draw
You got a Yellow 9
The computer played the card Red Skip
The computer played the card Red Draw Two
You will draw 2 cards now
You drew a Wild Draw Four
You drew a Green 3
The computer played the card Blue Draw Two
You will draw 2 cards now
You drew a Yellow 8
You drew a Wild Draw Four
The computer played the card Blue Reverse
The computer drawed a card
Current card at play: Blue Reverse
The computer has 6 cards
Cards in hand: Yellow 9; Blue 2; Blue 6; Wild Draw Four; Green 3; Yellow 8; Wild Draw Four;
Playable cards in hand: Blue 2; Blue 6; Wild Draw Four; Wild Draw Four;
Enter the card you would like to play (Type it as listed) or type draw to draw a card:

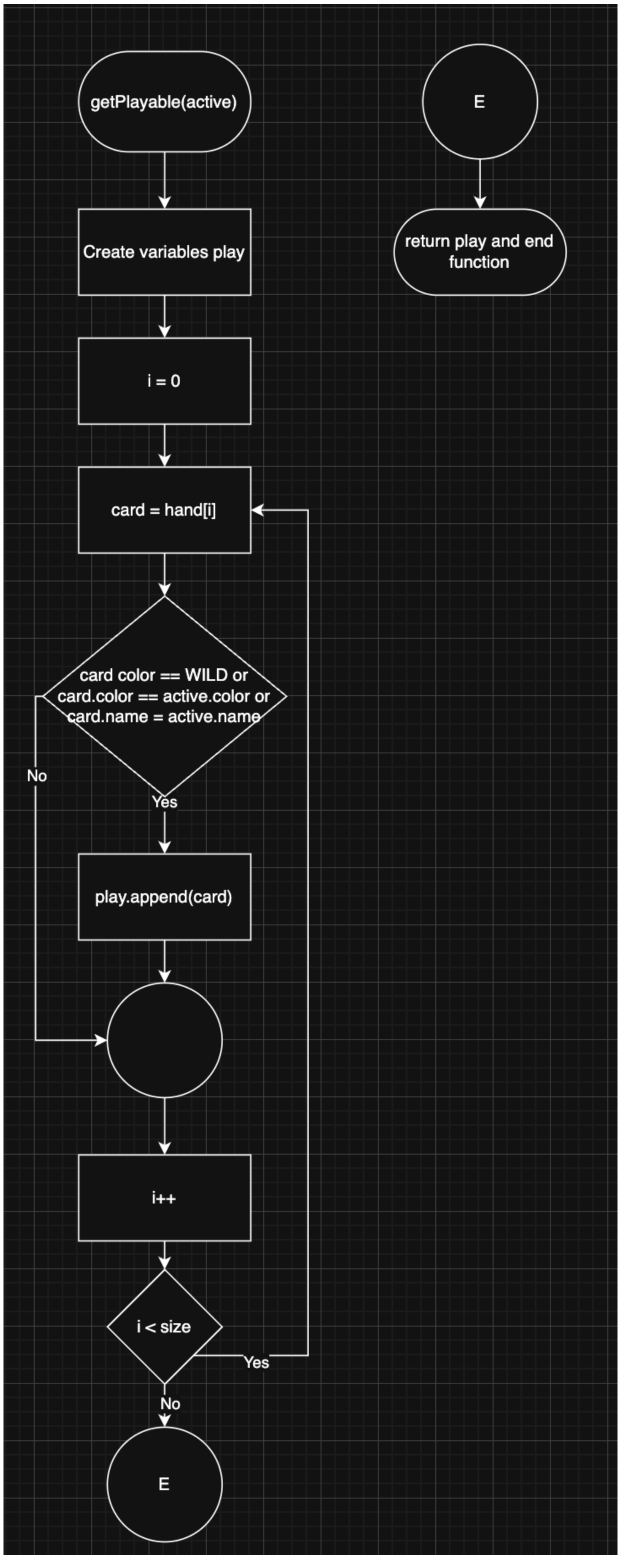
All of these primarily managed by the playGame function. To perform all of these actions, many smaller helper functions are called when needed. These functions include printDeck, printCard, getPlayable, challengeWin, calcPoints, strToCol, strToNm removeCard, drawCard, genColor, copy, addCard, setupPile, playCard, setupPile, and createSave.

Flowchart

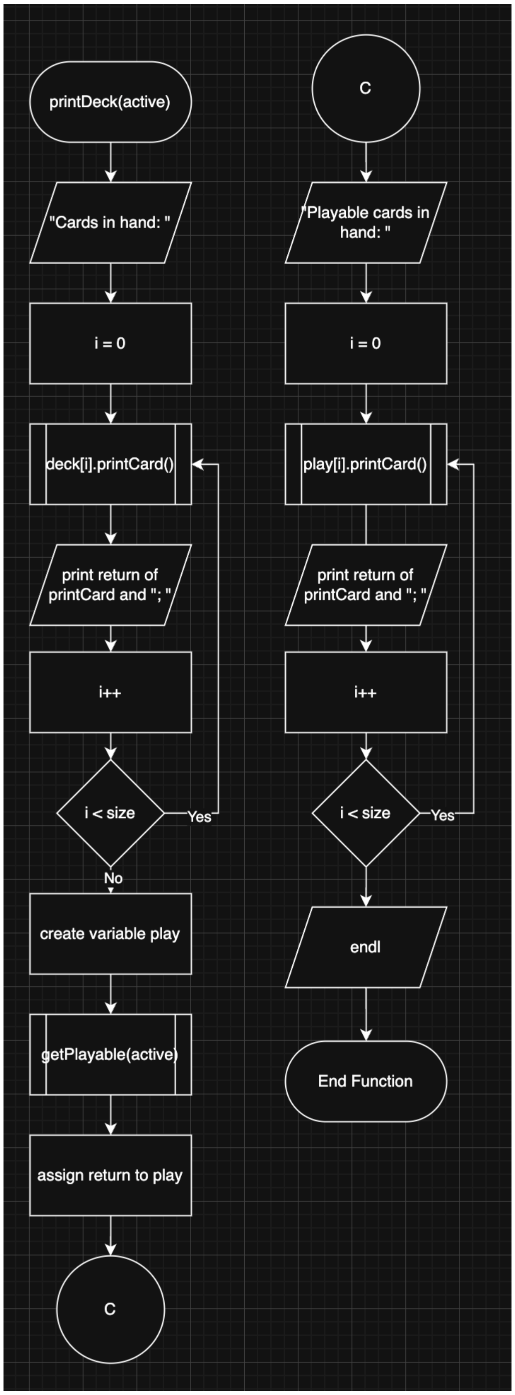
I have created 3 flowcharts for 3 different functions
Main:


Comments Describing file
Libraries iostream, cstdlib, ctime, fstream
User libraries Card.h, Game.h, GameS.h
Enums colors types
Function Prototypes
main
main
Declare variables game, saved, setup, save
genColor
Assign return to colors
Open save file using save
saveFile exist?
Yes
inform user
No
close, open, close, open file to create a new file and reenter read mode
Read file
is file from a completed game?
Yes
inform user
A
No
B
A
Open start.txt file
create Game instance with start.txt file
close start.txt
B
loadSave to save struct
create Game instance with save struct
Give user instructions
playGame
createSave
write to save file
free memory and exit

getPlayable:


getPlayable(active)
E
Create variables play
return play and end function
i = 0
card = hand[i]
card color == WILD or card.color == active.color or card.name = active.name
No
Yes
play.append(card)
i++
i < size
Yes
No
E

printDeck:


printDeck(active)
"Cards in hand: "
i = 0
deck[i].printCard()
print return of printCard and "; "
i++
i < size
Yes
No
create variable play
getPlayable(active)
assign return to play
C
C
"Playable cards in hand: "
i = 0
play[i].printCard()
print return of printCard and "; "
i++
i < size
Yes
endl
End Function

Pseudocode

Below is the pseudocode for the main function

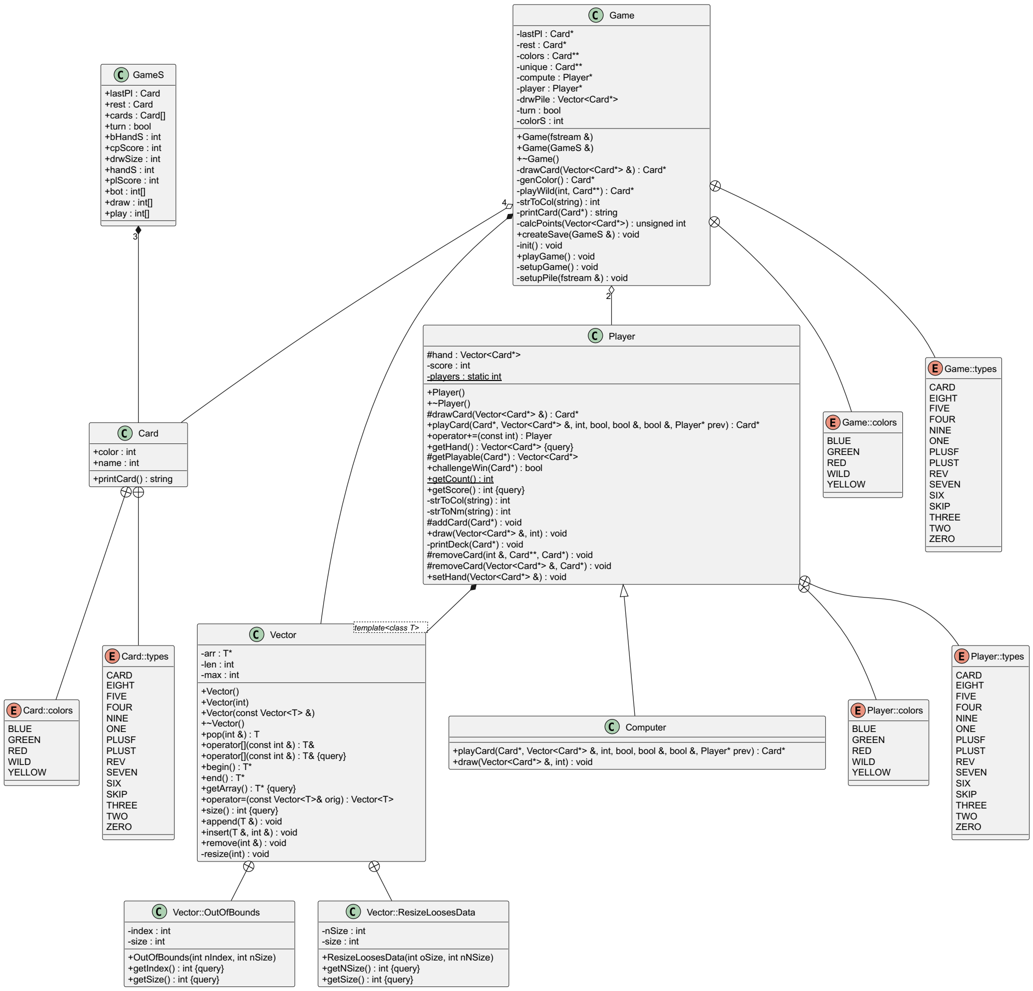
```
Declare variables
Open save file
If saveFile has a game, read it and create game instance using save file
Otherwise initialize a new game with the setup file
Display instructions
Play game
 Save game to file
exit
```

Below is the pseudocode for the remove card function

```
Search for the first element in the array with the same pointer
Save the index of the element in index
Starting at index+1 with I = index+1, set the element at i-1 to the element at i.
I++
Decrement size by 1
```

UML (classes and structs)

Below are the classes and structs I used in my program. This image is available on my Github repository. (there is transparency, so a dark background makes the arrows invisible).



Game

This class is the primary class that does all of the work. The most important function in it is playGame(). It loops until someone wins or the player tells the program to quit. Without the function, the game would not be able to perform the special cards such as draw two and it would not be able to manage and call the correct turn. The Game class also has the player, computer, last played card called lastPl, color restrictions, the actual cards in unique so they are not duplicated, the draw pile in drwPile, the current turn, and the size of the colors array. The Game class can be initialized either with a setup file or a save structure.

Player

This class is the base class for the Computer class. It stores the hand, score, and number of players in a static int. The constructor initializes the score and adds to the player count. The destructor does nothing. The playCard function shows the user their cards along side other information before asking what card they want to play. It uses multiple helper functions included in the library to read the input and perform its job. getHand is a getter function to make reading the hand in the Game class easier. setHand is used by the Game class to setup the hand at the beginning of the game. The += operator is used to add to the score. strToCol and strToNm both help with decoding the user input. Both removeCard functions search for a specific card and then remove them from the array. getScore and getCount are both getters for score and players respectively. printDeck is used to display the cards the user has. The draw function draws cards and displays what was drawn.

Computer

The computer class is a subclass of the Player class. The only thing it does is replace the playCard function with one that randomly plays a card and may forget or remember to both call uno and catch your uno. It also replaces the draw function that way it doesn't output what cards were drawn.

Vector

The Vector class is my version of the stl vector class. It stores the length, the current max size, and the actual array. It has a constructor that can initialize a blank array, initialize a blank array with a set max size, and copy a full array. The vector class has many getter functions including begin, end, [] operator which access the array, getArray, and size. There are also setter functions such as remove, insert, append, and pop.

Card

The card class just holds the color, type, and has a printCard function which returns the type of card as a string.

OutOfBounds

The OutOfBounds class stores an index and size. It has getters for each of them and a constructor to set them. It's supposed to be used if an attempt is made to access out of bounds of the array.

ResizeLosesData

The ResizeLosesData class stores the current size and new size. It has getters for each of them and a constructor to set them. It's supposed to be used incase an attempt is made to resize the array to be smaller than what is required to hold the existing data.

colors

This is an enum which represents the possible integer colors in a more easily readable and manageable way.

types

This is an enum which represents the possible integer types in a more easily readable and manageable way

GameS

This structure is used to store all of the data in a way that can easily be saved to a binary file.

Important Variables

The most important variables in this program are the game variable. It holds all the information for the entire game including the turn, player, and computer. Without it the game wouldn't work at all or would be very fragmented and not very modular. The unique variable and colors variable in the Game class are also very important. The unique variable and colors variable stores one pointer to each Card that way they can be deleted preventing memory leaks.

Code Topic Location

Topic	Where Line #"s
Classes	Card.h #13
Instance of a Class	main.cpp #47
Private Data Members	Game.h #20-28
Specification vs. Implementation	Game.h and Game.cpp
Inline	Player.h #38
Constructors	Game.h #40 #41
Destructors	Game.h #42
Arrays of Objects	Game.h #24
UML	Write up
More about Classes	
Static	Player.h #17
Friends	Player.h #43
Copy Constructors	Vector.h #21
Operator Overloading	Vector.h #41 #42 #48 Player.h #39
Aggregation	Game.h #27 #28
Inheritance	
Protected members	Player.h #22-30
Base Class to Derived	Computer.h #12 from Player.h
Polymorphic associations	Player.h #34 #35 Game.h #28

Abstract Classes	Card.h Game.h Player.h Computer.h Vector.h
Advanced Classes	
Exceptions	Vector.h #23-40 Vector.h #109 #142
Templates	Vector.h #10/the whole class
STL	Player.cpp #176
Sum	

References

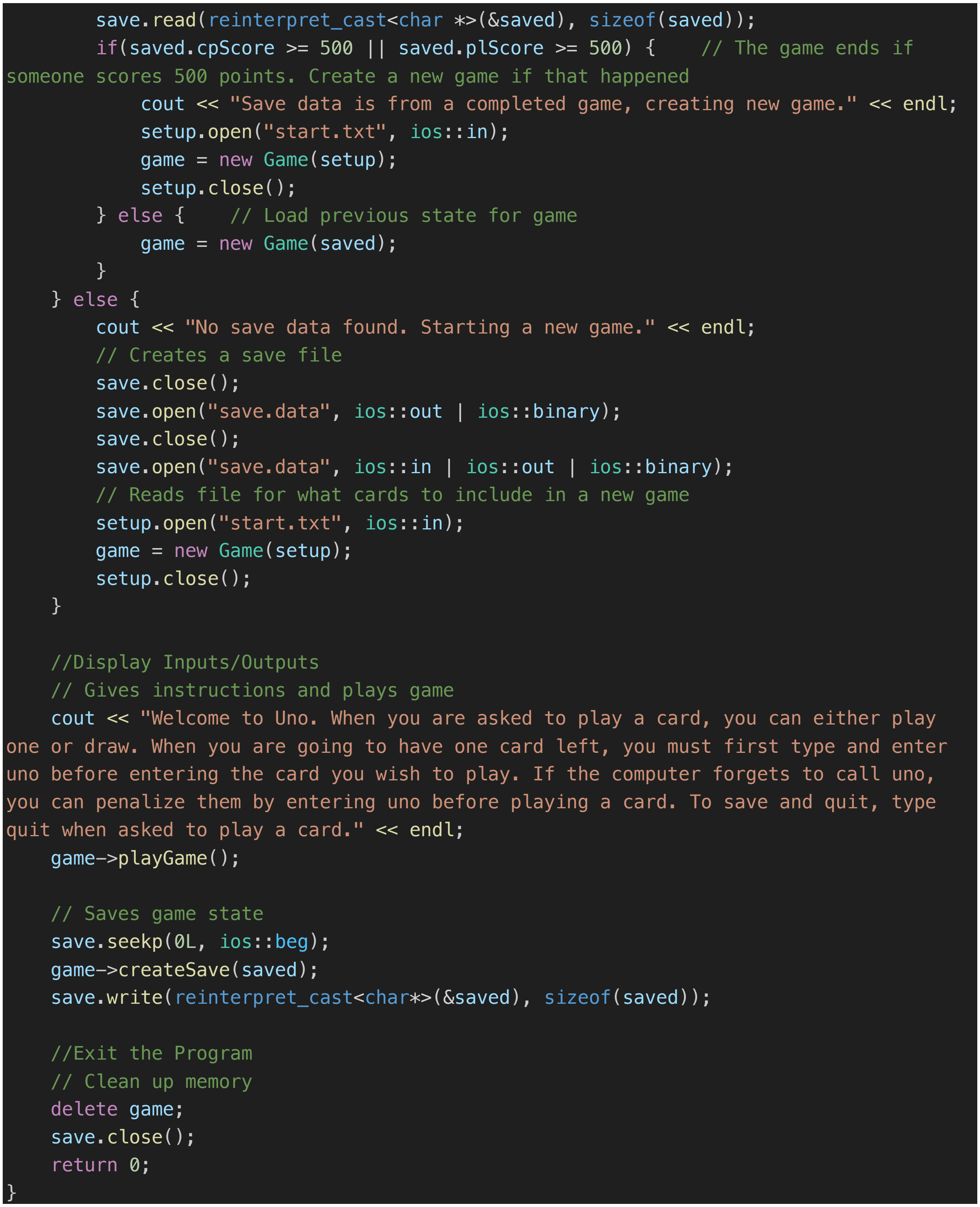
The program's structure was largely inspired by other projects I have written on my own for a game. Small snippets of code were taken from the class textbook such as operator overloading for []. This program was also largely based on Project 1's program. The references for that program will be below.

Project 1's program barrowed a lot from many of my previous projects both in classes and personal ones. One of these was the idea of working with purely pointers which was inspired by a project from my CIS-5 class. The input validation exists largely due to me experimenting with it during my midterm for CIS-17A even if it's not as good as the input validations in the midterm. Nothing was directly copied from my previous projects but I did copy small snippets of code from the class textbook including the demonstration of using rands with ctime and the equation for generating random numbers.

Below is my code

Main.cpp

```cpp
/*
 * File:   main.cpp
 * Author: Matthew Madrigal
 * Created on October 31st, 2025, 3:52 pm,
 * Purpose:  To create an Uno game in C++ using what has been learned so far. These are the rules used for this game: https://service.mattel.com/instruction_sheets/42001pr.pdf
 *           The blank cards are not included since they are for custom rules or replacing lost cards.
 */

//System Libraries
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <fstream>
using namespace std;

//User Libraries
#include "Card.h"
#include "Game.h"
#include "GameS.h"

//Global Constants - Math/Science/Conversions only

//Function Prototypes


//Execution Begins Here
int main(int argc, char** argv) {
    //Set random seed
    srand(time(0));
    //Declare Variables
    Game *game;
    GameS saved;
    fstream setup;  // File used to setup the draw pile for a new game
    fstream save;   // File used to restore the state of the program from when it was last ran

    //Initialize Variables

    //The Process -> Map Inputs to Outputs
    // Checks if a save file exists and if the game didn't end yet
    save.open("save.data", ios::in | ios::out | ios::binary);
    if(save.is_open()) {
        cout << "Save data found. Resuming using save data." << endl;
```



```
        save.read(reinterpret_cast<char *>(&saved), sizeof(saved));
        if(saved.cpScore >= 500 || saved.plScore >= 500) {    // The game ends if
someone scores 500 points. Create a new game if that happened
            cout << "Save data is from a completed game, creating new game." << endl;
            setup.open("start.txt", ios::in);
            game = new Game(setup);
            setup.close();
        } else {    // Load previous state for game
            game = new Game(saved);
        }
    } else {
        cout << "No save data found. Starting a new game." << endl;
        // Creates a save file
        save.close();
        save.open("save.data", ios::out | ios::binary);
        save.close();
        save.open("save.data", ios::in | ios::out | ios::binary);
        // Reads file for what cards to include in a new game
        setup.open("start.txt", ios::in);
        game = new Game(setup);
        setup.close();
    }

    //Display Inputs/Outputs
    // Gives instructions and plays game
    cout << "Welcome to Uno. When you are asked to play a card, you can either play
one or draw. When you are going to have one card left, you must first type and enter
uno before entering the card you wish to play. If the computer forgets to call uno,
you can penalize them by entering uno before playing a card. To save and quit, type
quit when asked to play a card." << endl;
    game->playGame();

    // Saves game state
    save.seekp(0L, ios::beg);
    game->createSave(saved);
    save.write(reinterpret_cast<char*>(&saved), sizeof(saved));

    //Exit the Program
    // Clean up memory
    delete game;
    save.close();
    return 0;
}
```

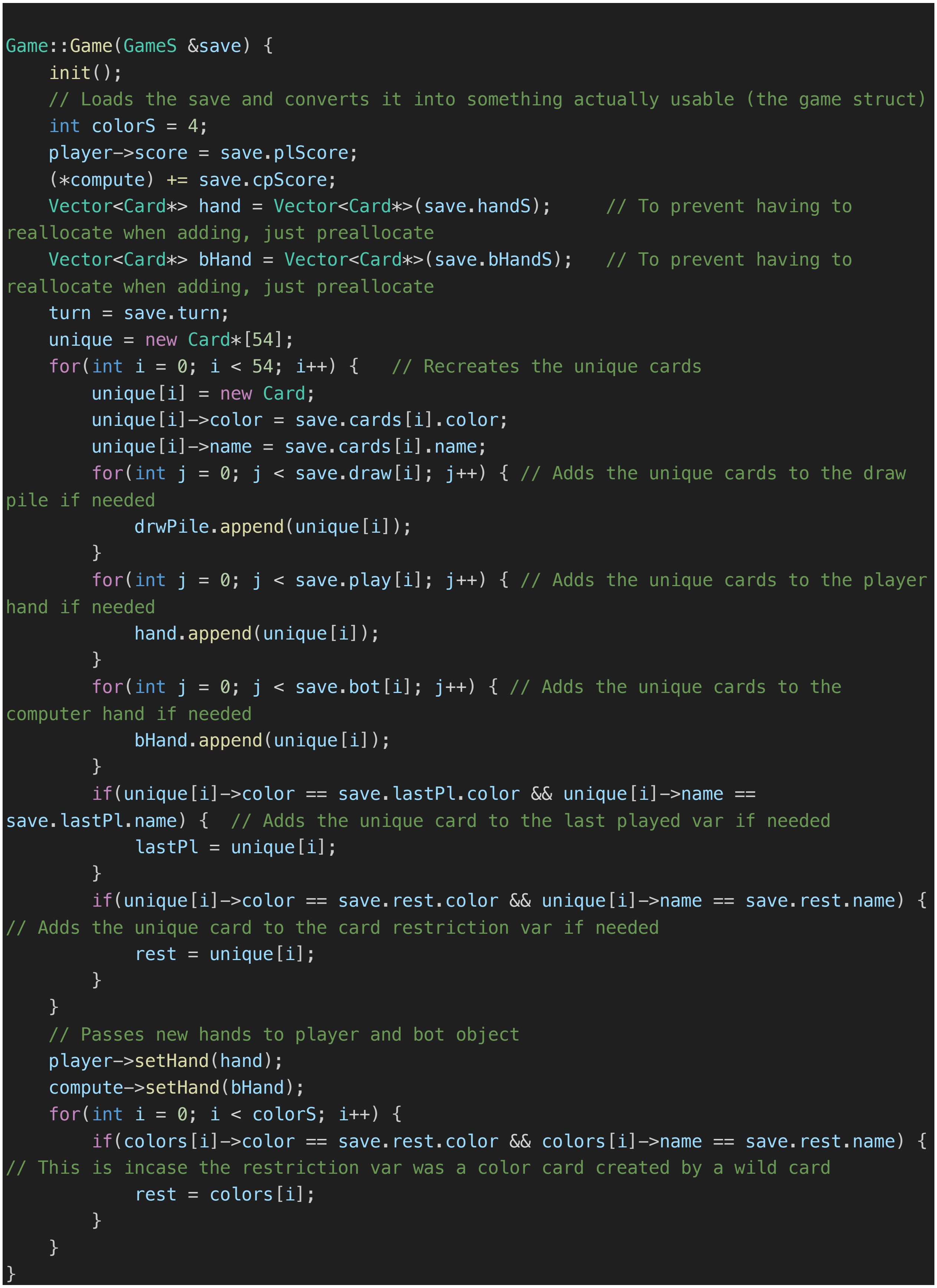
Game.h

```
/*
 * File:   Game.h
 * Author: Matthew Madrigal
 * Purpose:  Game class
 */

#ifndef GAME_H
#define GAME_H

#include "Card.h"
#include "GameS.h"
#include "Player.h"
#include <fstream>
using namespace std;

class Game{
private:
    enum colors {RED, GREEN, YELLOW, BLUE, WILD};
    enum types {ZERO, ONE, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, PLUST,
REV, SKIP, PLUSF, CARD};
    bool turn;                                  // True is for the player, false is
for the computer
    Card *lastPl;                               // The last played card
    Card *rest;                                 // The current restriction on what can
be played
    Vector<Card*> drwPile;                      // The cards waiting to be drawn that
way a game never has more than the max number according to the uno rules
    Card **colors;                              // Wild Card restrictions
    Card **unique;                              // Instead of creating duplicate cards
when there is more than one type of card, a single card is created and that pointer is
duplicated the number of times needed in the draw pile.
    int colorS;                                 // The number of color cards in the
color card array
    Player *player;                             // Player object
    Player *compute;                            // Computer object
    Card **genColor();                          // Generates the restriction cards
used when a wild card is played. They only restrict by color
    void setupPile(fstream &);                  // Gets a file and reads from it to
fill the draw pile with the correct cards. Makes modifying rules a bit easier
    void init();                                // Initializes variables to 0
    void setupGame();                           // Sets up a new game
    int strToCol(string);                       // Converts a string to a color enum
    Card *playWild(int, Card **);               // Asks the user what color to choose
for the wild card
    Card *drawCard(Vector<Card*> &);            // Draws a card from an array and
returns it's pointer
```

```
    string printCard(Card *);                   // Prints a card's color and type
    unsigned int calcPoints(Vector<Card*>);     // Calculates the number of points a person earns for winning
public:

    Game(fstream&);                             // Initializer
    Game(GameS&);                               // Start with save
    ~Game();                                    // Destructor
    void playGame();                            // The main game loop. Also handles special logic for cards such as +2, +4, and calls the playWild function when needed
    void createSave(GameS &);                   // Creates a save before the program terminates
};

#endif /* GAME_H */
```

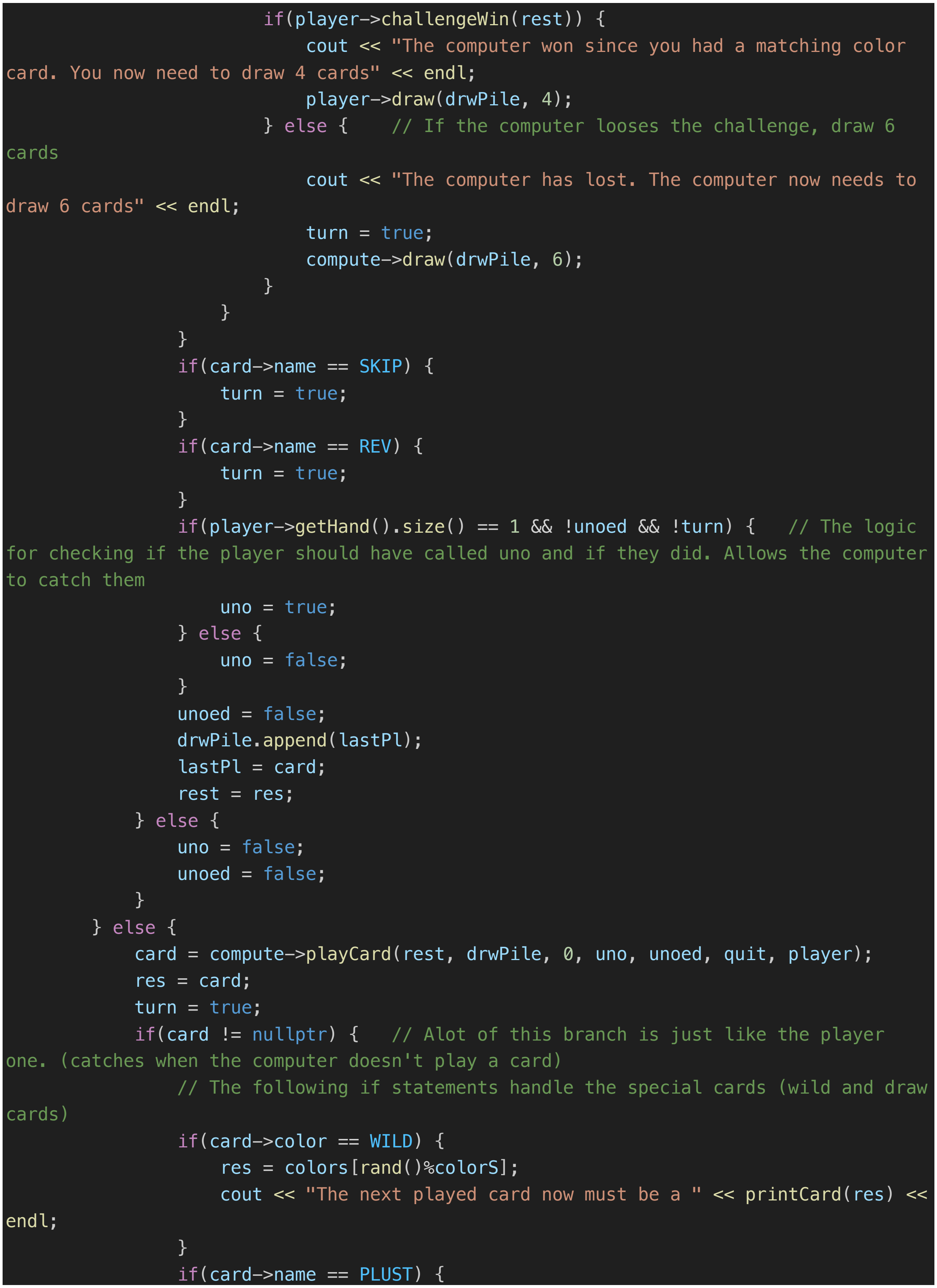
Game.cpp

```cpp
/*
 * File:   Game.cpp
 * Author: Matthew Madrigal
 * Created on December 12th, 2025, 11:39 am,
 * Purpose:  To define the functions in the Game class
 */

//System Libraries
#include <iostream>
using namespace std;

//User Libraries
#include "Game.h"
#include "Card.h"
#include "Player.h"
#include "Computer.h"

Game::Game(fstream &setFile) {
    init();
    setupPile(setFile);
    setupGame();
}

void Game::setupGame() {
    const int start = 7;    // The number of cards each player should start with
    Vector<Card*> nPHand;   // Player hand
    Vector<Card*> nBHand;   // Bot hand
    for(int i = 0; i < start; i++) {    // Fills the computer's and player's hand with cards
        Card *card = drawCard(drwPile);
        nPHand.append(card);
        card = drawCard(drwPile);
        nBHand.append(card);
    }
    player->setHand(nPHand);
    compute->setHand(nBHand);
    Card *card = drawCard(drwPile);
    while(card->color == WILD) {        // If the first played card is a wild, return it to the draw pile and draw another
        drwPile.append(card);
        card = drawCard(drwPile);
    }
    lastPl = card;
    rest = card;
    turn = true;
}
```

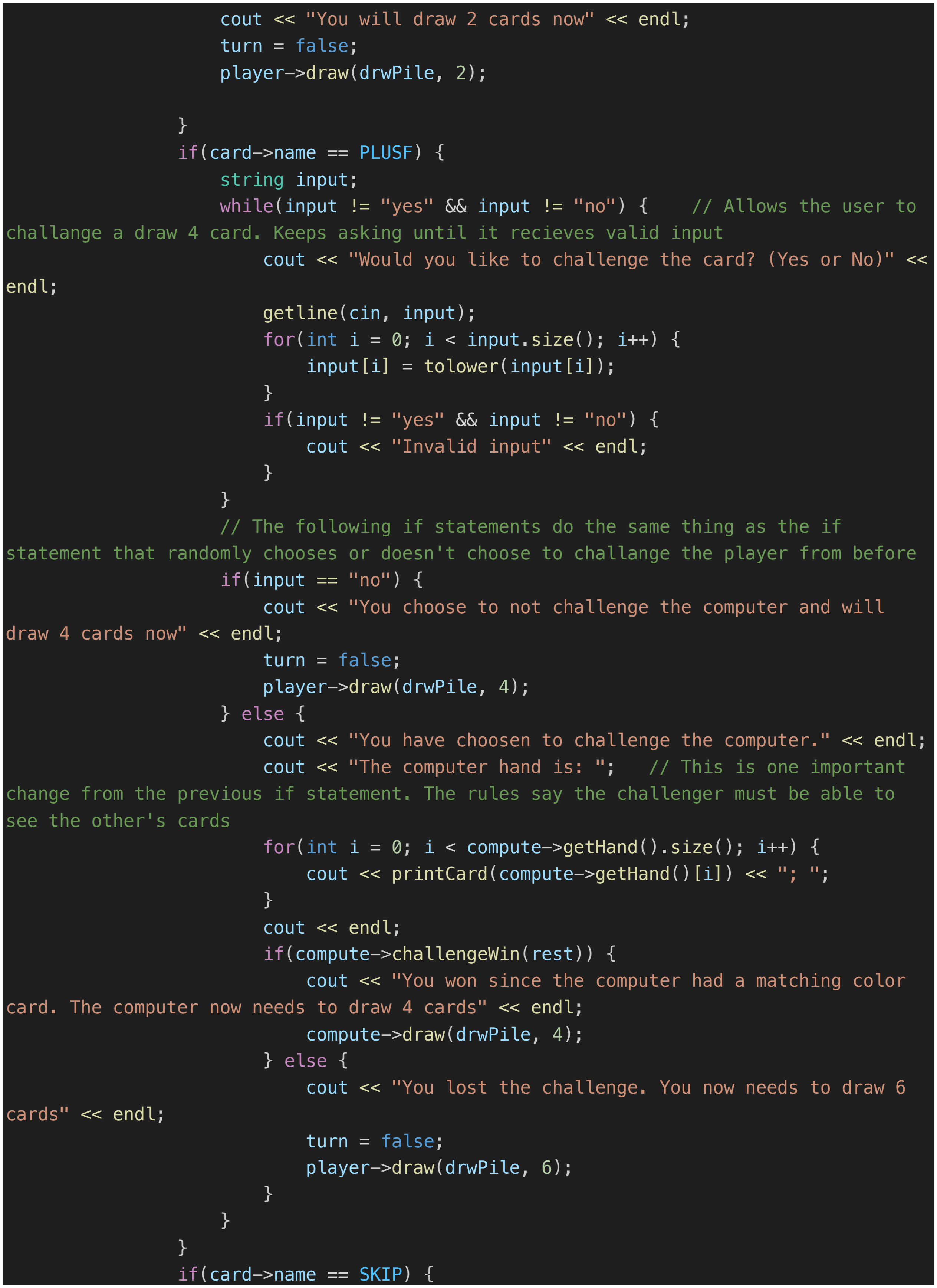


```
Game::Game(GameS &save) {
    init();
    // Loads the save and converts it into something actually usable (the game struct)
    int colorS = 4;
    player->score = save.plScore;
    (*compute) += save.cpScore;
    Vector<Card*> hand = Vector<Card*>(save.handS);     // To prevent having to reallocate when adding, just preallocate
    Vector<Card*> bHand = Vector<Card*>(save.bHandS);   // To prevent having to reallocate when adding, just preallocate
    turn = save.turn;
    unique = new Card*[54];
    for(int i = 0; i < 54; i++) {   // Recreates the unique cards
        unique[i] = new Card;
        unique[i]->color = save.cards[i].color;
        unique[i]->name = save.cards[i].name;
        for(int j = 0; j < save.draw[i]; j++) { // Adds the unique cards to the draw pile if needed
            drwPile.append(unique[i]);
        }
        for(int j = 0; j < save.play[i]; j++) { // Adds the unique cards to the player hand if needed
            hand.append(unique[i]);
        }
        for(int j = 0; j < save.bot[i]; j++) { // Adds the unique cards to the computer hand if needed
            bHand.append(unique[i]);
        }
        if(unique[i]->color == save.lastPl.color && unique[i]->name == save.lastPl.name) {  // Adds the unique card to the last played var if needed
            lastPl = unique[i];
        }
        if(unique[i]->color == save.rest.color && unique[i]->name == save.rest.name) { // Adds the unique card to the card restriction var if needed
            rest = unique[i];
        }
    }
    // Passes new hands to player and bot object
    player->setHand(hand);
    compute->setHand(bHand);
    for(int i = 0; i < colorS; i++) {
        if(colors[i]->color == save.rest.color && colors[i]->name == save.rest.name) { // This is incase the restriction var was a color card created by a wild card
            rest = colors[i];
        }
    }
}
```

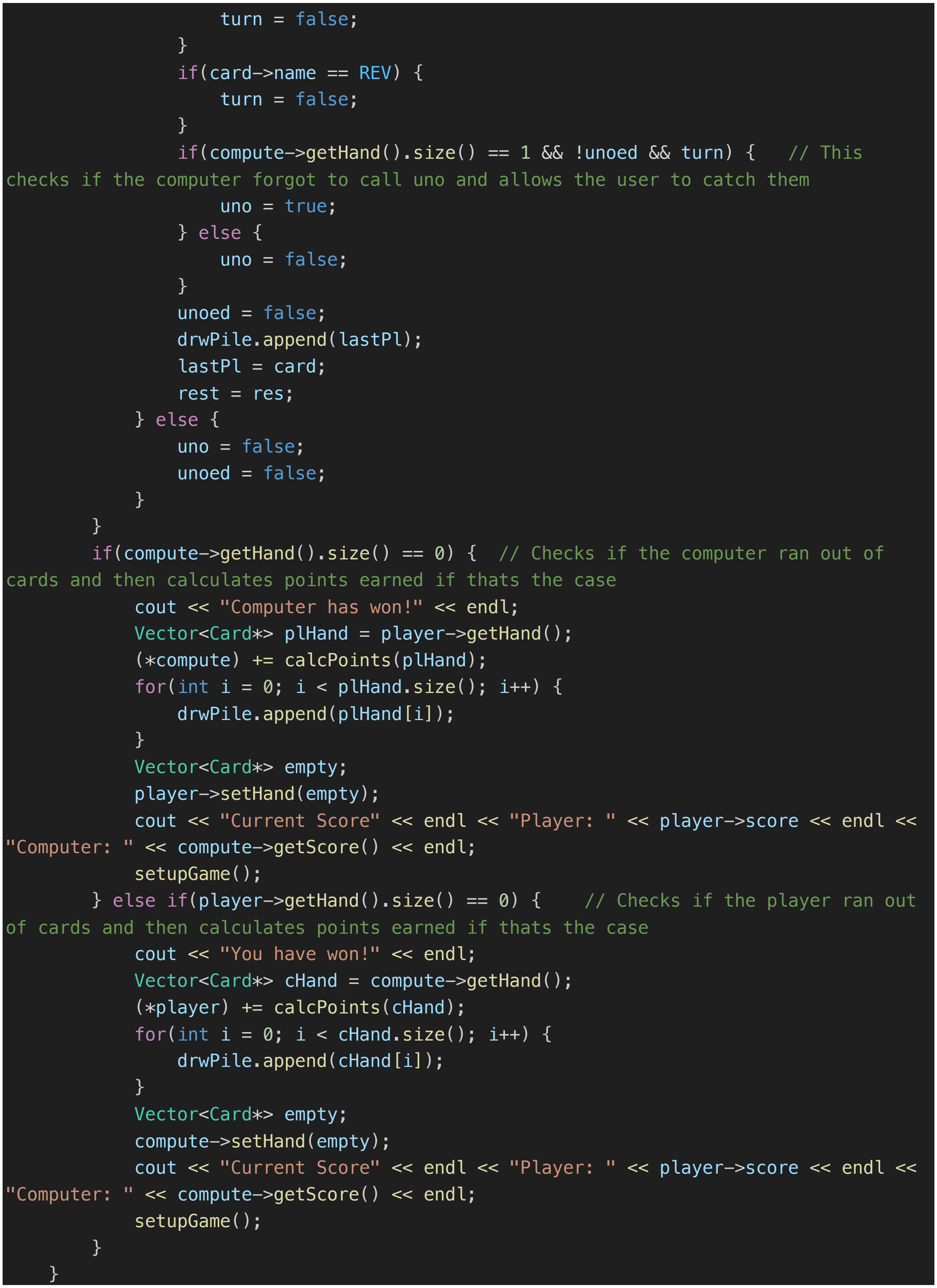
```cpp
void Game::init() {
    //Initialize Variables
    colorS = 0;
    colors = genColor();
    player = new Player();
    compute = new Computer();
    cout << "Game will have " << player->getCount() << " players/computers" << endl;
}

void Game::playGame() {
    bool uno = false, unoed = false, quit = false;  // Used to keep track of things in between turns
    while(player->score < 500 && compute->getScore() < 500) {   // A game ends when someone has 500 points or more
        Card *card;
        Card *res;
        if(turn) {
            cout << "Current card at play: " << lastPl->printCard() << endl;
            card = player->playCard(rest, drwPile, compute->getHand().size(), uno, unoed, quit, compute);
            if(quit) {  // Allows the player to exit the game
                break;
            }
            res = card;
            turn = false;
            if(card != nullptr) {   // This is to catch when the player draws a card and doesn't play it (or if they couldn't do anything)
                // The following if statements handle the special cards (wild and draw cards)
                if(card->color == WILD) {
                    res = playWild(colorS, colors);
                }
                if(card->name == PLUST) {
                    cout << "The computer will draw 2 cards now" << endl;
                    turn = true;
                    compute->draw(drwPile, 2);

                }
                if(card->name == PLUSF) {
                    if(rand()%2 == 1) { // The computer chooses randomly if it wants to challange you or not
                        cout << "The computer choose to not challenge you and will draw 4 cards now" << endl;
                        turn = true;
                        compute->draw(drwPile, 4);
                    } else {
                        cout << "The computer has choosen to challenge you." << endl;
```



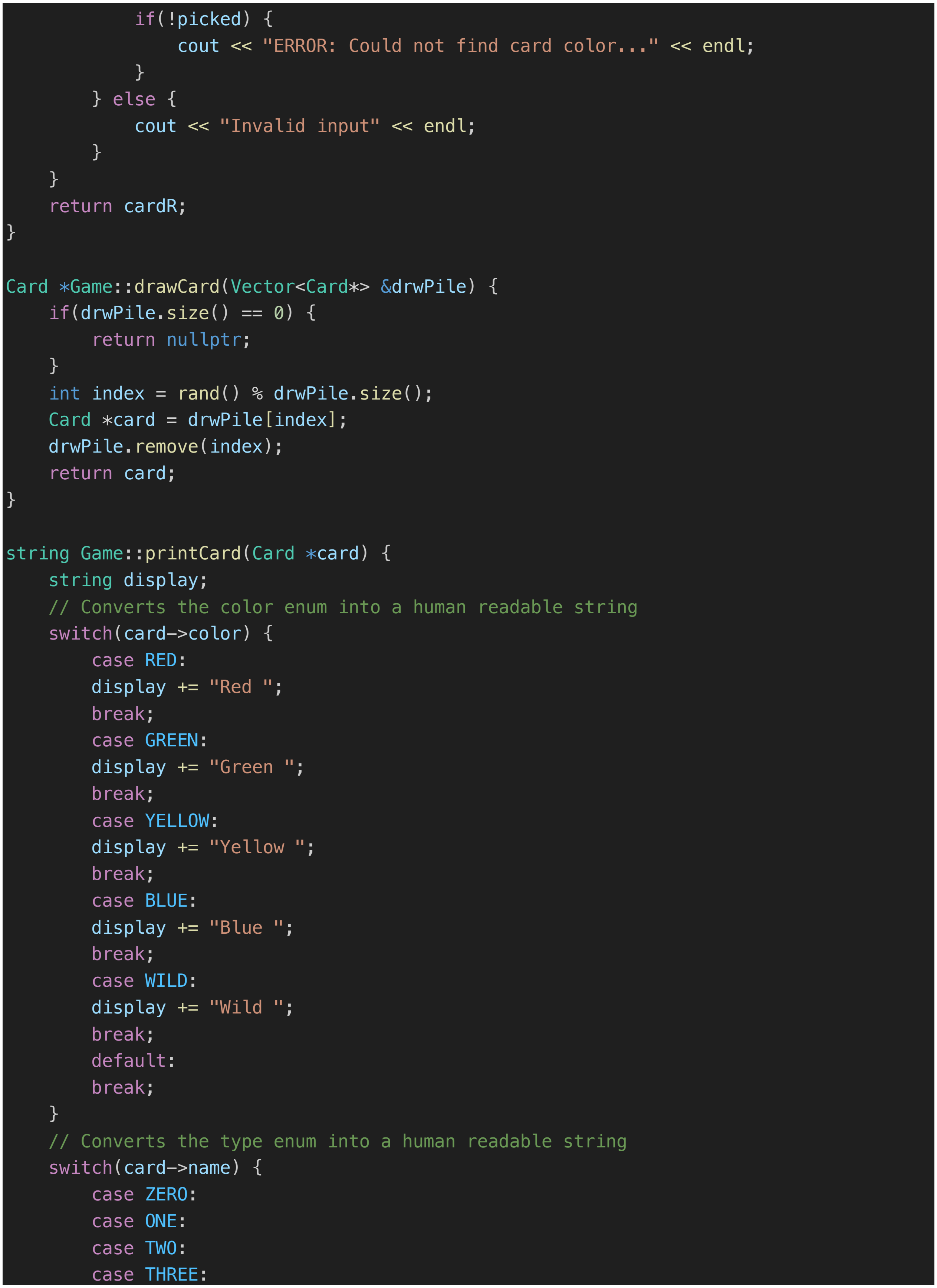
```
                        if(player->challengeWin(rest)) {
                            cout << "The computer won since you had a matching color
card. You now need to draw 4 cards" << endl;
                            player->draw(drwPile, 4);
                        } else {    // If the computer looses the challenge, draw 6
cards
                            cout << "The computer has lost. The computer now needs to
draw 6 cards" << endl;
                            turn = true;
                            compute->draw(drwPile, 6);
                        }
                    }
                }
                if(card->name == SKIP) {
                    turn = true;
                }
                if(card->name == REV) {
                    turn = true;
                }
                if(player->getHand().size() == 1 && !unoed && !turn) {   // The logic
for checking if the player should have called uno and if they did. Allows the computer
to catch them
                    uno = true;
                } else {
                    uno = false;
                }
                unoed = false;
                drwPile.append(lastPl);
                lastPl = card;
                rest = res;
            } else {
                uno = false;
                unoed = false;
            }
        } else {
            card = compute->playCard(rest, drwPile, 0, uno, unoed, quit, player);
            res = card;
            turn = true;
            if(card != nullptr) {   // Alot of this branch is just like the player
one. (catches when the computer doesn't play a card)
                // The following if statements handle the special cards (wild and draw
cards)
                if(card->color == WILD) {
                    res = colors[rand()%colorS];
                    cout << "The next played card now must be a " << printCard(res) <<
endl;
                }
                if(card->name == PLUST) {
```



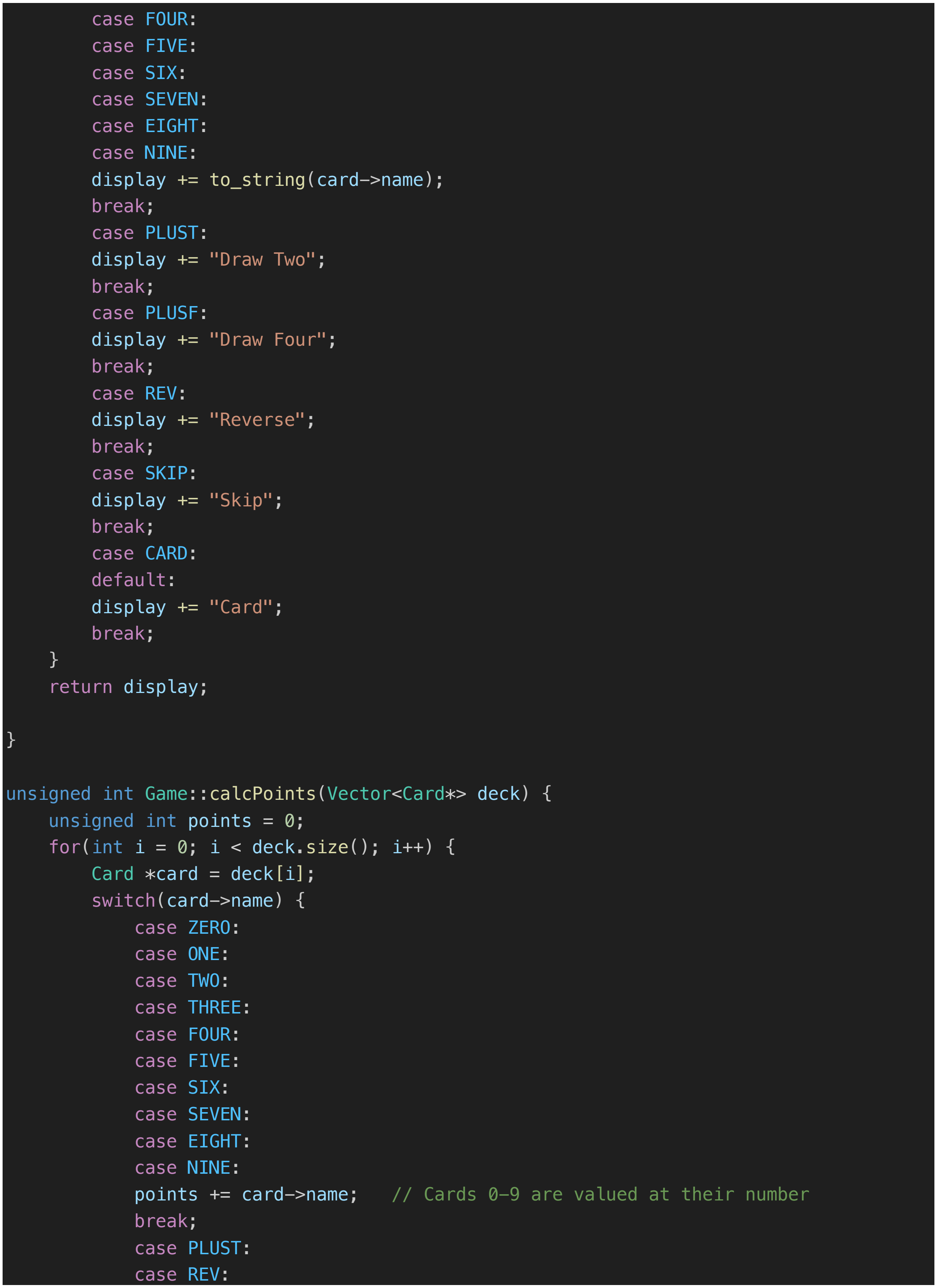
```
                    cout << "You will draw 2 cards now" << endl;
                    turn = false;
                    player->draw(drwPile, 2);

                }
                if(card->name == PLUSF) {
                    string input;
                    while(input != "yes" && input != "no") {    // Allows the user to
challange a draw 4 card. Keeps asking until it recieves valid input
                        cout << "Would you like to challenge the card? (Yes or No)" <<
endl;
                        getline(cin, input);
                        for(int i = 0; i < input.size(); i++) {
                            input[i] = tolower(input[i]);
                        }
                        if(input != "yes" && input != "no") {
                            cout << "Invalid input" << endl;
                        }
                    }
                    // The following if statements do the same thing as the if
statement that randomly chooses or doesn't choose to challange the player from before
                    if(input == "no") {
                        cout << "You choose to not challenge the computer and will
draw 4 cards now" << endl;
                        turn = false;
                        player->draw(drwPile, 4);
                    } else {
                        cout << "You have choosen to challenge the computer." << endl;
                        cout << "The computer hand is: ";   // This is one important
change from the previous if statement. The rules say the challenger must be able to
see the other's cards
                        for(int i = 0; i < compute->getHand().size(); i++) {
                            cout << printCard(compute->getHand()[i]) << "; ";
                        }
                        cout << endl;
                        if(compute->challengeWin(rest)) {
                            cout << "You won since the computer had a matching color
card. The computer now needs to draw 4 cards" << endl;
                            compute->draw(drwPile, 4);
                        } else {
                            cout << "You lost the challenge. You now needs to draw 6
cards" << endl;
                            turn = false;
                            player->draw(drwPile, 6);
                        }
                    }
                }
                if(card->name == SKIP) {
```



```
                    turn = false;
                }
                if(card->name == REV) {
                    turn = false;
                }
                if(compute->getHand().size() == 1 && !unoed && turn) {   // This
checks if the computer forgot to call uno and allows the user to catch them
                    uno = true;
                } else {
                    uno = false;
                }
                unoed = false;
                drwPile.append(lastPl);
                lastPl = card;
                rest = res;
            } else {
                uno = false;
                unoed = false;
            }
        }
        if(compute->getHand().size() == 0) {  // Checks if the computer ran out of
cards and then calculates points earned if thats the case
            cout << "Computer has won!" << endl;
            Vector<Card*> plHand = player->getHand();
            (*compute) += calcPoints(plHand);
            for(int i = 0; i < plHand.size(); i++) {
                drwPile.append(plHand[i]);
            }
            Vector<Card*> empty;
            player->setHand(empty);
            cout << "Current Score" << endl << "Player: " << player->score << endl <<
"Computer: " << compute->getScore() << endl;
            setupGame();
        } else if(player->getHand().size() == 0) {    // Checks if the player ran out
of cards and then calculates points earned if thats the case
            cout << "You have won!" << endl;
            Vector<Card*> cHand = compute->getHand();
            (*player) += calcPoints(cHand);
            for(int i = 0; i < cHand.size(); i++) {
                drwPile.append(cHand[i]);
            }
            Vector<Card*> empty;
            compute->setHand(empty);
            cout << "Current Score" << endl << "Player: " << player->score << endl <<
"Computer: " << compute->getScore() << endl;
            setupGame();
        }
    }
    if(player->score >= 500 || compute->getScore() >= 500) {    // If someone earns
500+ points, the game ends
        cout << "Game Complete! " << ((compute->getScore() >= 500) ? "Computer" :
"Player") << " won!" << endl;
    }
    cout << "Game will save" << endl;
}

int Game::strToCol(string input) {
    if(input == "red") {
        return RED;
    }
    if(input == "green") {
        return GREEN;
    }
    if(input == "yellow") {
        return YELLOW;
    }
    if(input == "blue") {
        return BLUE;
    }
    if(input == "wild") {
        return WILD;
    }
    return -1;
}

Card *Game::playWild(int size, Card **colors) {
    bool picked = false;
    Card *cardR;    // Card used to restrict to a color
    while(!picked) {    // keeps looping until a color is obtained
        cout << "Choose a color (Red, Green, Yellow, Blue): ";
        string input;
        getline(cin, input);
        for(int i = 0; i < input.length(); i++) {   // Lower case for easier parsing
            input[i] = tolower(input[i]);
        }
        int color;
        color = strToCol(input);
        if(color != -1 && color != WILD) {
            for(int i = 0; i < size; i++) {
                Card *card = colors[i];
                if(card->color == color) {
                    picked = true;
                    cardR = card;
                    break;
                }
            }
```

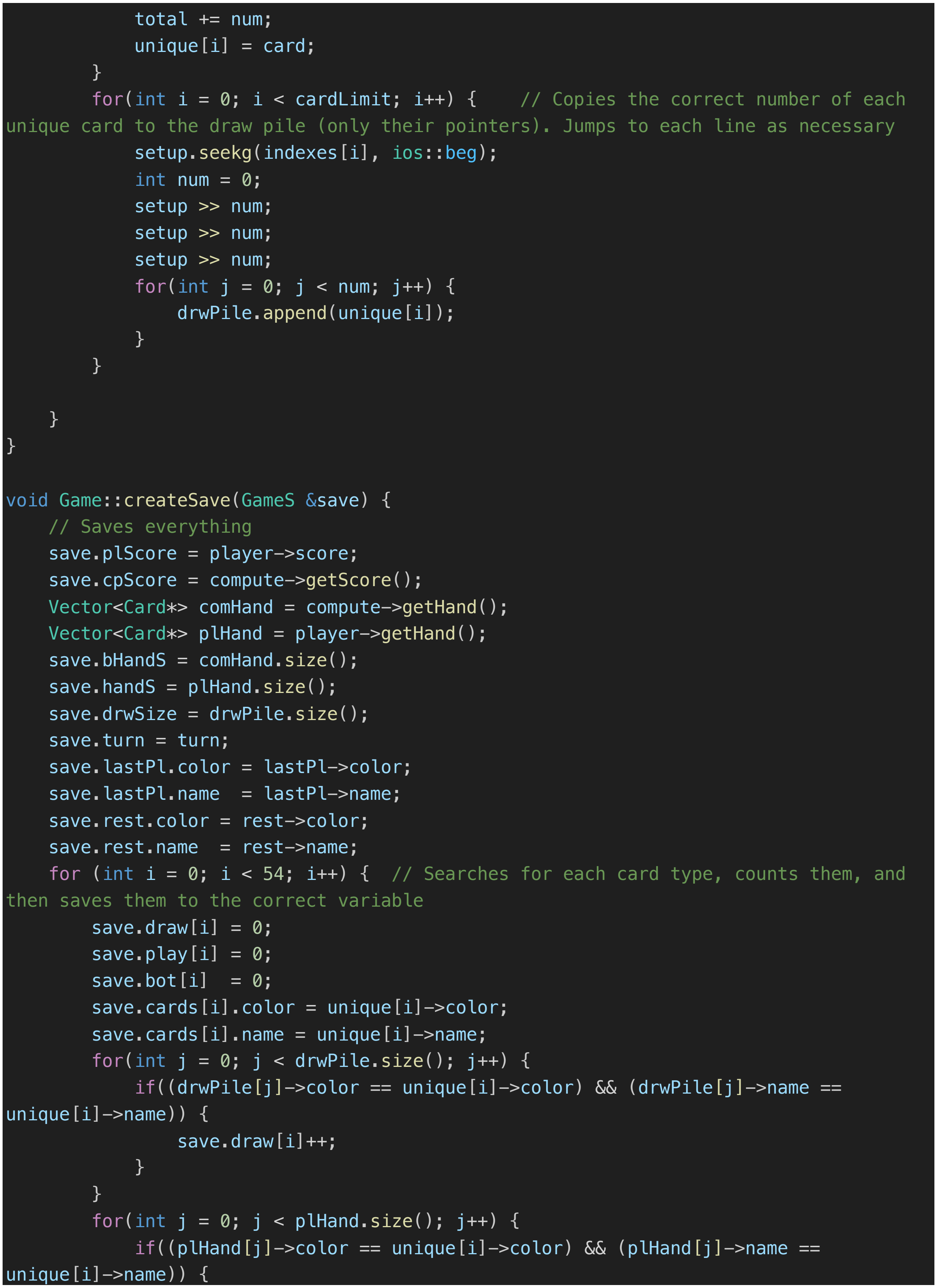


```
            if(!picked) {
                cout << "ERROR: Could not find card color..." << endl;
            }
        } else {
            cout << "Invalid input" << endl;
        }
    }
    return cardR;
}

Card *Game::drawCard(Vector<Card*> &drwPile) {
    if(drwPile.size() == 0) {
        return nullptr;
    }
    int index = rand() % drwPile.size();
    Card *card = drwPile[index];
    drwPile.remove(index);
    return card;
}

string Game::printCard(Card *card) {
    string display;
    // Converts the color enum into a human readable string
    switch(card->color) {
        case RED:
        display += "Red ";
        break;
        case GREEN:
        display += "Green ";
        break;
        case YELLOW:
        display += "Yellow ";
        break;
        case BLUE:
        display += "Blue ";
        break;
        case WILD:
        display += "Wild ";
        break;
        default:
        break;
    }
    // Converts the type enum into a human readable string
    switch(card->name) {
        case ZERO:
        case ONE:
        case TWO:
        case THREE:
```

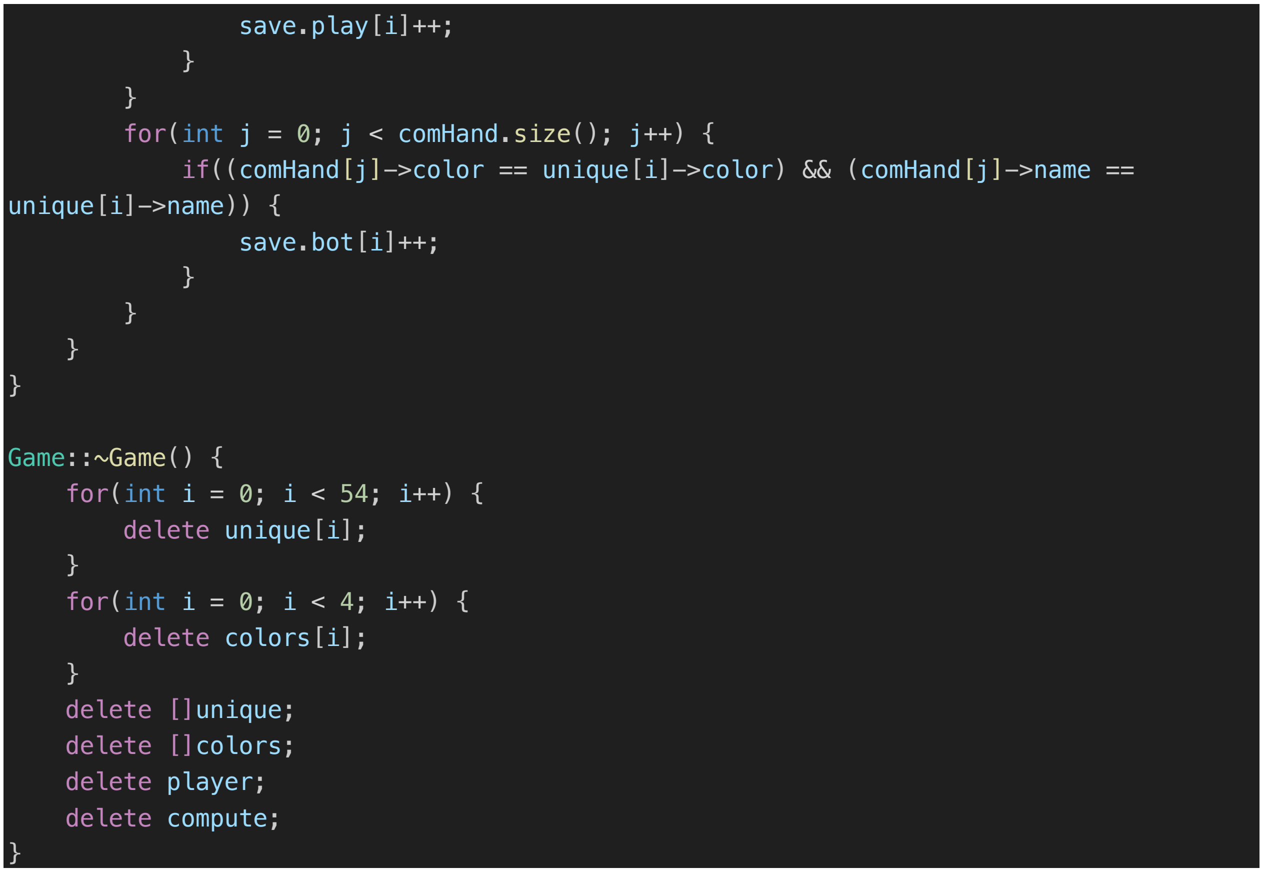


```
        case FOUR:
        case FIVE:
        case SIX:
        case SEVEN:
        case EIGHT:
        case NINE:
        display += to_string(card->name);
        break;
        case PLUST:
        display += "Draw Two";
        break;
        case PLUSF:
        display += "Draw Four";
        break;
        case REV:
        display += "Reverse";
        break;
        case SKIP:
        display += "Skip";
        break;
        case CARD:
        default:
        display += "Card";
        break;
    }
    return display;

}

unsigned int Game::calcPoints(Vector<Card*> deck) {
    unsigned int points = 0;
    for(int i = 0; i < deck.size(); i++) {
        Card *card = deck[i];
        switch(card->name) {
            case ZERO:
            case ONE:
            case TWO:
            case THREE:
            case FOUR:
            case FIVE:
            case SIX:
            case SEVEN:
            case EIGHT:
            case NINE:
            points += card->name;   // Cards 0-9 are valued at their number
            break;
            case PLUST:
            case REV:
```

```
            case SKIP:
            points += 20;           // Special non wild cards are valued at 20 points
each
            break;
            case CARD:
            case PLUSF:
            points += 50;           // Wild cards (type CARD and PLUSF will always be
a wild) are valued at 50 points each
            break;
            default:
            break;
        }
    }
    return points;
}

Card **Game::genColor() {
    colors = new Card*[4];
    colors[0] = new Card{RED, CARD};
    colors[1] = new Card{GREEN, CARD};
    colors[2] = new Card{YELLOW, CARD};
    colors[3] = new Card{BLUE, CARD};
    colorS = 4;
    return colors;
}

void Game::setupPile(fstream &setup) {
    if(drwPile.size() == 0) {   // Makes sure this function doesn't run if the draw
pile is already setup
        const int cardLimit = 54;   // Uno only has 54 unique cards under normal rules
        long int indexes[cardLimit] = {};   // Used to jump around to the correct
spots in the file
        unique = new Card*[cardLimit];
        int total = 0;
        for(int i = 1; i < cardLimit; i++) {    // Finds where the new lines are to
get the position each card is located at in the file (first one is alwasy 0)
            string line;
            getline(setup, line);
            indexes[i] = setup.tellg();
        }
        for(int i = 0; i < cardLimit; i++) {    // Starts by jumping to each index to
create each unique card and counts how large the draw pile needs to be
            setup.seekg(indexes[i], ios::beg);
            Card *card = new Card;
            int num = 0;
            setup >> card->color;
            setup >> card->name;
            setup >> num;
```



```
            total += num;
            unique[i] = card;
        }
        for(int i = 0; i < cardLimit; i++) {    // Copies the correct number of each
unique card to the draw pile (only their pointers). Jumps to each line as necessary
            setup.seekg(indexes[i], ios::beg);
            int num = 0;
            setup >> num;
            setup >> num;
            setup >> num;
            for(int j = 0; j < num; j++) {
                drwPile.append(unique[i]);
            }
        }

    }
}

void Game::createSave(GameS &save) {
    // Saves everything
    save.plScore = player->score;
    save.cpScore = compute->getScore();
    Vector<Card*> comHand = compute->getHand();
    Vector<Card*> plHand = player->getHand();
    save.bHandS = comHand.size();
    save.handS = plHand.size();
    save.drwSize = drwPile.size();
    save.turn = turn;
    save.lastPl.color = lastPl->color;
    save.lastPl.name  = lastPl->name;
    save.rest.color = rest->color;
    save.rest.name  = rest->name;
    for (int i = 0; i < 54; i++) {  // Searches for each card type, counts them, and
then saves them to the correct variable
        save.draw[i] = 0;
        save.play[i] = 0;
        save.bot[i]  = 0;
        save.cards[i].color = unique[i]->color;
        save.cards[i].name = unique[i]->name;
        for(int j = 0; j < drwPile.size(); j++) {
            if((drwPile[j]->color == unique[i]->color) && (drwPile[j]->name ==
unique[i]->name)) {
                save.draw[i]++;
            }
        }
        for(int j = 0; j < plHand.size(); j++) {
            if((plHand[j]->color == unique[i]->color) && (plHand[j]->name ==
unique[i]->name)) {
```



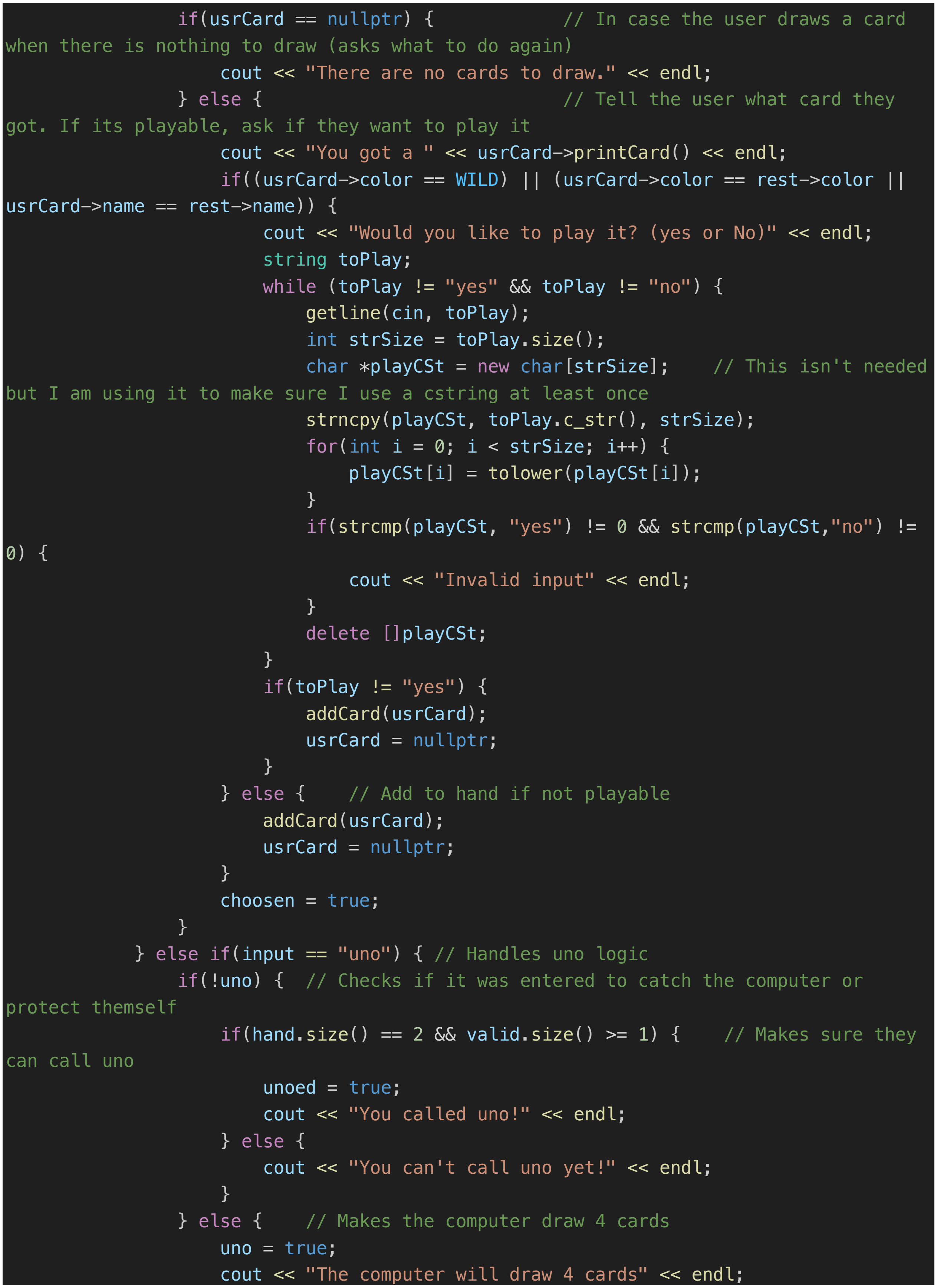
```
                save.play[i]++;
            }
        }
        for(int j = 0; j < comHand.size(); j++) {
            if((comHand[j]->color == unique[i]->color) && (comHand[j]->name ==
unique[i]->name)) {
                save.bot[i]++;
            }
        }
    }
}

Game::~Game() {
    for(int i = 0; i < 54; i++) {
        delete unique[i];
    }
    for(int i = 0; i < 4; i++) {
        delete colors[i];
    }
    delete []unique;
    delete []colors;
    delete player;
    delete compute;
}
```

Player.h

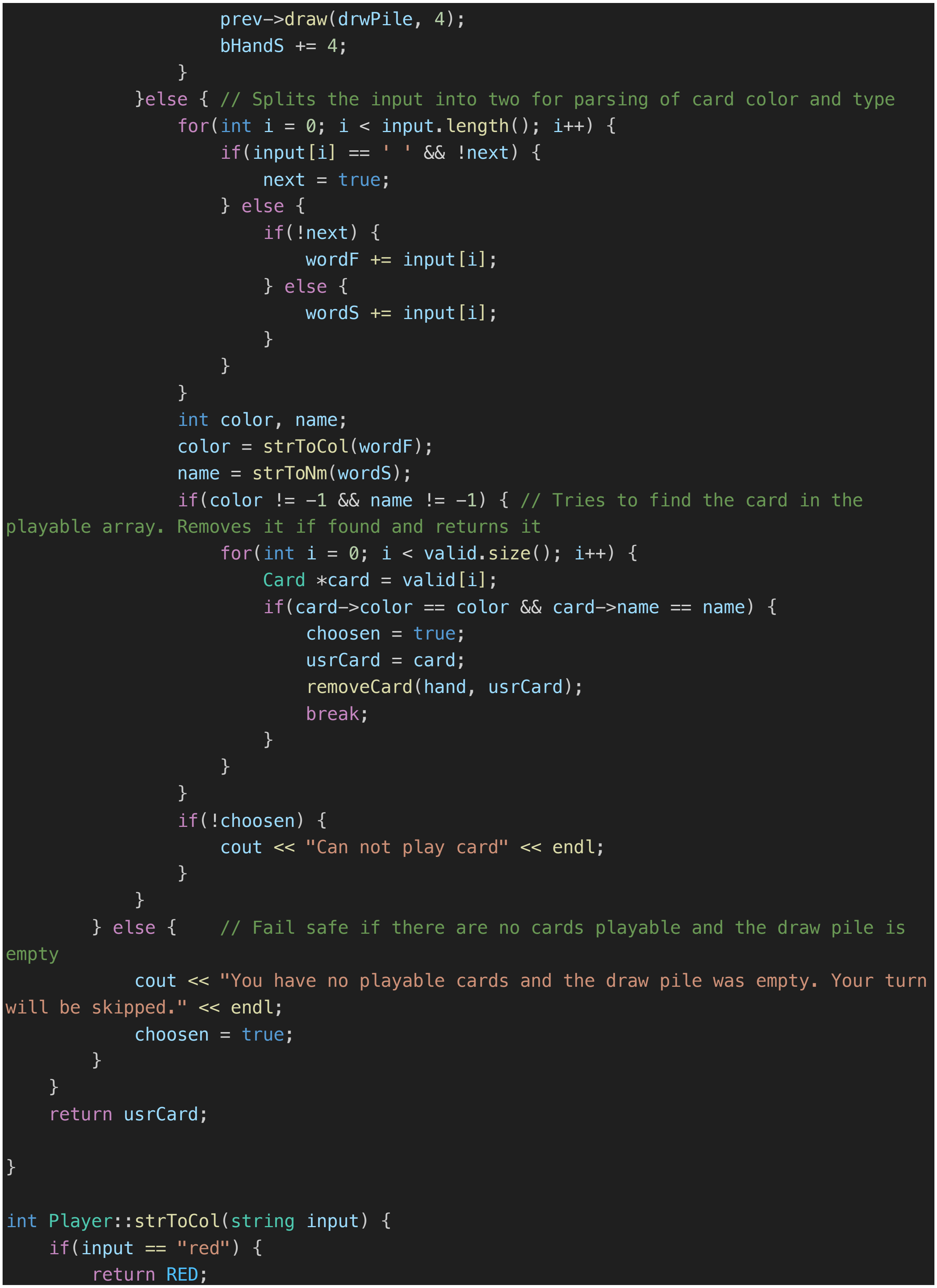
```
/*
 * File:   Player.h
 * Author: Matthew Madrigal
 * Purpose:  Player Class
 */

#ifndef PLAYER_H
#define PLAYER_H

#include "Vector.h"
#include "Card.h"

class Game; // Used for friend

class Player{
private:
    static int players;                     // Keeps track of the number of players
    int score;                              // Keeps track of score
    int strToCol(string);                   // Converts a string to a color enum
    int strToNm(string);                    // Converts a string to a type enum
    void printDeck(Card *);                 // Prints what cards are in the array and
what cards can be played.
protected:
    enum colors {RED, GREEN, YELLOW, BLUE, WILD};
    enum types {ZERO, ONE, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, PLUST,
REV, SKIP, PLUSF, CARD};
    Vector<Card*> hand;
    Vector<Card*> getPlayable(Card*);
    Card *drawCard(Vector<Card*> &);                     // Draws a card from an array
and returns it's pointer
    void addCard(Card *);       // Adds a card to an array. If needed, it will grow
the array size
    void removeCard(int &, Card **, Card *);            // Removes a card given its
pointer from an array and decreases its size by one
    void removeCard(Vector<Card*> &, Card *);             // Works with the Vector
object

public:
    Player() {score = 0; players++;};
    virtual ~Player() {}
    virtual Card *playCard(Card*, Vector<Card*> &, int, bool, bool &, bool &, Player
*prev);
    Vector<Card*> getHand() const;
    void setHand(Vector<Card*> &);
    int getScore() const {return score;}
    Player operator+=(const int);
    static int getCount() {return players;}
    virtual void draw(Vector<Card*> &, int);
    bool challengeWin(Card *);
    friend class Game;
};

#endif /* PLAYER_H */
```

Player.cpp

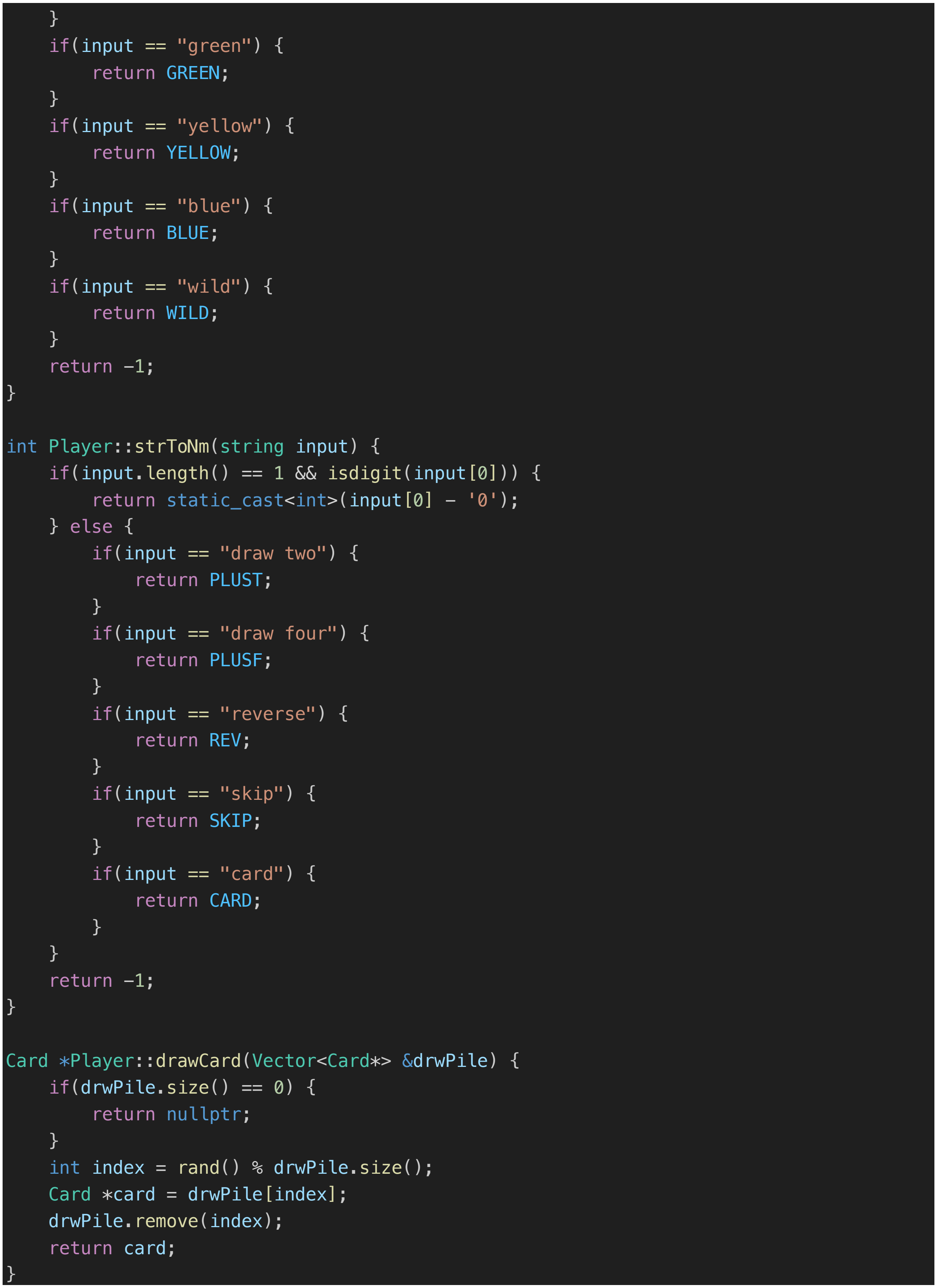
```cpp
/*
 * File:   Player.cpp
 * Author: Matthew Madrigal
 * Created on December 20th, 2025, 7:41 pm,
 * Purpose:  To define the functions in the player class
 */

//System Libraries
#include <iostream>
#include <algorithm>
using namespace std;

//User Libraries
#include "Player.h"

int Player::players = 0;

Card *Player::playCard(Card* rest, Vector<Card*> &drwPile, int bHandS, bool uno, bool &unoed, bool &quit, Player *prev) {
    bool choosen = false;
    Card *usrCard = nullptr;
    while(!choosen) {                                // This forces the user to play a valid card before continuing
        Vector<Card*> valid = getPlayable(rest);
        if(valid.size() != 0 || drwPile.size() != 0) {  // If its possible for the player to do something, allow them to play
            cout << "The computer has " << bHandS << " cards" << endl;
            printDeck(rest);
            cout << "Enter the card you would like to play (Type it as listed)" << ((drwPile.size() > 0) ?" or type draw to draw a card" : "") << (uno? ". Since the computer didn't call uno, you can type uno to force them to draw" : "") << ": ";
            string input, wordF, wordS;
            getline(cin, input);                    // Gets the card the player wants to play
            bool next = false;
            for(int i = 0; i < input.length(); i++) {
                input[i] = tolower(input[i]);
            }
            if(input == "quit") {                   // Special case for quiting the game
                quit = true;
                return nullptr;
            }
            if(input == "draw") {                   // Special case for drawing from the pile
                usrCard = drawCard(drwPile);
```



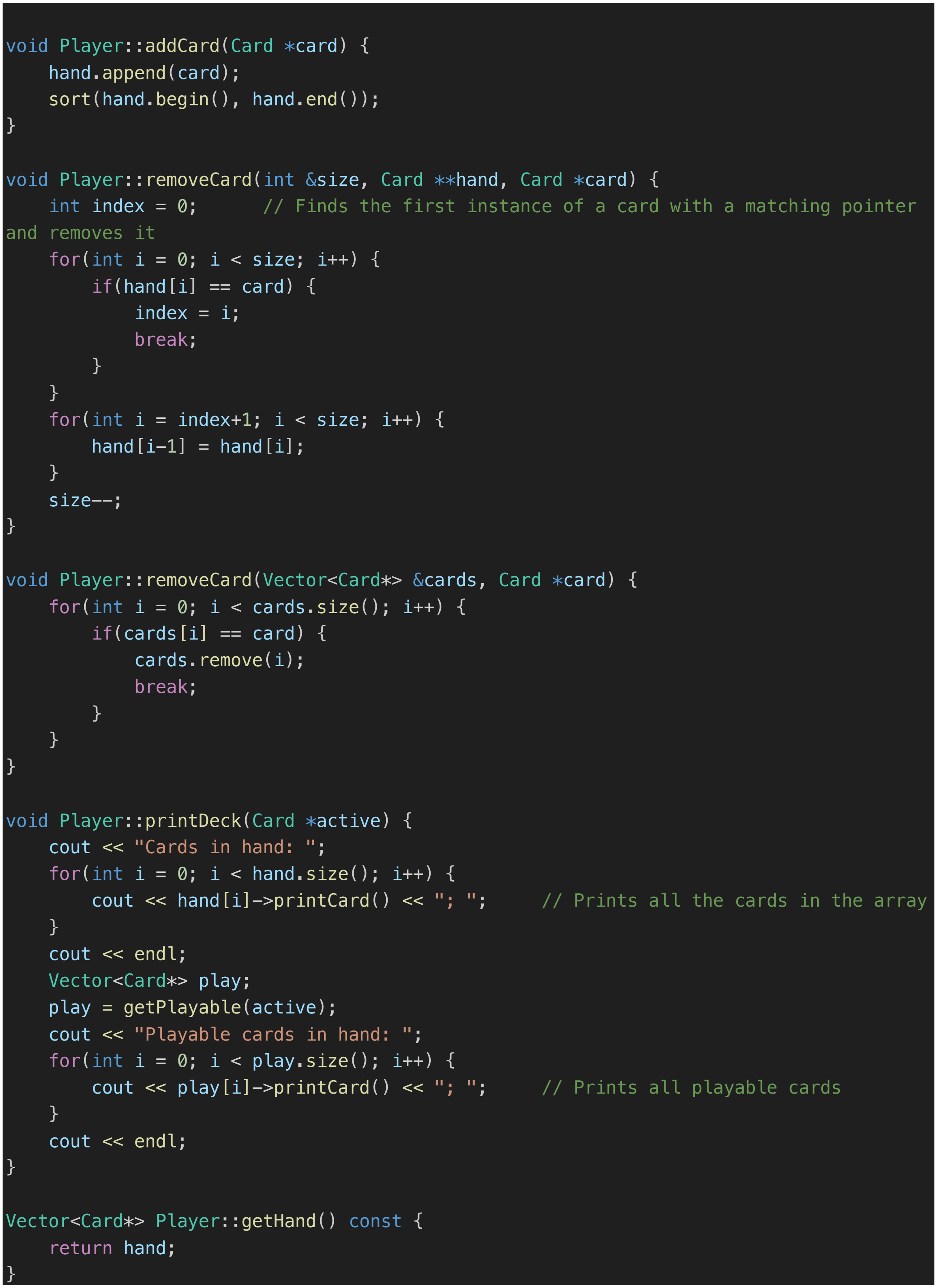
```
                if(usrCard == nullptr) {            // In case the user draws a card
when there is nothing to draw (asks what to do again)
                    cout << "There are no cards to draw." << endl;
                } else {                            // Tell the user what card they
got. If its playable, ask if they want to play it
                    cout << "You got a " << usrCard->printCard() << endl;
                    if((usrCard->color == WILD) || (usrCard->color == rest->color ||
usrCard->name == rest->name)) {
                        cout << "Would you like to play it? (yes or No)" << endl;
                        string toPlay;
                        while (toPlay != "yes" && toPlay != "no") {
                            getline(cin, toPlay);
                            int strSize = toPlay.size();
                            char *playCSt = new char[strSize];    // This isn't needed
but I am using it to make sure I use a cstring at least once
                            strncpy(playCSt, toPlay.c_str(), strSize);
                            for(int i = 0; i < strSize; i++) {
                                playCSt[i] = tolower(playCSt[i]);
                            }
                            if(strcmp(playCSt, "yes") != 0 && strcmp(playCSt,"no") !=
0) {
                                cout << "Invalid input" << endl;
                            }
                            delete []playCSt;
                        }
                        if(toPlay != "yes") {
                            addCard(usrCard);
                            usrCard = nullptr;
                        }
                    } else {    // Add to hand if not playable
                        addCard(usrCard);
                        usrCard = nullptr;
                    }
                    choosen = true;
                }
            } else if(input == "uno") { // Handles uno logic
                if(!uno) {  // Checks if it was entered to catch the computer or
protect themself
                    if(hand.size() == 2 && valid.size() >= 1) {    // Makes sure they
can call uno
                        unoed = true;
                        cout << "You called uno!" << endl;
                    } else {
                        cout << "You can't call uno yet!" << endl;
                    }
                } else {    // Makes the computer draw 4 cards
                    uno = true;
                    cout << "The computer will draw 4 cards" << endl;
```



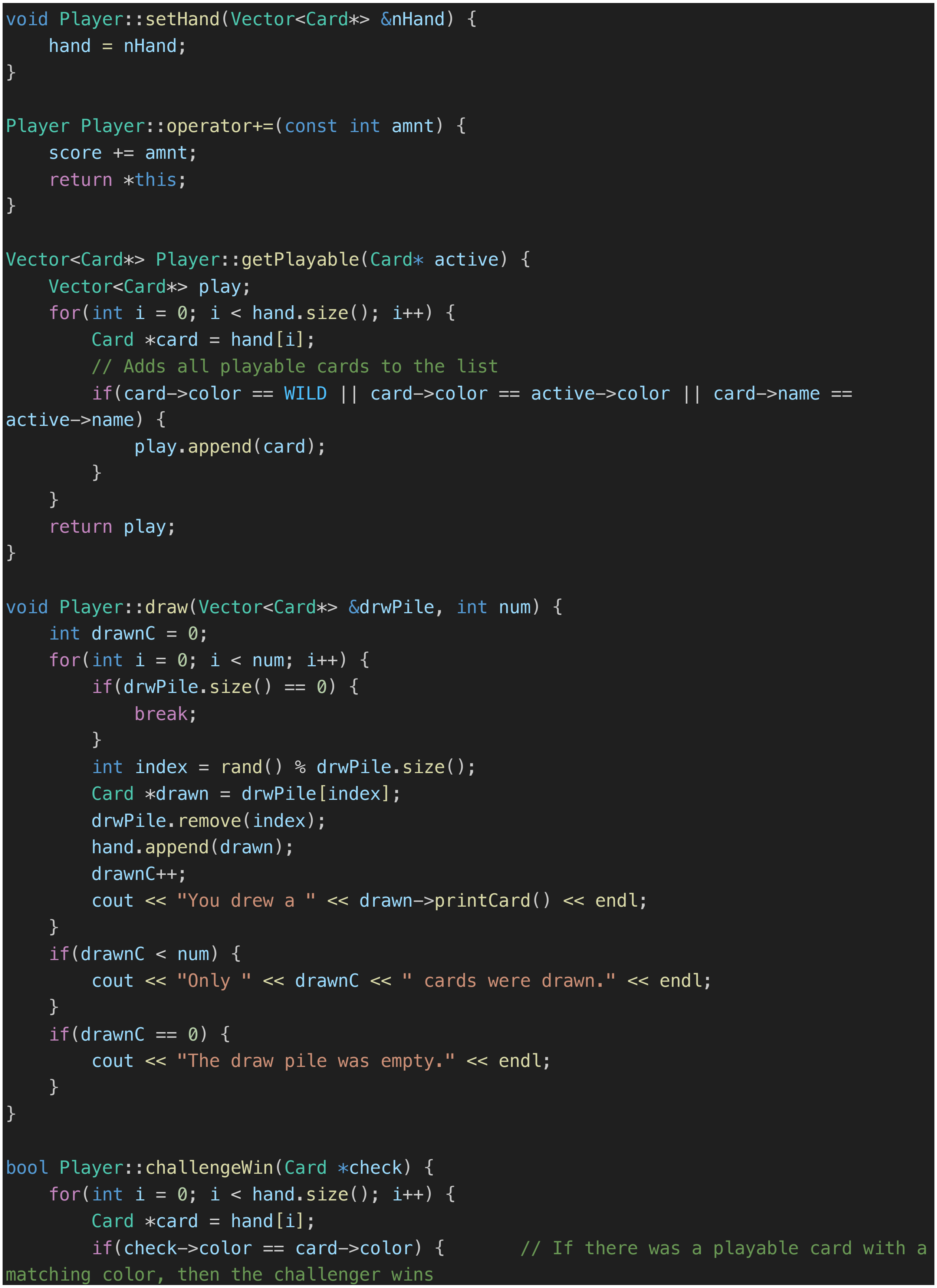
```
                    prev->draw(drwPile, 4);
                    bHandS += 4;
                }
            }else { // Splits the input into two for parsing of card color and type
                for(int i = 0; i < input.length(); i++) {
                    if(input[i] == ' ' && !next) {
                        next = true;
                    } else {
                        if(!next) {
                            wordF += input[i];
                        } else {
                            wordS += input[i];
                        }
                    }
                }
                int color, name;
                color = strToCol(wordF);
                name = strToNm(wordS);
                if(color != -1 && name != -1) { // Tries to find the card in the playable array. Removes it if found and returns it
                    for(int i = 0; i < valid.size(); i++) {
                        Card *card = valid[i];
                        if(card->color == color && card->name == name) {
                            choosen = true;
                            usrCard = card;
                            removeCard(hand, usrCard);
                            break;
                        }
                    }
                }
                if(!choosen) {
                    cout << "Can not play card" << endl;
                }
            }
        } else {    // Fail safe if there are no cards playable and the draw pile is empty
            cout << "You have no playable cards and the draw pile was empty. Your turn will be skipped." << endl;
            choosen = true;
        }
    }
    return usrCard;

}

int Player::strToCol(string input) {
    if(input == "red") {
        return RED;
```



```
    }
    if(input == "green") {
        return GREEN;
    }
    if(input == "yellow") {
        return YELLOW;
    }
    if(input == "blue") {
        return BLUE;
    }
    if(input == "wild") {
        return WILD;
    }
    return -1;
}

int Player::strToNm(string input) {
    if(input.length() == 1 && isdigit(input[0])) {
        return static_cast<int>(input[0] - '0');
    } else {
        if(input == "draw two") {
            return PLUST;
        }
        if(input == "draw four") {
            return PLUSF;
        }
        if(input == "reverse") {
            return REV;
        }
        if(input == "skip") {
            return SKIP;
        }
        if(input == "card") {
            return CARD;
        }
    }
    return -1;
}

Card *Player::drawCard(Vector<Card*> &drwPile) {
    if(drwPile.size() == 0) {
        return nullptr;
    }
    int index = rand() % drwPile.size();
    Card *card = drwPile[index];
    drwPile.remove(index);
    return card;
}
```



```
void Player::addCard(Card *card) {
    hand.append(card);
    sort(hand.begin(), hand.end());
}

void Player::removeCard(int &size, Card **hand, Card *card) {
    int index = 0;      // Finds the first instance of a card with a matching pointer
and removes it
    for(int i = 0; i < size; i++) {
        if(hand[i] == card) {
            index = i;
            break;
        }
    }
    for(int i = index+1; i < size; i++) {
        hand[i-1] = hand[i];
    }
    size--;
}

void Player::removeCard(Vector<Card*> &cards, Card *card) {
    for(int i = 0; i < cards.size(); i++) {
        if(cards[i] == card) {
            cards.remove(i);
            break;
        }
    }
}

void Player::printDeck(Card *active) {
    cout << "Cards in hand: ";
    for(int i = 0; i < hand.size(); i++) {
        cout << hand[i]->printCard() << "; ";     // Prints all the cards in the array
    }
    cout << endl;
    Vector<Card*> play;
    play = getPlayable(active);
    cout << "Playable cards in hand: ";
    for(int i = 0; i < play.size(); i++) {
        cout << play[i]->printCard() << "; ";     // Prints all playable cards
    }
    cout << endl;
}

Vector<Card*> Player::getHand() const {
    return hand;
}
```



```
void Player::setHand(Vector<Card*> &nHand) {
    hand = nHand;
}

Player Player::operator+=(const int amnt) {
    score += amnt;
    return *this;
}

Vector<Card*> Player::getPlayable(Card* active) {
    Vector<Card*> play;
    for(int i = 0; i < hand.size(); i++) {
        Card *card = hand[i];
        // Adds all playable cards to the list
        if(card->color == WILD || card->color == active->color || card->name ==
active->name) {
            play.append(card);
        }
    }
    return play;
}

void Player::draw(Vector<Card*> &drwPile, int num) {
    int drawnC = 0;
    for(int i = 0; i < num; i++) {
        if(drwPile.size() == 0) {
            break;
        }
        int index = rand() % drwPile.size();
        Card *drawn = drwPile[index];
        drwPile.remove(index);
        hand.append(drawn);
        drawnC++;
        cout << "You drew a " << drawn->printCard() << endl;
    }
    if(drawnC < num) {
        cout << "Only " << drawnC << " cards were drawn." << endl;
    }
    if(drawnC == 0) {
        cout << "The draw pile was empty." << endl;
    }
}

bool Player::challengeWin(Card *check) {
    for(int i = 0; i < hand.size(); i++) {
        Card *card = hand[i];
        if(check->color == card->color) {       // If there was a playable card with a
matching color, then the challenger wins
```

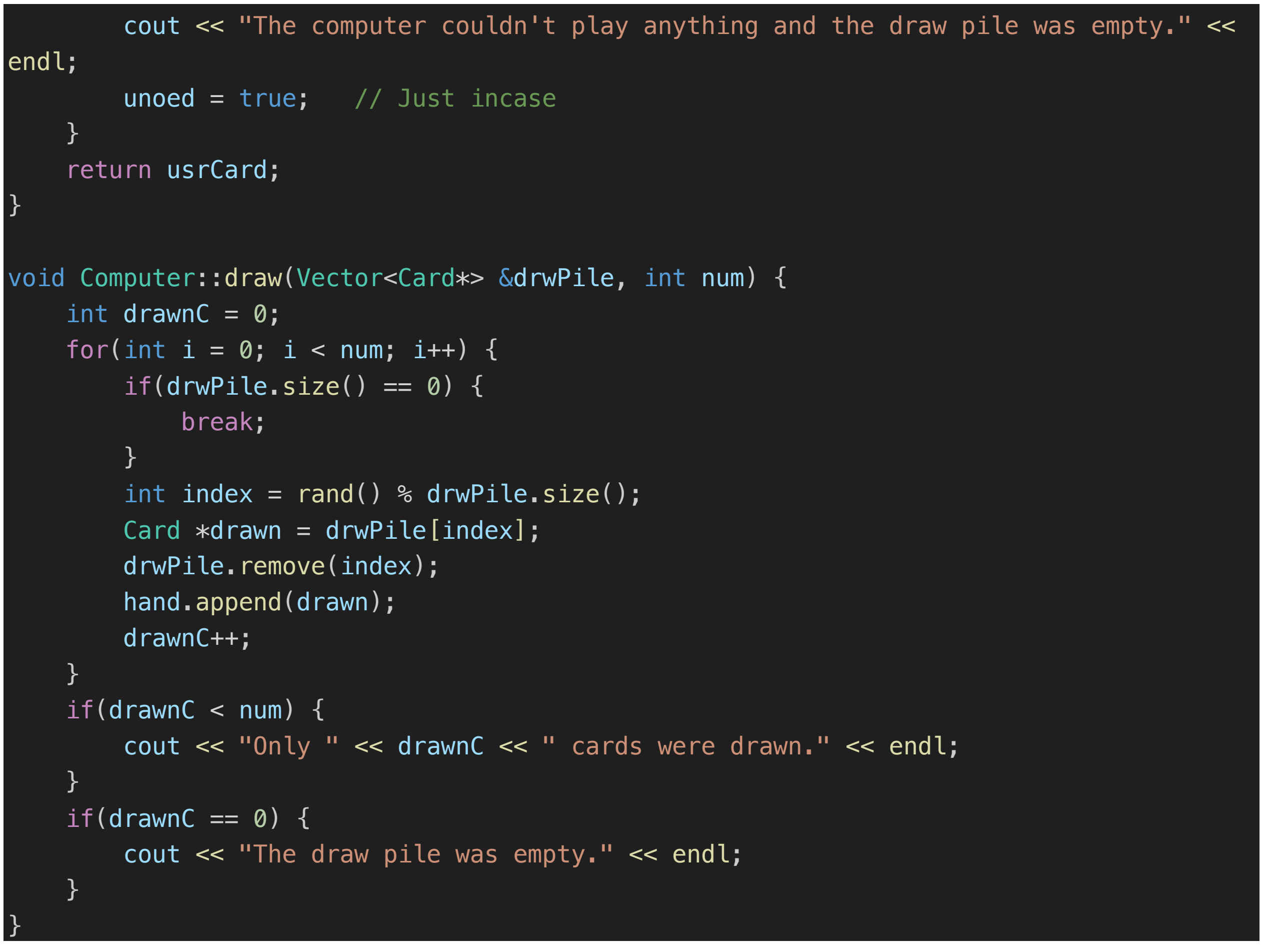
```
            return true;
        }
    }
    return false;
}
```

Computer.h

```
/*
 * File:   Computer.h
 * Author: Matthew Madrigal
 * Purpose:  Computer Structure
 */

#ifndef COMPUTER_H
#define COMPUTER_H

#include "Player.h"

class Computer : public Player{
private:
public:
    virtual Card *playCard(Card*, Vector<Card*> &, int, bool, bool &, bool &, Player
*prev);
    virtual void draw(Vector<Card*> &, int);
};

#endif /* COMPUTER_H */
```

Computer.cpp

```
/*
 * File:   Computer.cpp
 * Author: Matthew Madrigal
 * Created on December 21st, 2025, 10:10 am,
 * Purpose:  To define the functions in the Computer class
 */

//System Libraries
#include <iostream>
using namespace std;

//User Libraries
#include "Computer.h"
#include "Vector.h"

Card *Computer::playCard(Card* rest, Vector<Card*> &drwPile, int bHandS, bool uno,
bool &unoed, bool &quit, Player *prev) {
    if(uno) {   // If the player forgot to call uno, there is a 50% change the
computer catches it
        if(rand()%2 == 1) {
            cout << "The computer caught you with one card!" << endl;
            prev->draw(drwPile, 1);
        }
    }
    Card *usrCard = nullptr;
    Vector<Card*> valid = getPlayable(rest);
    if(valid.size() != 0 || drwPile.size() != 0) {  // If the computer can do
something
        if(valid.size() != 0) {                   // If the computer doesn't have to
draw, play a card
            int index = rand()%valid.size();
            usrCard = valid[index];
            removeCard(hand, usrCard);
            cout << "The computer played the card " << usrCard->printCard() << endl;
        } else {
            cout << "The computer drawed a card" << endl;
            Card *drawn = drawCard(drwPile);
            addCard(drawn);
        }
        if(hand.size() == 1 && rand()%2 == 1) { // There is a 50% chance the computer
remembers to call uno. I know this computer is really dumb
            cout << "The computer called uno!" << endl;
            unoed = true;
        }
    } else {    // If the computer can't play anything and for some reason the draw
pile is completly empty, skip their turn
```



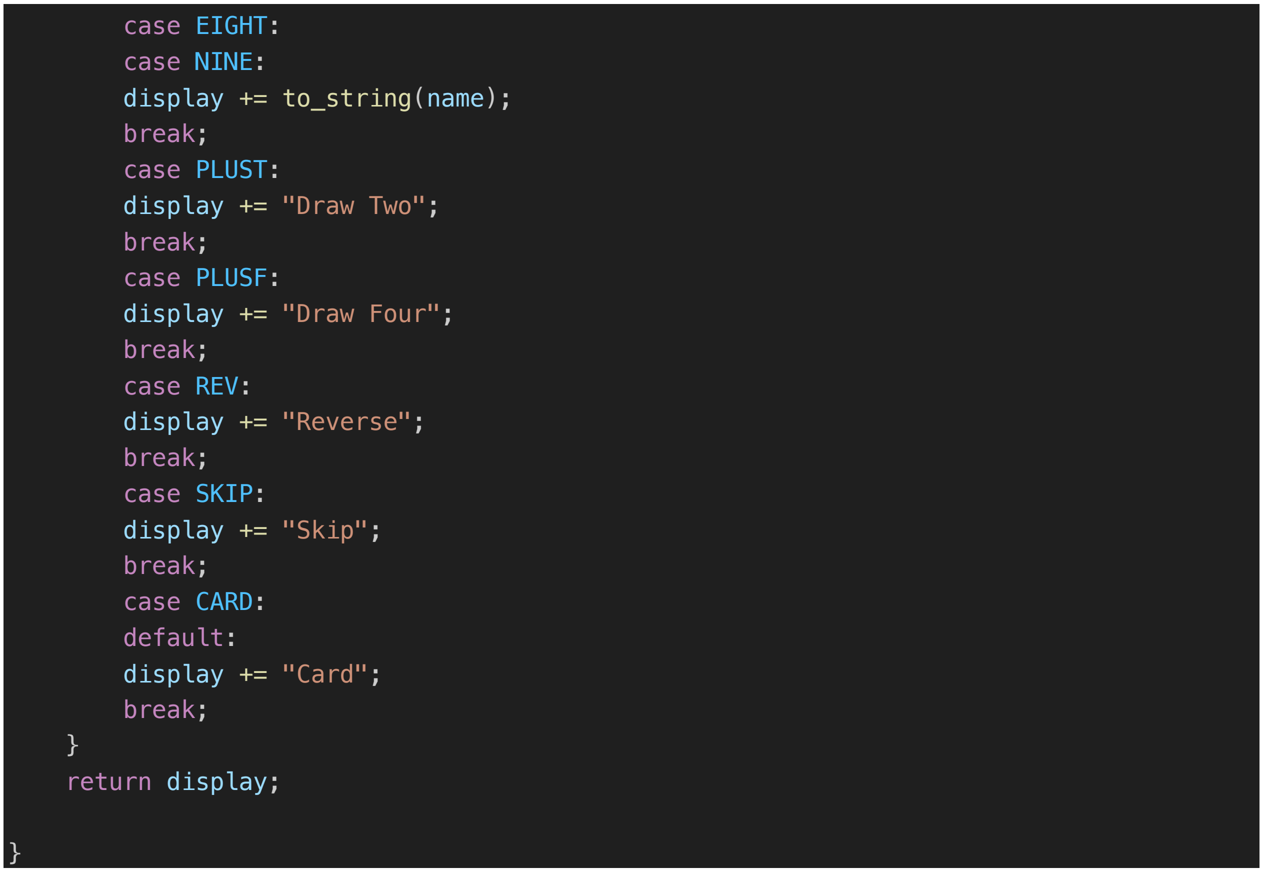
```
        cout << "The computer couldn't play anything and the draw pile was empty." <<
endl;
        unoed = true;   // Just incase
    }
    return usrCard;
}

void Computer::draw(Vector<Card*> &drwPile, int num) {
    int drawnC = 0;
    for(int i = 0; i < num; i++) {
        if(drwPile.size() == 0) {
            break;
        }
        int index = rand() % drwPile.size();
        Card *drawn = drwPile[index];
        drwPile.remove(index);
        hand.append(drawn);
        drawnC++;
    }
    if(drawnC < num) {
        cout << "Only " << drawnC << " cards were drawn." << endl;
    }
    if(drawnC == 0) {
        cout << "The draw pile was empty." << endl;
    }
}
```

Card.h

```cpp
/*
 * File:   Card.h
 * Author: Matthew Madrigal
 * Purpose:  Card Structure
 */

#ifndef CARD_H
#define CARD_H

#include <string>
using namespace std;

class Card{
private:
    enum colors {RED, GREEN, YELLOW, BLUE, WILD};
    enum types {ZERO, ONE, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, PLUST,
REV, SKIP, PLUSF, CARD};
public:
    int color;
    int name;
    string printCard();   // Prints a card's color and type
};

#endif /* CARD_H */
```

Card.cpp

```
/*
 * File:   Card.cpp
 * Author: Matthew Madrigal
 * Created on December 12th, 2025, 11:47 am,
 * Purpose:  To define the functions in the Card Class
 */

//System Libraries
#include <iostream>
using namespace std;

//User Libraries
#include "Card.h"

string Card::printCard() {
    string display;
    // Converts the color enum into a human readable string
    switch(color) {
        case RED:
        display += "Red ";
        break;
        case GREEN:
        display += "Green ";
        break;
        case YELLOW:
        display += "Yellow ";
        break;
        case BLUE:
        display += "Blue ";
        break;
        case WILD:
        display += "Wild ";
        break;
        default:
        break;
    }
    // Converts the type enum into a human readable string
    switch(name) {
        case ZERO:
        case ONE:
        case TWO:
        case THREE:
        case FOUR:
        case FIVE:
        case SIX:
        case SEVEN:
```

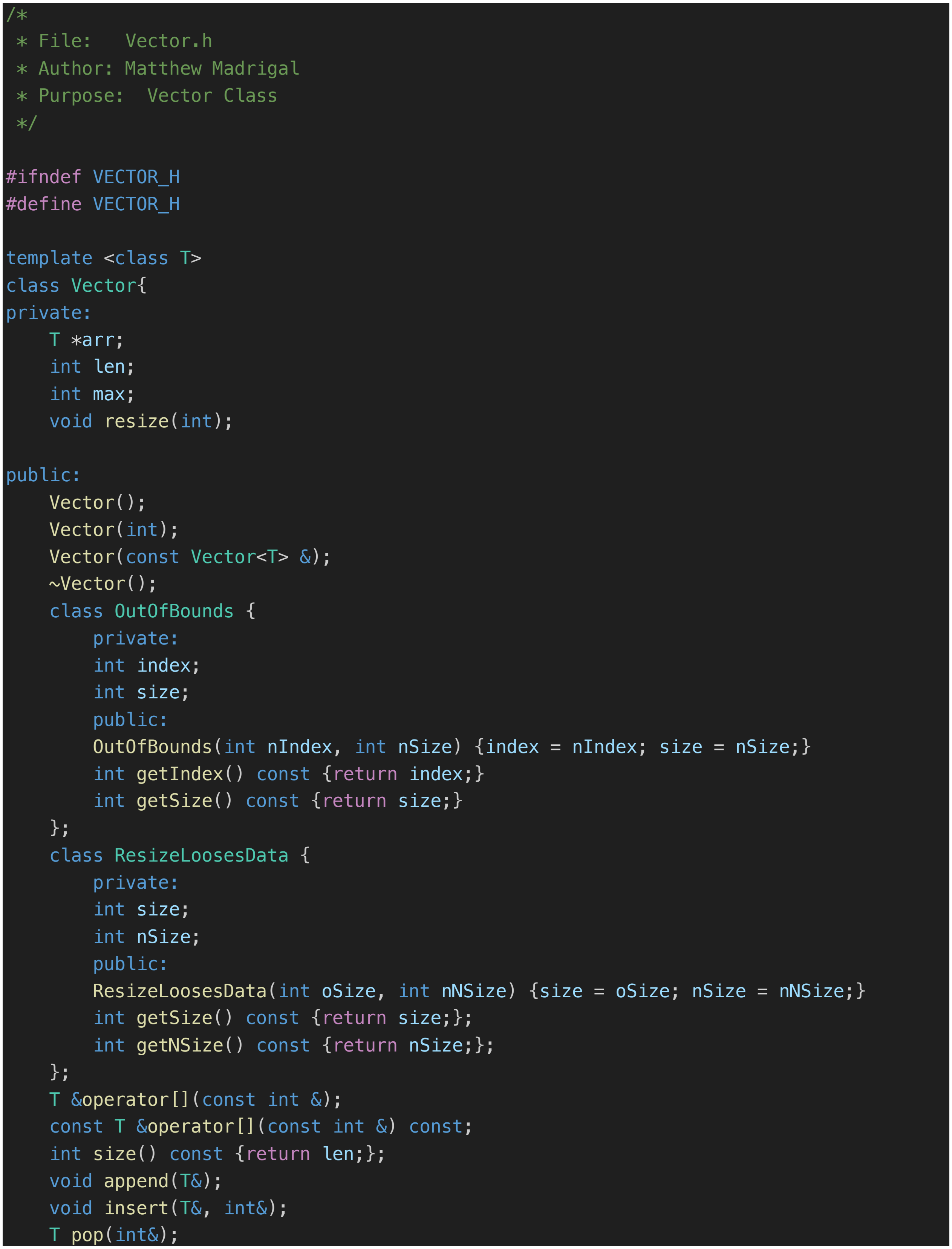


```
        case EIGHT:
        case NINE:
        display += to_string(name);
        break;
        case PLUST:
        display += "Draw Two";
        break;
        case PLUSF:
        display += "Draw Four";
        break;
        case REV:
        display += "Reverse";
        break;
        case SKIP:
        display += "Skip";
        break;
        case CARD:
        default:
        display += "Card";
        break;
    }
    return display;

}
```

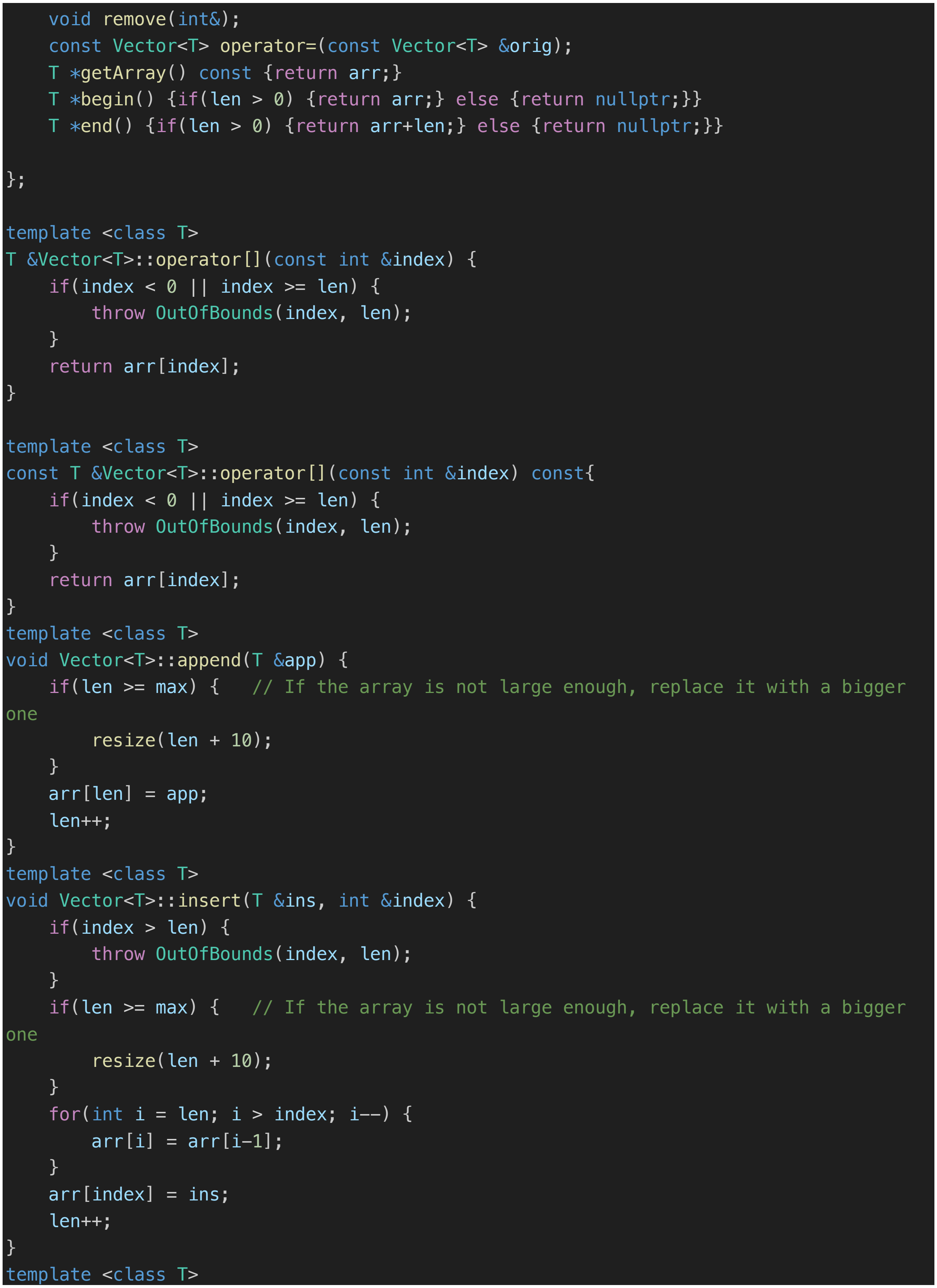
GameS.h

```
/*
 * File:   GameS.h
 * Author: Matthew Madrigal
 * Purpose:  This structure is used to save the game state to a file
 */

#ifndef GAMES_H
#define GAMES_H

#include "Card.h"

struct GameS{
    bool turn;      // True is for the player, false is for the computer
    int plScore;    // The score for the player
    int cpScore;    // The score for the computer
    int drwSize;    // Size of draw pile array
    int handS;      // The size of the array of the cards the player currently has
    int bHandS;     // The size of the array of the cards the computer currently has
    Card rest;      // The recent restriction on cards to play (useful for wild cards)
    Card lastPl;    // The last played card
    Card cards[54]; // The list of unique cards
    int draw[54];   // The amount of each unique card in the draw pile (index is the
same as the index of the unique card in cards)
    int play[54];   // The amount of each unique card in the player hand (index is the
same as the index of the unique card in cards
    int bot[54];    // The amount of each unique card in the computer hand (index is
the same as the index of the unique card in cards
};

#endif /* Game_S_H */
```

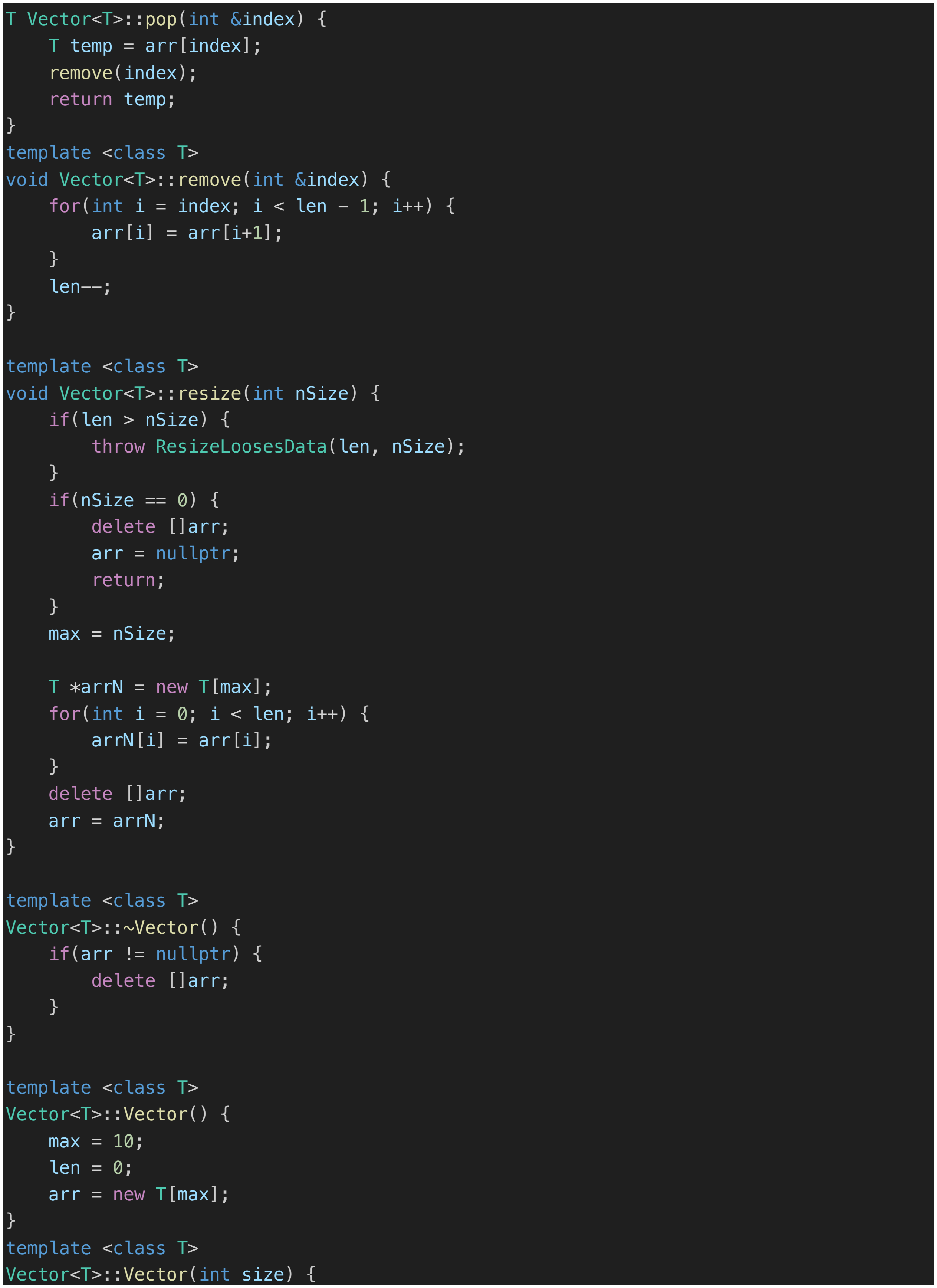
Vector.h



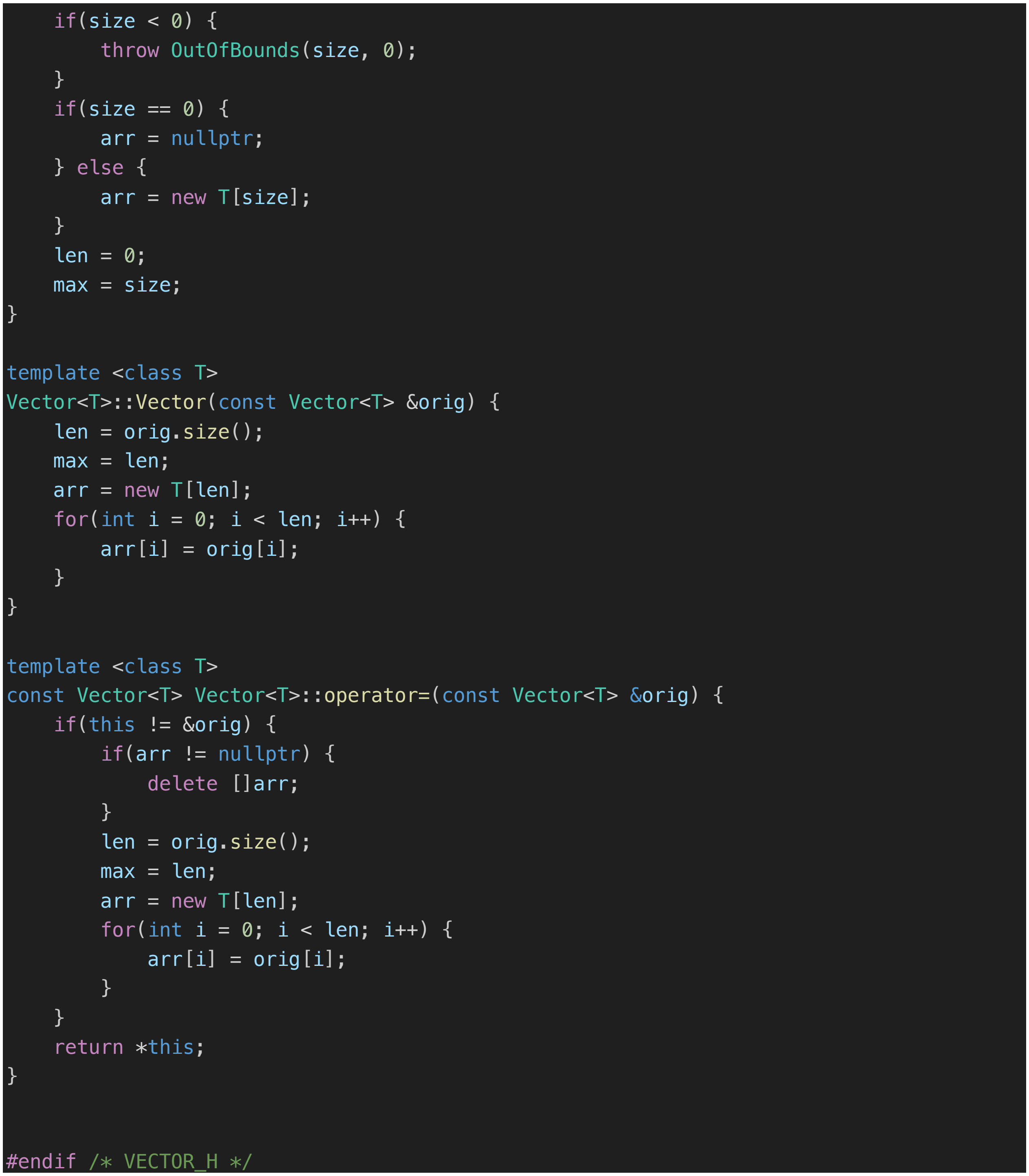
```cpp
/*
 * File:   Vector.h
 * Author: Matthew Madrigal
 * Purpose:  Vector Class
 */

#ifndef VECTOR_H
#define VECTOR_H

template <class T>
class Vector{
private:
    T *arr;
    int len;
    int max;
    void resize(int);

public:
    Vector();
    Vector(int);
    Vector(const Vector<T> &);
    ~Vector();
    class OutOfBounds {
        private:
        int index;
        int size;
        public:
        OutOfBounds(int nIndex, int nSize) {index = nIndex; size = nSize;}
        int getIndex() const {return index;}
        int getSize() const {return size;}
    };
    class ResizeLoosesData {
        private:
        int size;
        int nSize;
        public:
        ResizeLoosesData(int oSize, int nNSize) {size = oSize; nSize = nNSize;}
        int getSize() const {return size;};
        int getNSize() const {return nSize;};
    };
    T &operator[](const int &);
    const T &operator[](const int &) const;
    int size() const {return len;};
    void append(T&);
    void insert(T&, int&);
    T pop(int&);
```



```
    void remove(int&);
    const Vector<T> operator=(const Vector<T> &orig);
    T *getArray() const {return arr;}
    T *begin() {if(len > 0) {return arr;} else {return nullptr;}}
    T *end() {if(len > 0) {return arr+len;} else {return nullptr;}}

};

template <class T>
T &Vector<T>::operator[](const int &index) {
    if(index < 0 || index >= len) {
        throw OutOfBounds(index, len);
    }
    return arr[index];
}

template <class T>
const T &Vector<T>::operator[](const int &index) const{
    if(index < 0 || index >= len) {
        throw OutOfBounds(index, len);
    }
    return arr[index];
}
template <class T>
void Vector<T>::append(T &app) {
    if(len >= max) {   // If the array is not large enough, replace it with a bigger one
        resize(len + 10);
    }
    arr[len] = app;
    len++;
}
template <class T>
void Vector<T>::insert(T &ins, int &index) {
    if(index > len) {
        throw OutOfBounds(index, len);
    }
    if(len >= max) {   // If the array is not large enough, replace it with a bigger one
        resize(len + 10);
    }
    for(int i = len; i > index; i--) {
        arr[i] = arr[i-1];
    }
    arr[index] = ins;
    len++;
}
template <class T>
```



```
T Vector<T>::pop(int &index) {
    T temp = arr[index];
    remove(index);
    return temp;
}
template <class T>
void Vector<T>::remove(int &index) {
    for(int i = index; i < len - 1; i++) {
        arr[i] = arr[i+1];
    }
    len--;
}

template <class T>
void Vector<T>::resize(int nSize) {
    if(len > nSize) {
        throw ResizeLoosesData(len, nSize);
    }
    if(nSize == 0) {
        delete []arr;
        arr = nullptr;
        return;
    }
    max = nSize;

    T *arrN = new T[max];
    for(int i = 0; i < len; i++) {
        arrN[i] = arr[i];
    }
    delete []arr;
    arr = arrN;
}

template <class T>
Vector<T>::~Vector() {
    if(arr != nullptr) {
        delete []arr;
    }
}

template <class T>
Vector<T>::Vector() {
    max = 10;
    len = 0;
    arr = new T[max];
}
template <class T>
Vector<T>::Vector(int size) {
```



```
    if(size < 0) {
        throw OutOfBounds(size, 0);
    }
    if(size == 0) {
        arr = nullptr;
    } else {
        arr = new T[size];
    }
    len = 0;
    max = size;
}

template <class T>
Vector<T>::Vector(const Vector<T> &orig) {
    len = orig.size();
    max = len;
    arr = new T[len];
    for(int i = 0; i < len; i++) {
        arr[i] = orig[i];
    }
}

template <class T>
const Vector<T> Vector<T>::operator=(const Vector<T> &orig) {
    if(this != &orig) {
        if(arr != nullptr) {
            delete []arr;
        }
        len = orig.size();
        max = len;
        arr = new T[len];
        for(int i = 0; i < len; i++) {
            arr[i] = orig[i];
        }
    }
    return *this;
}


#endif /* VECTOR_H */
```

Original Proposal

Introduction

The primary goal of this project was to recreate a board game or card game which has been around for more than 10 years in C++. For this project, I decided to recreate Uno in C++. There are already many versions of uno out there already with their own graphics and fancy animations. However, this version of Uno is useful because its free and it runs in text only which could be useful if using a computer with very few resources. It can also be helpful to those sensitive to the flashy animations other versions include.

The fundamental rules of uno are that a card is placed, you and an opponent have cards, and whoever goes next either has to play a card with the same color, number, or draw from the draw pile. There are exceptions to this rule, such as if the player has a wild card, then it can be placed no matter the previous card. There are also special cards that may force the opponent to draw cards or even skip the opponents turn. These are called skip cards, reverse cards, draw 2 cards, and draw 4 cards. The skip card and reverse card skip the turn of the opponent in a two-player game. The draw 2 and wild draw 4 card skip the opponent's turn and forces them to draw 2 or 4 cards respectively. When the game starts, players have to start with 7 random cards. The card to start it all off is drawn from the draw pile and can't be a wild card. During gameplay, a player is not supposed to play a draw 4 wild card unless they don't have a matching color. They are allowed to do it but that opens them up to being challenged by the opponent where they can see their hand and if the player has a matching color, they instead are forced to draw 4. When a player draws a card, they can only draw once. When they draw the card, they can play it if its playable during the same turn where they draw.

Development Summary

There are 968 Lines in the main.cpp file alone. 32 of the lines are blank spaces and 32 of those are comments only. That leaves 904 lines that have some code on them. This project makes use of structs, pointers, dynamically allocated memory, random reads and writes from files, reading and writing to the same file, reading from more than 1 file, strings, functions, and other things. I found this project very challenging to do due to how many different situations that can occur with uno. It took me at around 4 full days (about 96 hours) to put together. The scariest part of the project was when I decided to only have 54 instances of cards and to duplicate the pointers and move those around instead of creating an instance of each. There were many opportunities for me to accidently forget to free a structure, array, or accidently double free something. However, through careful planning, I was able to implement some methods that would reduce the chances of that such as keeping an array of all the original cards to make freeing them easier. Managing all the curly brackets was also a challenge with how many were nested at some parts of the code. A concern I had when planning this project was how I would create the enum and make it accessible inside the struct. I then remembered that you can assign enums to ints without any form of conversion and decided to use ints in the structures. A major help from the textbook was the section that covered the random function. Without it, it would have taken a lot of work to write something that is even remotely close to as random as the built in one for c++.

Description

Structures and layout

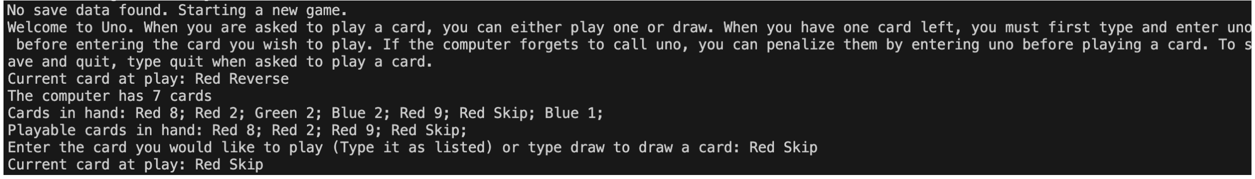
I programmed this game by using a main structure which stored the current state of the game and splitting up all the tasks into smaller functions to make reusing code and debugging easier. The main structure is first created in the main function and is passed to almost every function from that point onward. The structure is called game. It contains the cards in the draw pile, the cards in both opponents' hands, last played card, and the size of the arrays. I also have a structure for a card which contains the color (I consider wild a color in this program) and the type which can be either a number or a special card such as draw 2. Finally, I have a structure for saving the game which is a more basic version of the main game structure. Instead of having dynamic arrays, it has static arrays of the unique cards, and how many are in each dynamic array from the main game structure. This makes restoring the program state after a save and load much easier

Program start

When the program is ran, the game first checks if a save file exists. If it does, it tries to read from it. If it notices the save file is of a game that already ended, it will ignore it and create a new game. If its not, it will recreate the game with a helper function and then resume it. If there was no save file it would then continue to create a new game. Finally, before running the main loop, the program explains what the program is and some important things that might not be obvious at first.

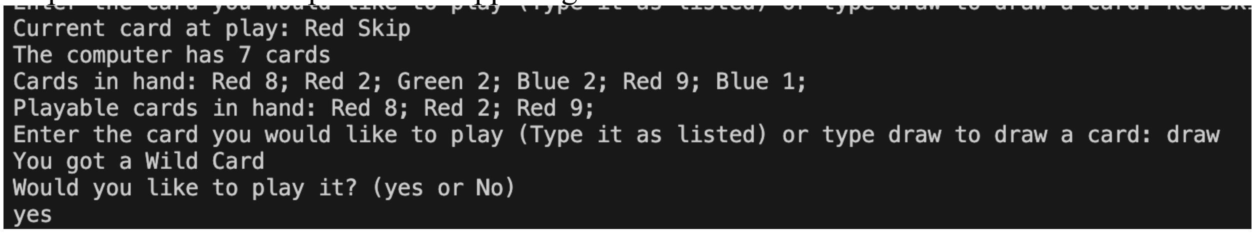
Game Loop

The first thing the program then asks from the user is what card they want to play. The cards the player has and the cards the player can play, and the number of cards the computer has are printed for the player to see before this. The player is then expected to type out one of the playable cards, the word draw, quit, or uno. The program then performs the actions required for the card played and then continues. Below is an example of the player playing a red skip card.



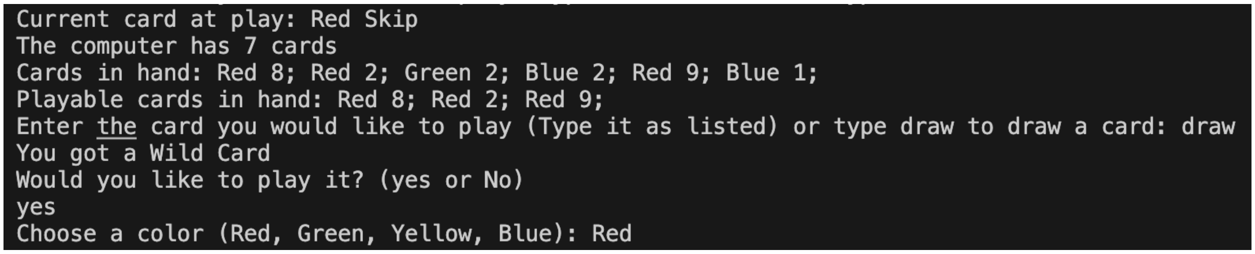
```
No save data found. Starting a new game.
Welcome to Uno. When you are asked to play a card, you can either play one or draw. When you have one card left, you must first type and enter uno
 before entering the card you wish to play. If the computer forgets to call uno, you can penalize them by entering uno before playing a card. To s
ave and quit, type quit when asked to play a card.
Current card at play: Red Reverse
The computer has 7 cards
Cards in hand: Red 8; Red 2; Green 2; Blue 2; Red 9; Red Skip; Blue 1;
Playable cards in hand: Red 8; Red 2; Red 9; Red Skip;
Enter the card you would like to play (Type it as listed) or type draw to draw a card: Red Skip
Current card at play: Red Skip
```

Normally, the turn will alternate every time a card is played, however in some situations, such as the example input/output, this is not the case and the turn doesn't change. Other cards that can cause this include a skip, reverse, and draw 4 card. In the situation that the player decides to draw, the game gives them a card from the draw pile which if playable they can choose to play or keep. Below is an example of this happening.



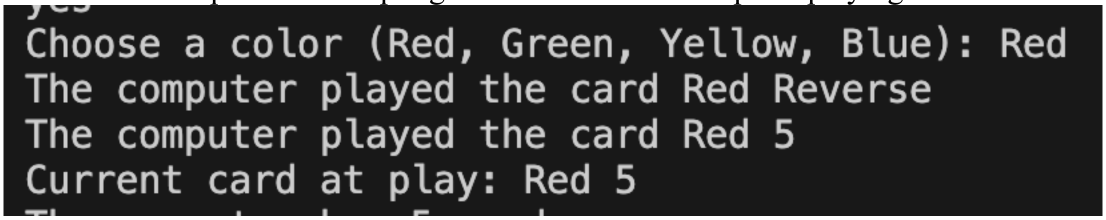
```
Enter the card you would like to play (Type it as listed) or type draw to draw a card: Red Sk
Current card at play: Red Skip
The computer has 7 cards
Cards in hand: Red 8; Red 2; Green 2; Blue 2; Red 9; Blue 1;
Playable cards in hand: Red 8; Red 2; Red 9;
Enter the card you would like to play (Type it as listed) or type draw to draw a card: draw
You got a Wild Card
Would you like to play it? (yes or No)
yes
```

When the player plays any wild card, they will be asked what color they want. This is shown in the same example input which is extended below.



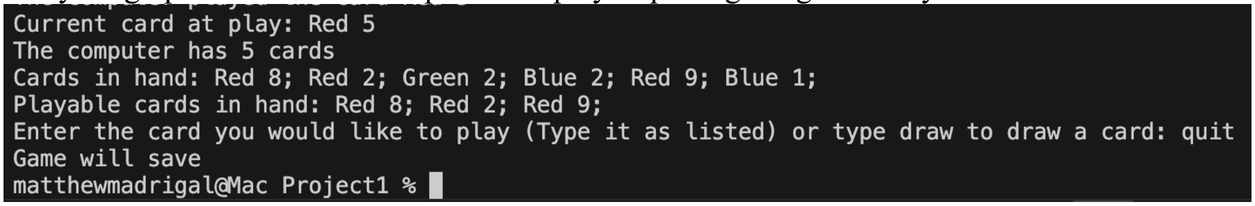
```
Current card at play: Red Skip
The computer has 7 cards
Cards in hand: Red 8; Red 2; Green 2; Blue 2; Red 9; Blue 1;
Playable cards in hand: Red 8; Red 2; Red 9;
Enter the card you would like to play (Type it as listed) or type draw to draw a card: draw
You got a Wild Card
Would you like to play it? (yes or No)
yes
Choose a color (Red, Green, Yellow, Blue): Red
```

After this, the computer will get to play with most of the same things happening. The primary difference is that the choices are made by a random number generator instead of the player and some of the stuff shown to the player such as the hand isn't shown when the computer plays. Here is an example of the output generated from the computer playing two cards in a row.



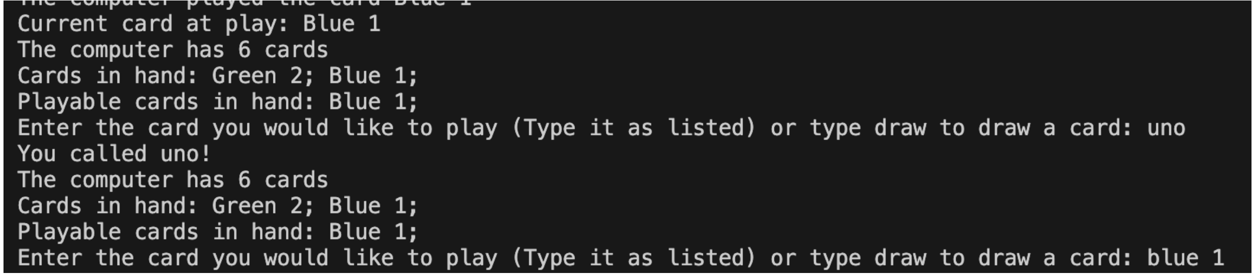
```
yes
Choose a color (Red, Green, Yellow, Blue): Red
The computer played the card Red Reverse
The computer played the card Red 5
Current card at play: Red 5
```

While the player continues to play, the game will keep track of the number of cards they have to handle the uno feature and end the round or game when needed. When the round ends, the game calculates the score of the winning player by examining the value of the cards the losing player had. Once a player gets over 500 points, the game ends and the program saves the game before cleaning up everything and exiting. The player is also able to quit the program early by typing quit when asked to play a card which will still result in the game saving before cleaning everything up. Below is an example of the player quitting the game early.



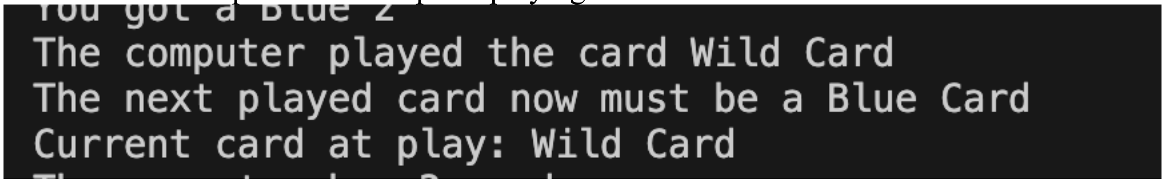
```
Current card at play: Red 5
The computer has 5 cards
Cards in hand: Red 8; Red 2; Green 2; Blue 2; Red 9; Blue 1;
Playable cards in hand: Red 8; Red 2; Red 9;
Enter the card you would like to play (Type it as listed) or type draw to draw a card: quit
Game will save
matthewmadrigal@Mac Project1 % 
```

Below is an example of the player calling uno before having one card left



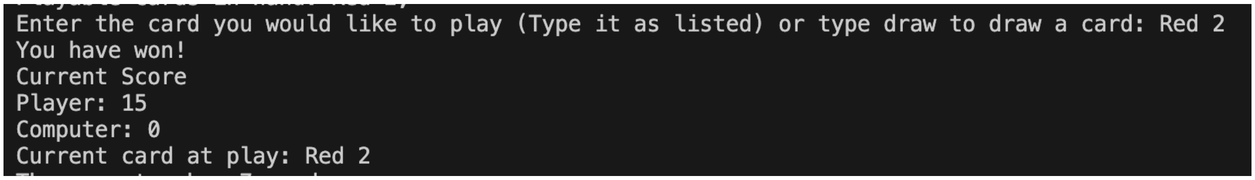
```
The computer played the card Blue 1
Current card at play: Blue 1
The computer has 6 cards
Cards in hand: Green 2; Blue 1;
Playable cards in hand: Blue 1;
Enter the card you would like to play (Type it as listed) or type draw to draw a card: uno
You called uno!
The computer has 6 cards
Cards in hand: Green 2; Blue 1;
Playable cards in hand: Blue 1;
Enter the card you would like to play (Type it as listed) or type draw to draw a card: blue 1
```

Below is an example of the computer playing a wild card



```
You got a Blue 2
The computer played the card Wild Card
The next played card now must be a Blue Card
Current card at play: Wild Card
```

Below is an example of the player winning and the game starting a new round

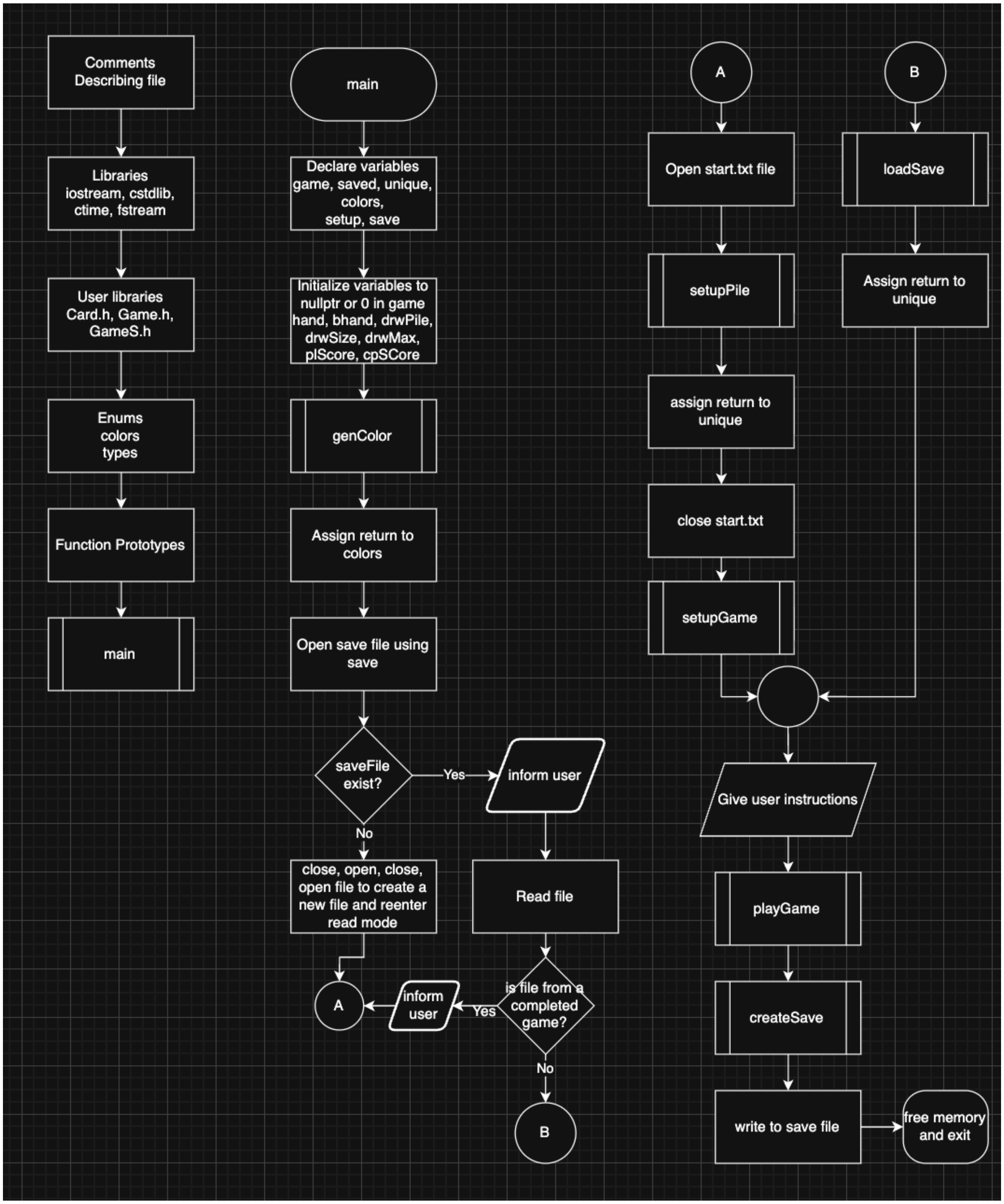


```
Enter the card you would like to play (Type it as listed) or type draw to draw a card: Red 2
You have won!
Current Score
Player: 15
Computer: 0
Current card at play: Red 2
```

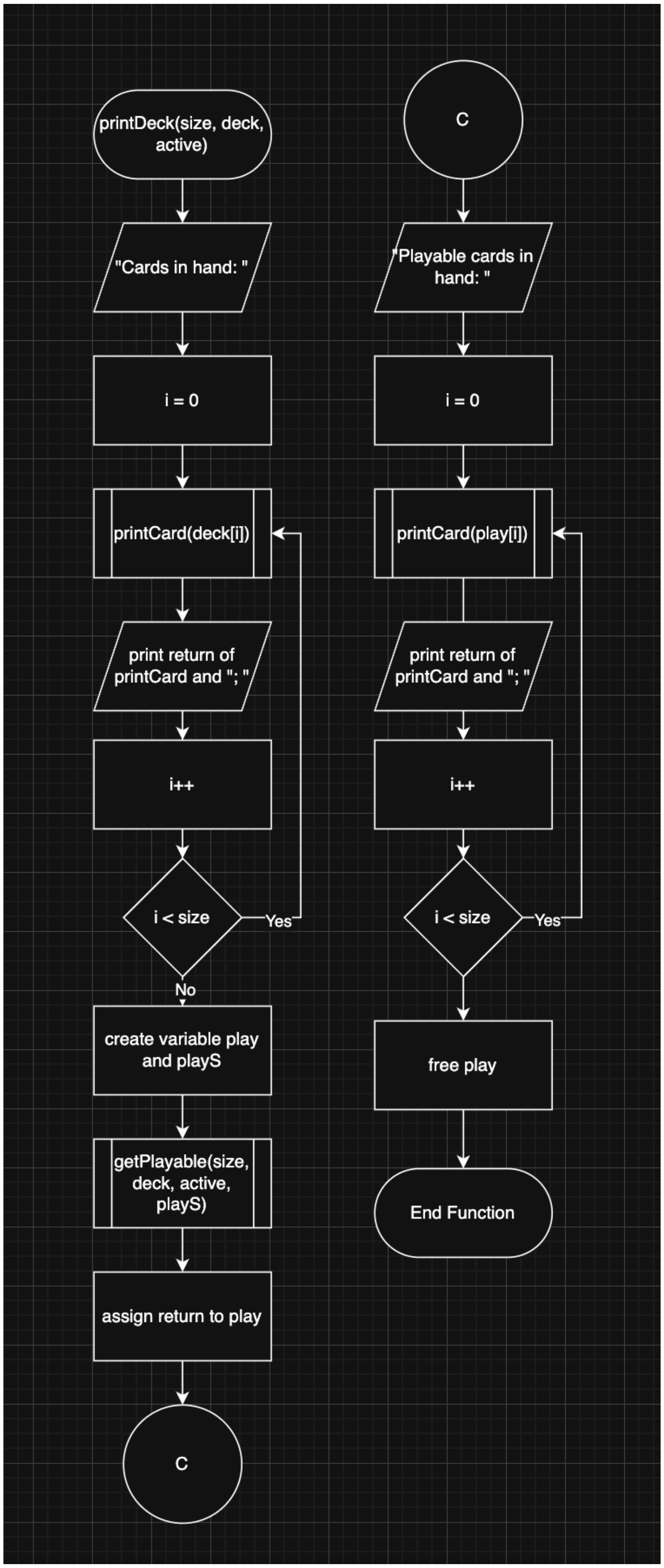
All of these primarily managed by the playGame function. To perform all of these actions, many smaller helper functions are called when needed. These functions include printDeck, printCard, getPlayable, challengeWin, calcPoints, strToCol, strToNm removeCard, drawCard, genColor, copy, addCard, setupGame, playCard, playCom, setupPile, loadSave, and createSave.

Flowcharts

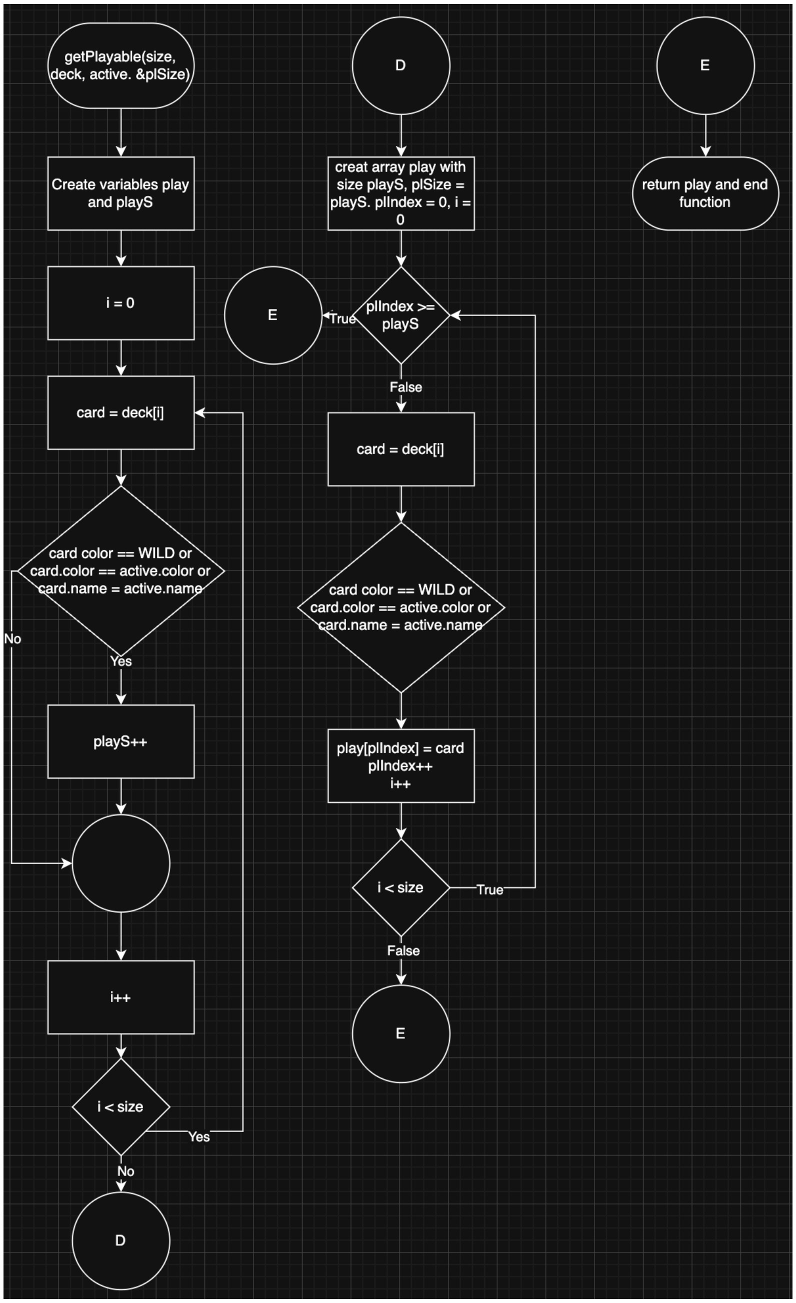
I have created 3 flow charts for different functions.
main:


Comments Describing file
Libraries iostream, cstdlib, ctime, fstream
User libraries Card.h, Game.h, GameS.h
Enums colors types
Function Prototypes
main
main
Declare variables game, saved, unique, colors, setup, save
Initialize variables to nullptr or 0 in game hand, bhand, drwPile, drwSize, drwMax, plScore, cpSCore
genColor
Assign return to colors
Open save file using save
saveFile exist?
Yes
inform user
Read file
No
close, open, close, open file to create a new file and reenter read mode
is file from a completed game?
Yes
inform user
A
No
B
A
Open start.txt file
setupPile
assign return to unique
close start.txt
setupGame
B
loadSave
Assign return to unique
Give user instructions
playGame
createSave
write to save file
free memory and exit

printDeck:


printDeck(size, deck, active)
"Cards in hand: "
i = 0
printCard(deck[i])
print return of printCard and "; "
i++
i < size
Yes
No
create variable play and playS
getPlayable(size, deck, active, playS)
assign return to play
C
C
"Playable cards in hand: "
i = 0
printCard(play[i])
print return of printCard and "; "
i++
i < size
Yes
free play
End Function

getPlayable:


getPlayable(size, deck, active. &plSize)
Create variables play and playS
i = 0
card = deck[i]
card color == WILD or card.color == active.color or card.name = active.name
No
Yes
playS++
i++
i < size
Yes
No
D
D
creat array play with size playS, plSize = playS. plIndex = 0, i = 0
plIndex >= playS
True
E
False
card = deck[i]
card color == WILD or card.color == active.color or card.name = active.name
play[plIndex] = card
plIndex++
i++
i < size
True
False
E
E
return play and end function

Pseudocode

Below is the pseudocode for the main function

```
Declare variables
Initialize all game variables to nullptr or 0
Generate color arrays for wild card
Open save file
If saveFile has a game, read it and set all necessary properties
Otherwise setup a new game
Display instructions
```

```
Play game
 Save game to file
Free allocated memory and exit
```

Below is the pseudocode for the remove card function

```
Search for the first element in the array with the same pointer
Save the index of the element in index
Starting at index+1 with I = index+1, set the element at i-1 to the element at i.
I++
Decrement size by 1
```

Important variables

This was already partially covered but I would like to focus on the ones defined in the main function and list their line number. The most crucial variables in the program include the game variable on line 54 which is a Game struct from Game.h, the unique variable on line 55 which is an array of pointers to Card structures from Card.h, and the colors variable on line 56 which is an array of pointers to Card structures from Card.h. These variables are necessary to provide the basic features of Uno without considering other features such as saving or challenges. They are also passed around a lot to multiple different functions.

Concepts

I have learned a few concepts so far in the class. The list of concepts I have used are below, along side where they are found

Pointer Variables

These are used everywhere in the program. The main struct Game has a pointer to a dynamically allocated array which stores pointers to Cards. This is found in the Game.h file on line 20. Line 16 also has a variable which is just a pointer. Most of the functions I use has arguments with pointers too.

Arrays/pointers

I use arrays everywhere in my program. A easy to find one is also in Game.h. The line 20 I mentioned before is an a pointer to an array and line 23 is another one.

Function Parameters

All of my functions have parameters. In main.cpp on line 127 I have a function defined which has parameters set. On 135 in the same function I call a function passing in arguments to its parameters.

Memory Allocation

Main.cpp line 514

Return Parameters Pointers

Main.cpp line 270

C-Strings

Main.cpp line 303-304

Strings

Main.cpp line 280-281

Structured data Arrays

Main.cpp line 202 or GameS.h line 221

Nested structures

GameS.h line 19

Structure Function Arguments

Main.cpp line 144

Structure Function Return

Main.cpp line 472

Pointers in structures

Game.h line 17

Enumeration

Main.cpp line 20-21

File Formatting

Main.cpp line 76

Files Function Parameters

Main.cpp line 847

File Member Functions

Main.cpp line 859

Multiple files

The main function has two files open at once when setting up a new game at line 92 in main.cpp

Binary Files

Main.cpp line 73

Records with structures

Main.cpp line 108

Random Access Files

Main.cpp lines 889-936

Input output simultaneous

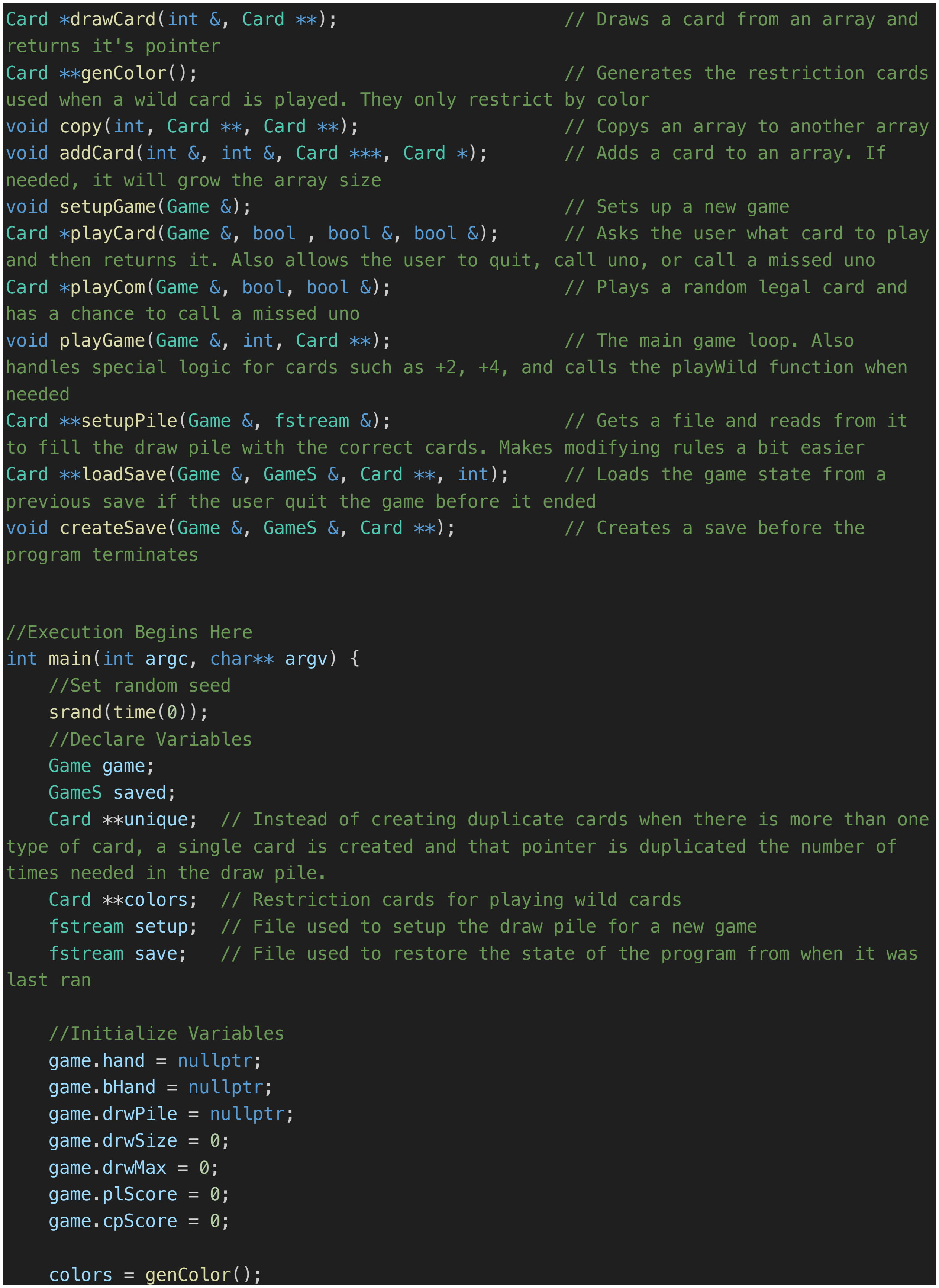
The main function opens a file for read and write and in certain cases and read to it and write to it within the same session without closing it. Main.cpp lines 49-125. File is opened line 73.

References

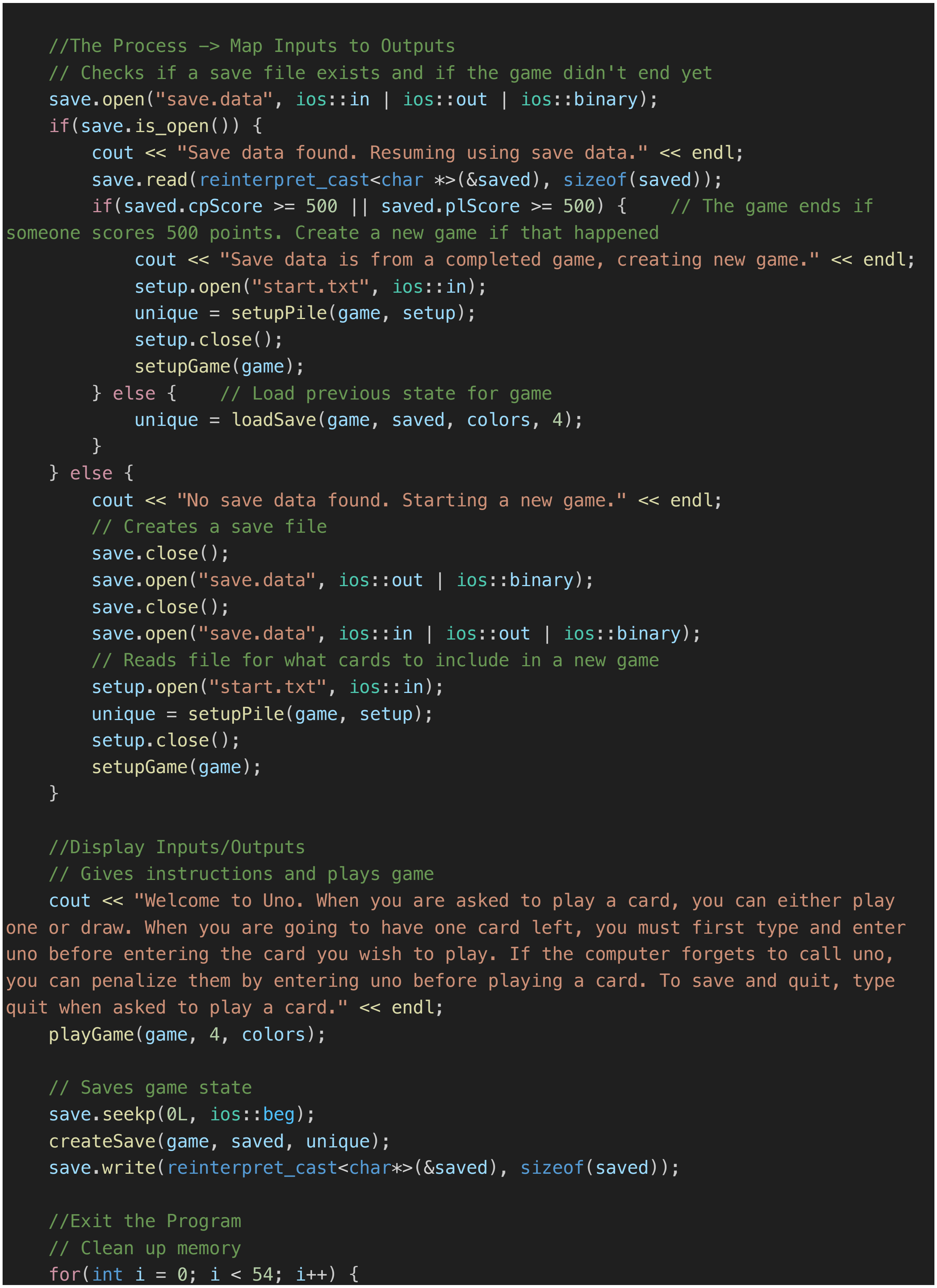
This program barrowed a lot from many of my previous projects both in classes and personal ones. One of these was the idea of working with purely pointers which was inspired by a project from my CIS-5 class. The input validation exists largely due to me experimenting with it during my midterm for CIS-17A even if it’s not as good as the input validations in the midterm. Nothing was directly copied from my previous projects but I did copy small snippets of code from the class textbook including the demonstration of using rands with ctime and the equation for generating random numbers.

Main.cpp

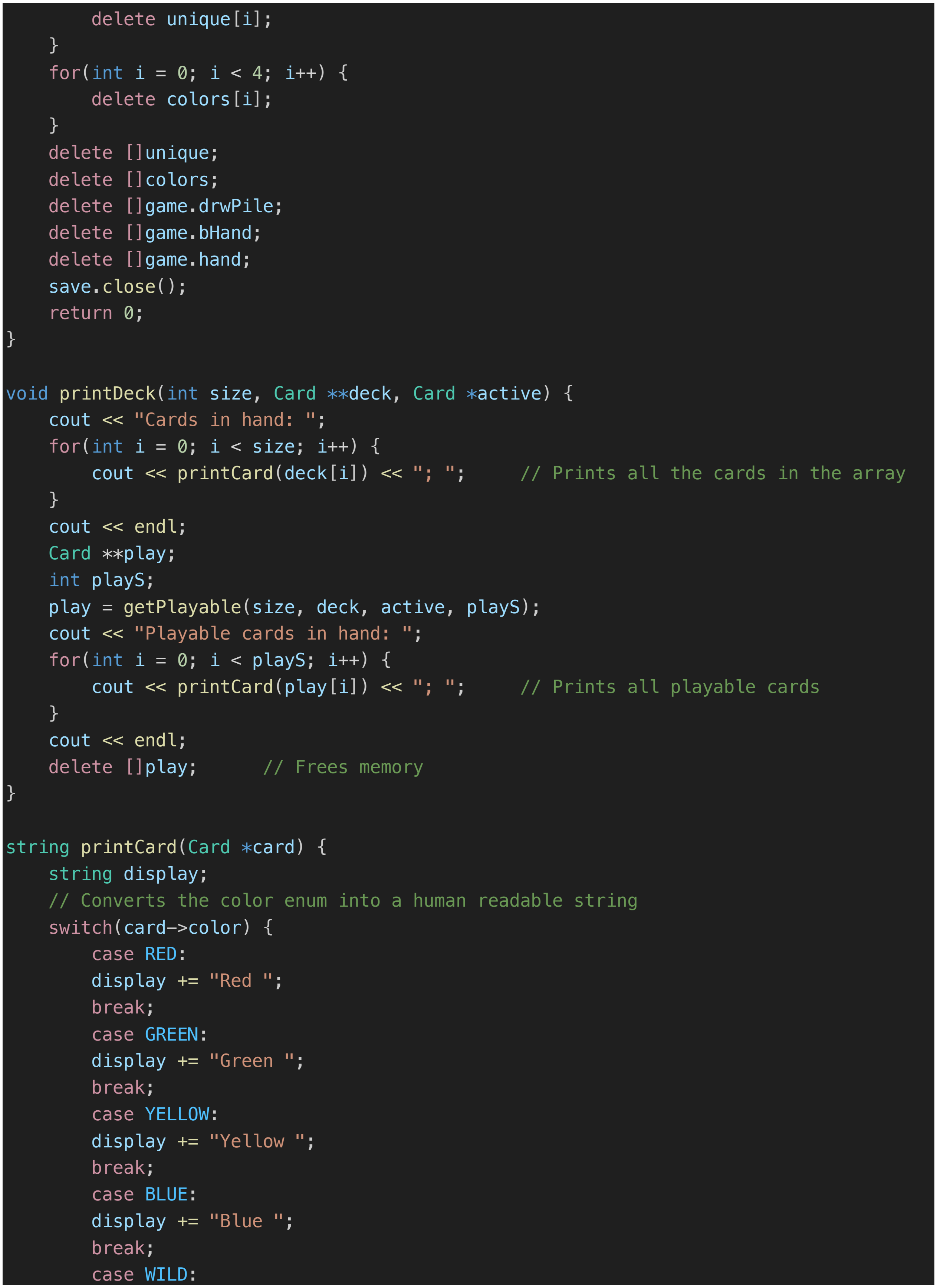
```
/*
 * File:   main.cpp
 * Author: Matthew Madrigal
 * Created on October 31st, 2025, 3:52 pm,
 * Purpose:  To create an Uno game in C++ using what has been learned so far. These are the rules used for this game: https://service.mattel.com/instruction_sheets/42001pr.pdf
 *           The blank cards are not included since they are for custom rules or replacing lost cards.
 */

//System Libraries
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <fstream>
using namespace std;

//User Libraries
#include "Card.h"
#include "Game.h"
#include "GameS.h"
enum colors {RED, GREEN, YELLOW, BLUE, WILD};
enum types {ZERO, ONE, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, PLUST, REV, SKIP, PLUSF, CARD};

//Global Constants - Math/Science/Conversions only

//Function Prototypes
void printDeck(int, Card **, Card *);                // Prints what cards are in the array and what cards can be played.
string printCard(Card *);                            // Prints a card's color and type
Card **getPlayable(int, Card **, Card *, int &);     // Creates a new dynamically allocated array with the cards that can be played given a restriction card
bool challengeWin(Card *, Card **, int);             // Checks if a hand has a card with a matching color to the restriction. Needed for challenges
unsigned int calcPoints(Card **, int);               // Calculates the number of points a person earns for winning
int strToCol(string);                                // Converts a string to a color enum
int strToNm(string);                                 // Converts a string to a type enum
void removeCard(int &, Card **, Card *);             // Removes a card given its pointer from an array and decreases its size by one
Card *playWild(int, Card **);                        // Asks the user what color to choose for the wild card
```



```
Card *drawCard(int &, Card **);                     // Draws a card from an array and
returns it's pointer
Card **genColor();                                  // Generates the restriction cards
used when a wild card is played. They only restrict by color
void copy(int, Card **, Card **);                   // Copys an array to another array
void addCard(int &, int &, Card ***, Card *);       // Adds a card to an array. If
needed, it will grow the array size
void setupGame(Game &);                             // Sets up a new game
Card *playCard(Game &, bool , bool &, bool &);      // Asks the user what card to play
and then returns it. Also allows the user to quit, call uno, or call a missed uno
Card *playCom(Game &, bool, bool &);                // Plays a random legal card and
has a chance to call a missed uno
void playGame(Game &, int, Card **);                // The main game loop. Also
handles special logic for cards such as +2, +4, and calls the playWild function when
needed
Card **setupPile(Game &, fstream &);                // Gets a file and reads from it
to fill the draw pile with the correct cards. Makes modifying rules a bit easier
Card **loadSave(Game &, GameS &, Card **, int);     // Loads the game state from a
previous save if the user quit the game before it ended
void createSave(Game &, GameS &, Card **);          // Creates a save before the
program terminates


//Execution Begins Here
int main(int argc, char** argv) {
    //Set random seed
    srand(time(0));
    //Declare Variables
    Game game;
    GameS saved;
    Card **unique;  // Instead of creating duplicate cards when there is more than one
type of card, a single card is created and that pointer is duplicated the number of
times needed in the draw pile.
    Card **colors;  // Restriction cards for playing wild cards
    fstream setup;  // File used to setup the draw pile for a new game
    fstream save;   // File used to restore the state of the program from when it was
last ran

    //Initialize Variables
    game.hand = nullptr;
    game.bHand = nullptr;
    game.drwPile = nullptr;
    game.drwSize = 0;
    game.drwMax = 0;
    game.plScore = 0;
    game.cpScore = 0;

    colors = genColor();
```



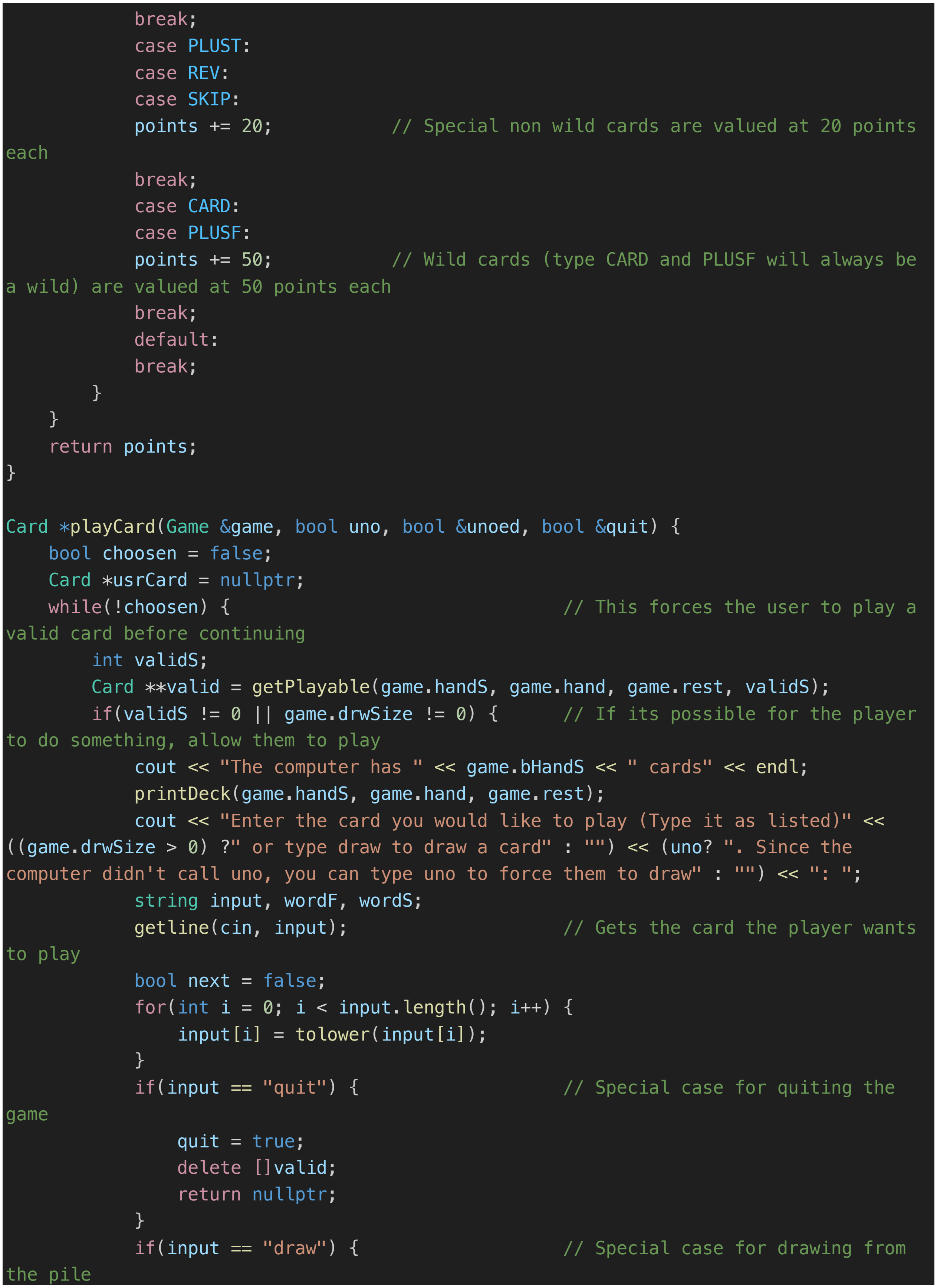
```
    //The Process -> Map Inputs to Outputs
    // Checks if a save file exists and if the game didn't end yet
    save.open("save.data", ios::in | ios::out | ios::binary);
    if(save.is_open()) {
        cout << "Save data found. Resuming using save data." << endl;
        save.read(reinterpret_cast<char *>(&saved), sizeof(saved));
        if(saved.cpScore >= 500 || saved.plScore >= 500) {    // The game ends if
someone scores 500 points. Create a new game if that happened
            cout << "Save data is from a completed game, creating new game." << endl;
            setup.open("start.txt", ios::in);
            unique = setupPile(game, setup);
            setup.close();
            setupGame(game);
        } else {    // Load previous state for game
            unique = loadSave(game, saved, colors, 4);
        }
    } else {
        cout << "No save data found. Starting a new game." << endl;
        // Creates a save file
        save.close();
        save.open("save.data", ios::out | ios::binary);
        save.close();
        save.open("save.data", ios::in | ios::out | ios::binary);
        // Reads file for what cards to include in a new game
        setup.open("start.txt", ios::in);
        unique = setupPile(game, setup);
        setup.close();
        setupGame(game);
    }

    //Display Inputs/Outputs
    // Gives instructions and plays game
    cout << "Welcome to Uno. When you are asked to play a card, you can either play
one or draw. When you are going to have one card left, you must first type and enter
uno before entering the card you wish to play. If the computer forgets to call uno,
you can penalize them by entering uno before playing a card. To save and quit, type
quit when asked to play a card." << endl;
    playGame(game, 4, colors);

    // Saves game state
    save.seekp(0L, ios::beg);
    createSave(game, saved, unique);
    save.write(reinterpret_cast<char*>(&saved), sizeof(saved));

    //Exit the Program
    // Clean up memory
    for(int i = 0; i < 54; i++) {
```



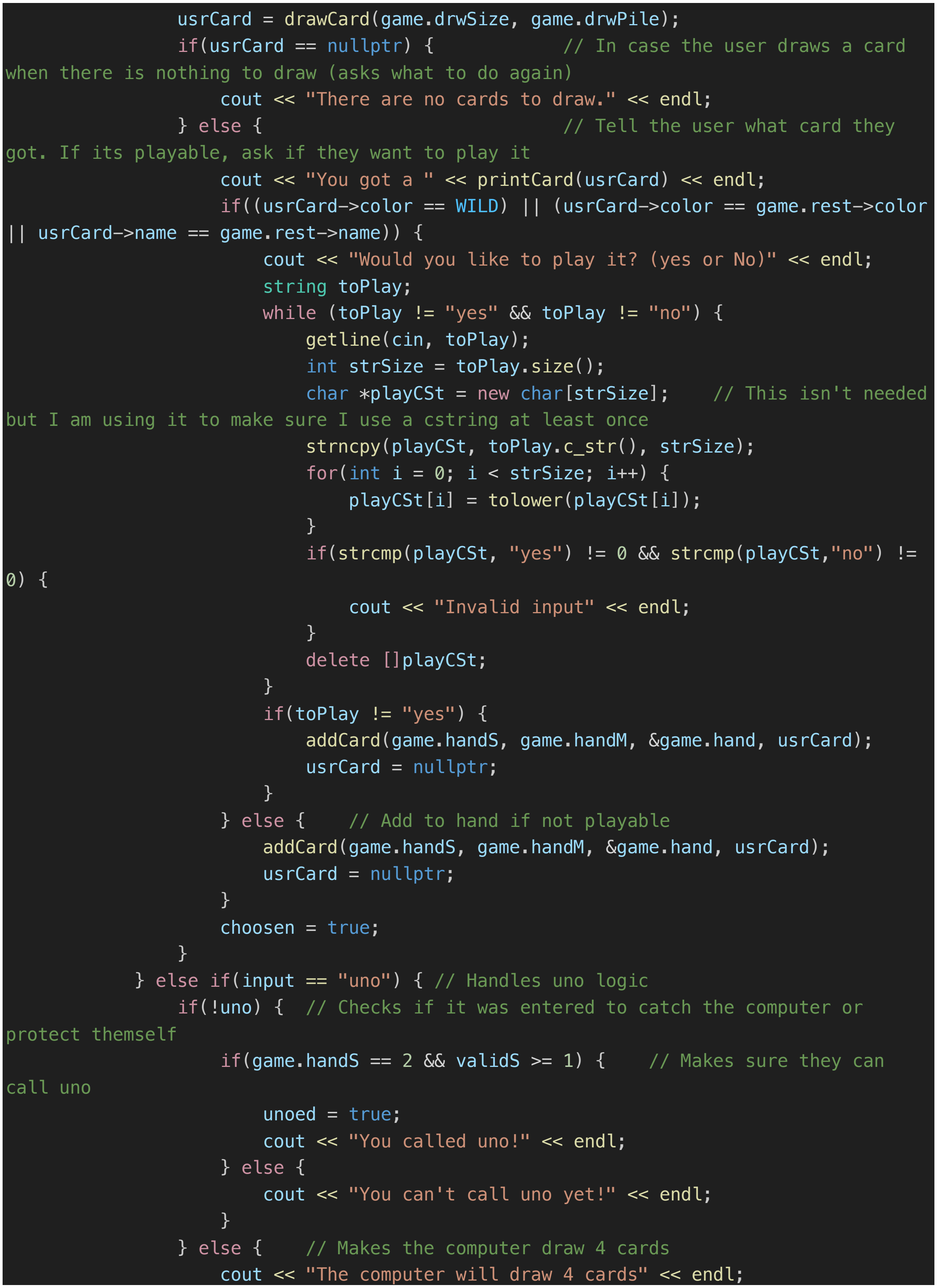
```
        delete unique[i];
    }
    for(int i = 0; i < 4; i++) {
        delete colors[i];
    }
    delete []unique;
    delete []colors;
    delete []game.drwPile;
    delete []game.bHand;
    delete []game.hand;
    save.close();
    return 0;
}

void printDeck(int size, Card **deck, Card *active) {
    cout << "Cards in hand: ";
    for(int i = 0; i < size; i++) {
        cout << printCard(deck[i]) << "; ";     // Prints all the cards in the array
    }
    cout << endl;
    Card **play;
    int playS;
    play = getPlayable(size, deck, active, playS);
    cout << "Playable cards in hand: ";
    for(int i = 0; i < playS; i++) {
        cout << printCard(play[i]) << "; ";     // Prints all playable cards
    }
    cout << endl;
    delete []play;      // Frees memory
}

string printCard(Card *card) {
    string display;
    // Converts the color enum into a human readable string
    switch(card->color) {
        case RED:
        display += "Red ";
        break;
        case GREEN:
        display += "Green ";
        break;
        case YELLOW:
        display += "Yellow ";
        break;
        case BLUE:
        display += "Blue ";
        break;
        case WILD:
```

```
        display += "Wild ";
        break;
        default:
        break;
    }
    // Converts the type enum into a human readable string
    switch(card->name) {
        case ZERO:
        case ONE:
        case TWO:
        case THREE:
        case FOUR:
        case FIVE:
        case SIX:
        case SEVEN:
        case EIGHT:
        case NINE:
        display += to_string(card->name);
        break;
        case PLUST:
        display += "Draw Two";
        break;
        case PLUSF:
        display += "Draw Four";
        break;
        case REV:
        display += "Reverse";
        break;
        case SKIP:
        display += "Skip";
        break;
        case CARD:
        default:
        display += "Card";
        break;
    }
    return display;

}

Card **getPlayable(int size, Card **deck, Card *active, int &plSize) {
    Card **play;                        // Playable cards
    int playS = 0;                      // Playable cards size
    for(int i = 0; i < size; i++) {     // Calculates how large the new array needs to
be to hold the playable cards
        Card *card = deck[i];
        if(card->color == WILD || card->color == active->color || card->name ==
active->name) {
```

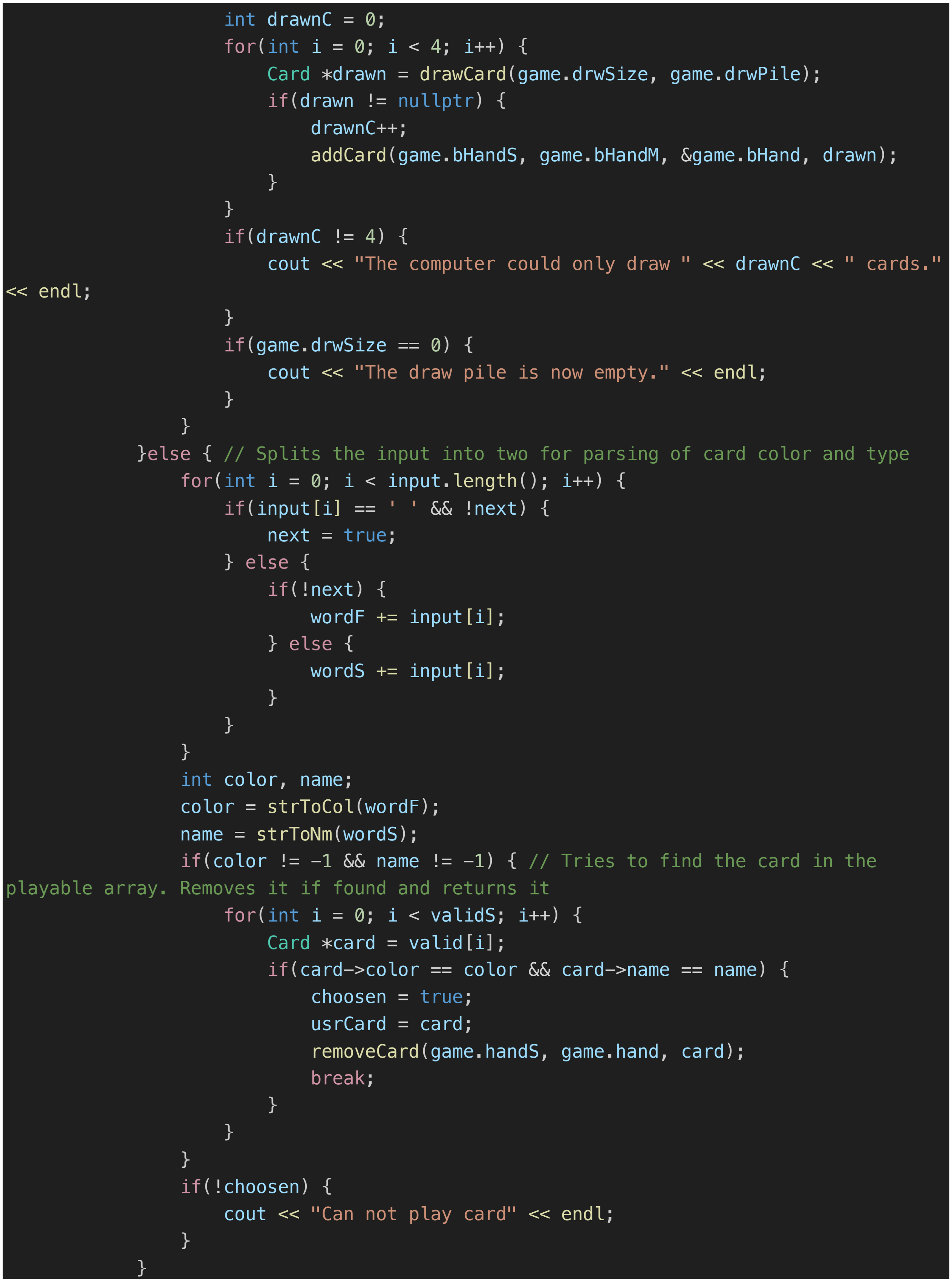
```
            playS++;
        }
    }
    play = new Card*[playS];            // Creates the array with the correct size
    plSize = playS;
    int plIndex = 0;
    for(int i = 0; i < size; i++) {
        if(plIndex >= playS) {          // Ends loop early when it should be done
            break;
        }
        Card *card = deck[i];
        // Adds all playable cards to the list
        if(card->color == WILD || card->color == active->color || card->name ==
active->name) {
            play[plIndex] = card;
            plIndex++;
        }
    }
    return play;
}

bool challengeWin(Card *check, Card **deck, int size) {
    for(int i = 0; i < size; i++) {
        Card *card = deck[i];
        if(check->color == card->color) {       // If there was a playable card with a
matching color, then the challenger wins
            return true;
        }
    }
    return false;
}

unsigned int calcPoints(Card **deck, int size) {
    unsigned int points = 0;
    for(int i = 0; i < size; i++) {
        Card *card = deck[i];
        switch(card->name) {
            case ZERO:
            case ONE:
            case TWO:
            case THREE:
            case FOUR:
            case FIVE:
            case SIX:
            case SEVEN:
            case EIGHT:
            case NINE:
            points += card->name;   // Cards 0-9 are valued at their number
```



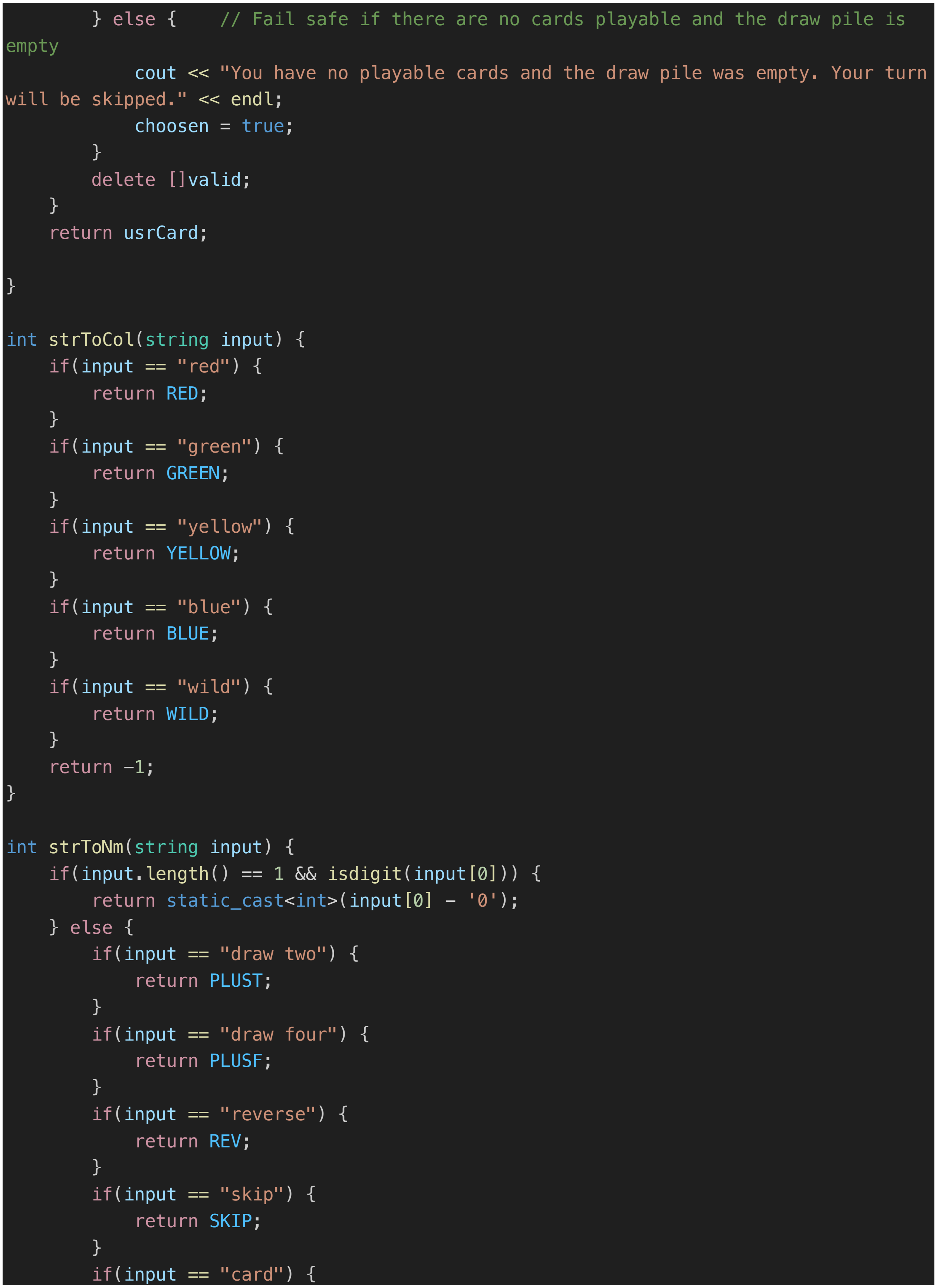
```
            break;
            case PLUST:
            case REV:
            case SKIP:
            points += 20;           // Special non wild cards are valued at 20 points
each
            break;
            case CARD:
            case PLUSF:
            points += 50;           // Wild cards (type CARD and PLUSF will always be
a wild) are valued at 50 points each
            break;
            default:
            break;
        }
    }
    return points;
}

Card *playCard(Game &game, bool uno, bool &unoed, bool &quit) {
    bool choosen = false;
    Card *usrCard = nullptr;
    while(!choosen) {                               // This forces the user to play a
valid card before continuing
        int validS;
        Card **valid = getPlayable(game.handS, game.hand, game.rest, validS);
        if(validS != 0 || game.drwSize != 0) {      // If its possible for the player
to do something, allow them to play
            cout << "The computer has " << game.bHandS << " cards" << endl;
            printDeck(game.handS, game.hand, game.rest);
            cout << "Enter the card you would like to play (Type it as listed)" <<
((game.drwSize > 0) ?" or type draw to draw a card" : "") << (uno? ". Since the
computer didn't call uno, you can type uno to force them to draw" : "") << ": ";
            string input, wordF, wordS;
            getline(cin, input);                    // Gets the card the player wants
to play
            bool next = false;
            for(int i = 0; i < input.length(); i++) {
                input[i] = tolower(input[i]);
            }
            if(input == "quit") {                   // Special case for quiting the
game
                quit = true;
                delete []valid;
                return nullptr;
            }
            if(input == "draw") {                   // Special case for drawing from
the pile
```



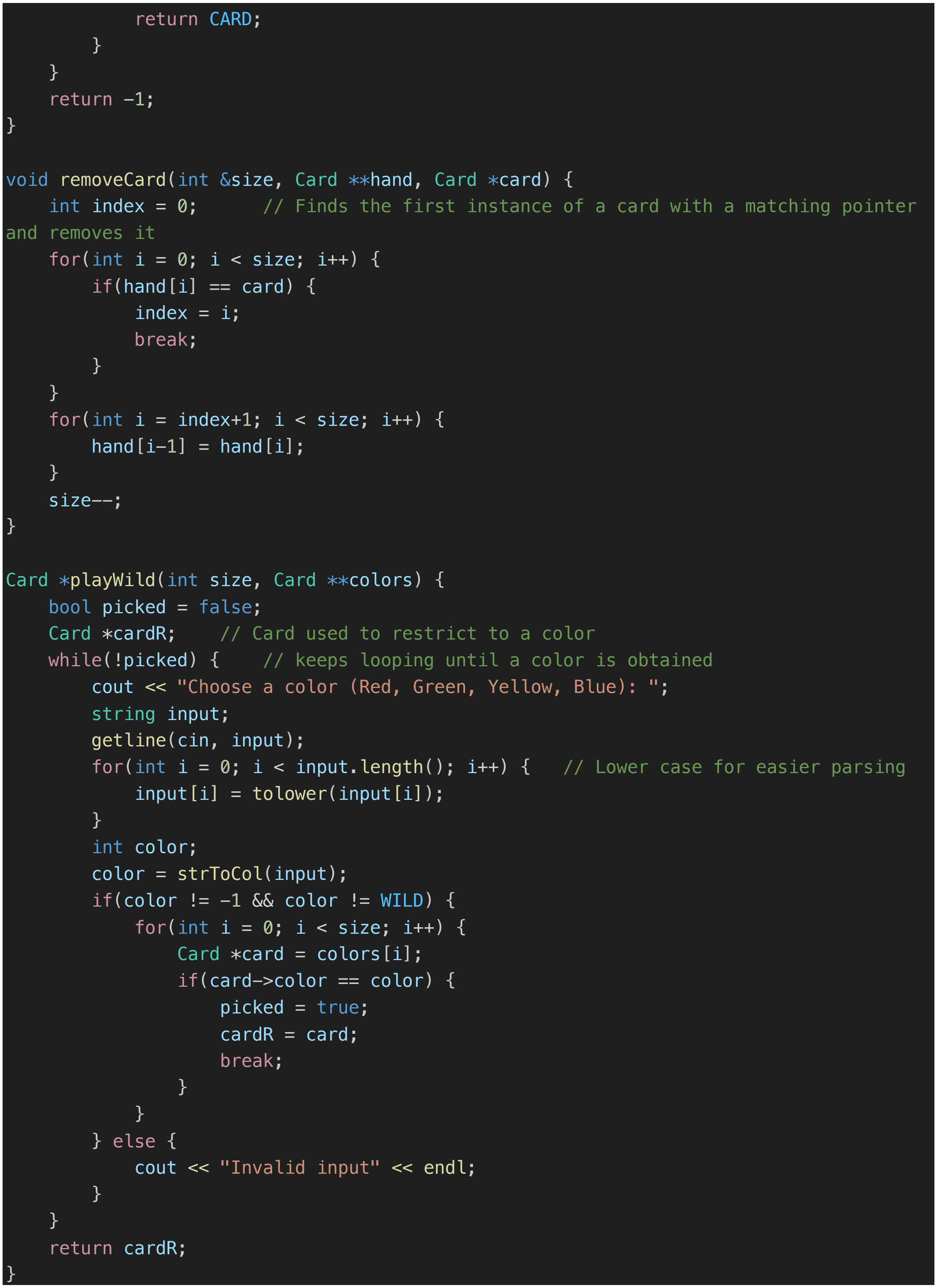
```
                usrCard = drawCard(game.drwSize, game.drwPile);
                if(usrCard == nullptr) {            // In case the user draws a card
when there is nothing to draw (asks what to do again)
                    cout << "There are no cards to draw." << endl;
                } else {                            // Tell the user what card they
got. If its playable, ask if they want to play it
                    cout << "You got a " << printCard(usrCard) << endl;
                    if((usrCard->color == WILD) || (usrCard->color == game.rest->color
|| usrCard->name == game.rest->name)) {
                        cout << "Would you like to play it? (yes or No)" << endl;
                        string toPlay;
                        while (toPlay != "yes" && toPlay != "no") {
                            getline(cin, toPlay);
                            int strSize = toPlay.size();
                            char *playCSt = new char[strSize];    // This isn't needed
but I am using it to make sure I use a cstring at least once
                            strncpy(playCSt, toPlay.c_str(), strSize);
                            for(int i = 0; i < strSize; i++) {
                                playCSt[i] = tolower(playCSt[i]);
                            }
                            if(strcmp(playCSt, "yes") != 0 && strcmp(playCSt,"no") !=
0) {
                                cout << "Invalid input" << endl;
                            }
                            delete []playCSt;
                        }
                        if(toPlay != "yes") {
                            addCard(game.handS, game.handM, &game.hand, usrCard);
                            usrCard = nullptr;
                        }
                    } else {    // Add to hand if not playable
                        addCard(game.handS, game.handM, &game.hand, usrCard);
                        usrCard = nullptr;
                    }
                    choosen = true;
                }
            } else if(input == "uno") { // Handles uno logic
                if(!uno) {  // Checks if it was entered to catch the computer or
protect themself
                    if(game.handS == 2 && validS >= 1) {    // Makes sure they can
call uno
                        unoed = true;
                        cout << "You called uno!" << endl;
                    } else {
                        cout << "You can't call uno yet!" << endl;
                    }
                } else {    // Makes the computer draw 4 cards
                    cout << "The computer will draw 4 cards" << endl;
```



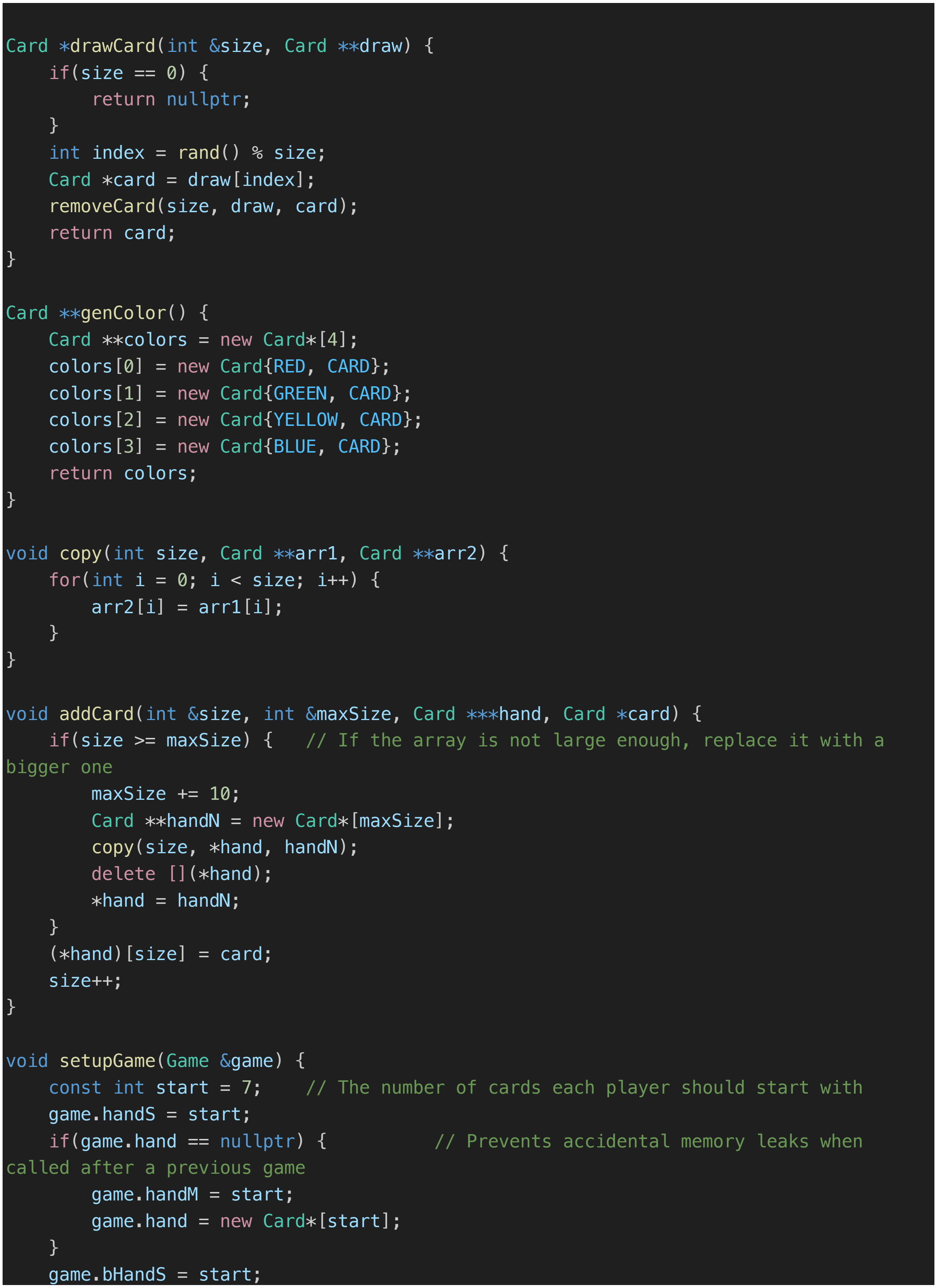
```
                    int drawnC = 0;
                    for(int i = 0; i < 4; i++) {
                        Card *drawn = drawCard(game.drwSize, game.drwPile);
                        if(drawn != nullptr) {
                            drawnC++;
                            addCard(game.bHandS, game.bHandM, &game.bHand, drawn);
                        }
                    }
                    if(drawnC != 4) {
                        cout << "The computer could only draw " << drawnC << " cards."
<< endl;
                    }
                    if(game.drwSize == 0) {
                        cout << "The draw pile is now empty." << endl;
                    }
                }
            }else { // Splits the input into two for parsing of card color and type
                for(int i = 0; i < input.length(); i++) {
                    if(input[i] == ' ' && !next) {
                        next = true;
                    } else {
                        if(!next) {
                            wordF += input[i];
                        } else {
                            wordS += input[i];
                        }
                    }
                }
                int color, name;
                color = strToCol(wordF);
                name = strToNm(wordS);
                if(color != -1 && name != -1) { // Tries to find the card in the
playable array. Removes it if found and returns it
                    for(int i = 0; i < validS; i++) {
                        Card *card = valid[i];
                        if(card->color == color && card->name == name) {
                            choosen = true;
                            usrCard = card;
                            removeCard(game.handS, game.hand, card);
                            break;
                        }
                    }
                }
                if(!choosen) {
                    cout << "Can not play card" << endl;
                }
            }
```



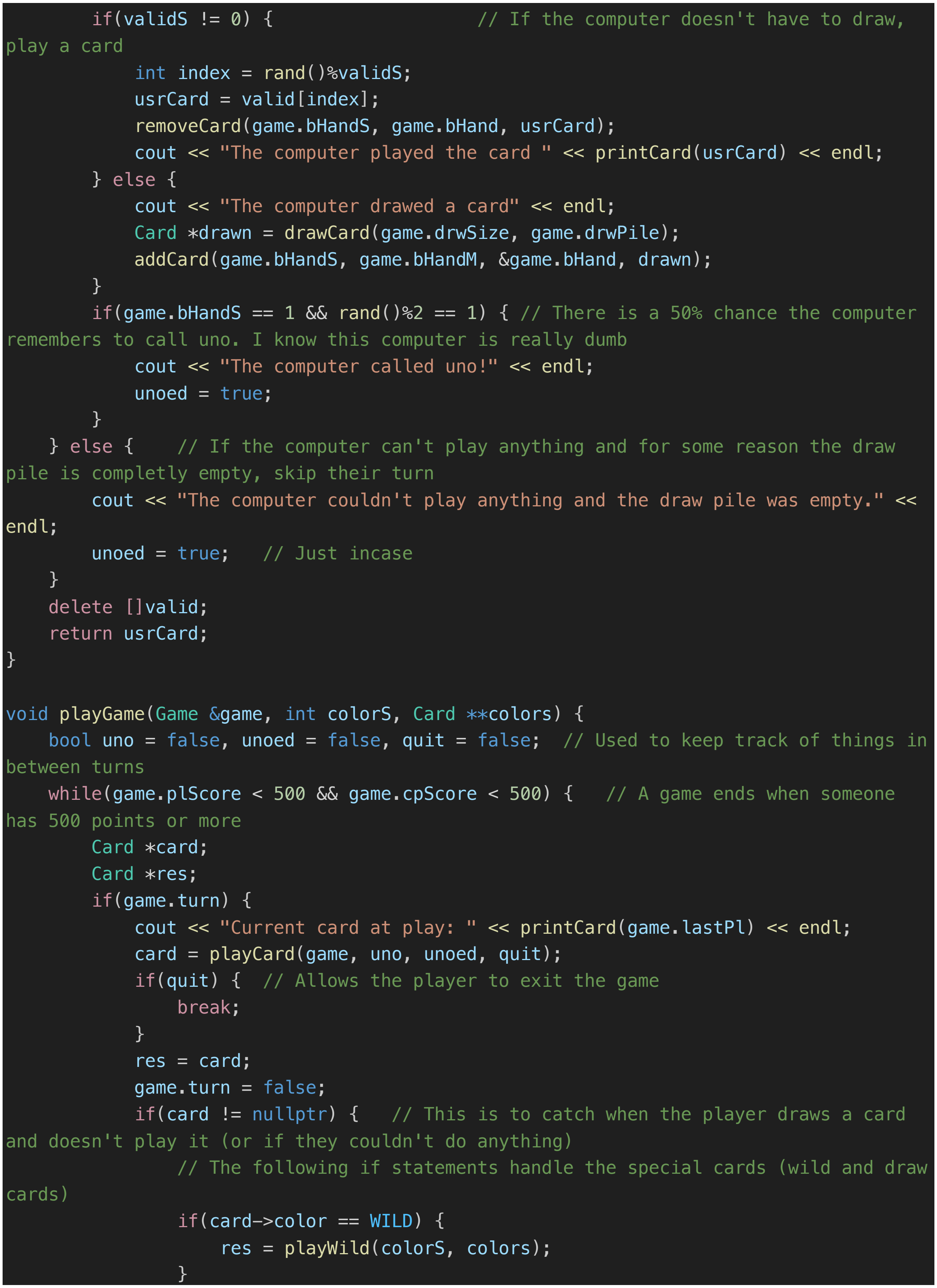
```
        } else {    // Fail safe if there are no cards playable and the draw pile is
empty
            cout << "You have no playable cards and the draw pile was empty. Your turn
will be skipped." << endl;
            choosen = true;
        }
        delete []valid;
    }
    return usrCard;

}

int strToCol(string input) {
    if(input == "red") {
        return RED;
    }
    if(input == "green") {
        return GREEN;
    }
    if(input == "yellow") {
        return YELLOW;
    }
    if(input == "blue") {
        return BLUE;
    }
    if(input == "wild") {
        return WILD;
    }
    return -1;
}

int strToNm(string input) {
    if(input.length() == 1 && isdigit(input[0])) {
        return static_cast<int>(input[0] - '0');
    } else {
        if(input == "draw two") {
            return PLUST;
        }
        if(input == "draw four") {
            return PLUSF;
        }
        if(input == "reverse") {
            return REV;
        }
        if(input == "skip") {
            return SKIP;
        }
        if(input == "card") {
```



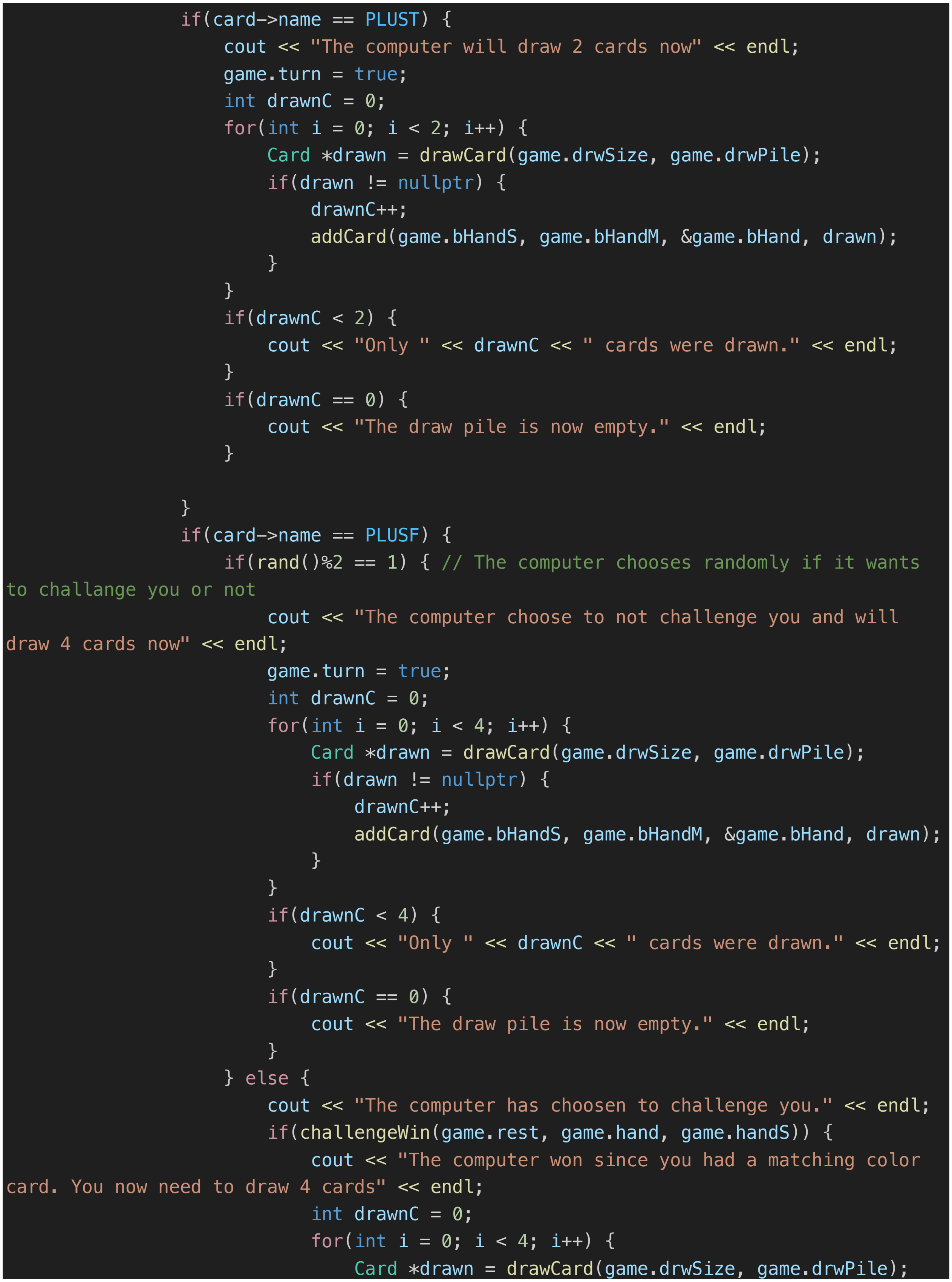
```
            return CARD;
        }
    }
    return -1;
}

void removeCard(int &size, Card **hand, Card *card) {
    int index = 0;      // Finds the first instance of a card with a matching pointer
and removes it
    for(int i = 0; i < size; i++) {
        if(hand[i] == card) {
            index = i;
            break;
        }
    }
    for(int i = index+1; i < size; i++) {
        hand[i-1] = hand[i];
    }
    size--;
}

Card *playWild(int size, Card **colors) {
    bool picked = false;
    Card *cardR;    // Card used to restrict to a color
    while(!picked) {    // keeps looping until a color is obtained
        cout << "Choose a color (Red, Green, Yellow, Blue): ";
        string input;
        getline(cin, input);
        for(int i = 0; i < input.length(); i++) {   // Lower case for easier parsing
            input[i] = tolower(input[i]);
        }
        int color;
        color = strToCol(input);
        if(color != -1 && color != WILD) {
            for(int i = 0; i < size; i++) {
                Card *card = colors[i];
                if(card->color == color) {
                    picked = true;
                    cardR = card;
                    break;
                }
            }
        } else {
            cout << "Invalid input" << endl;
        }
    }
    return cardR;
}
```



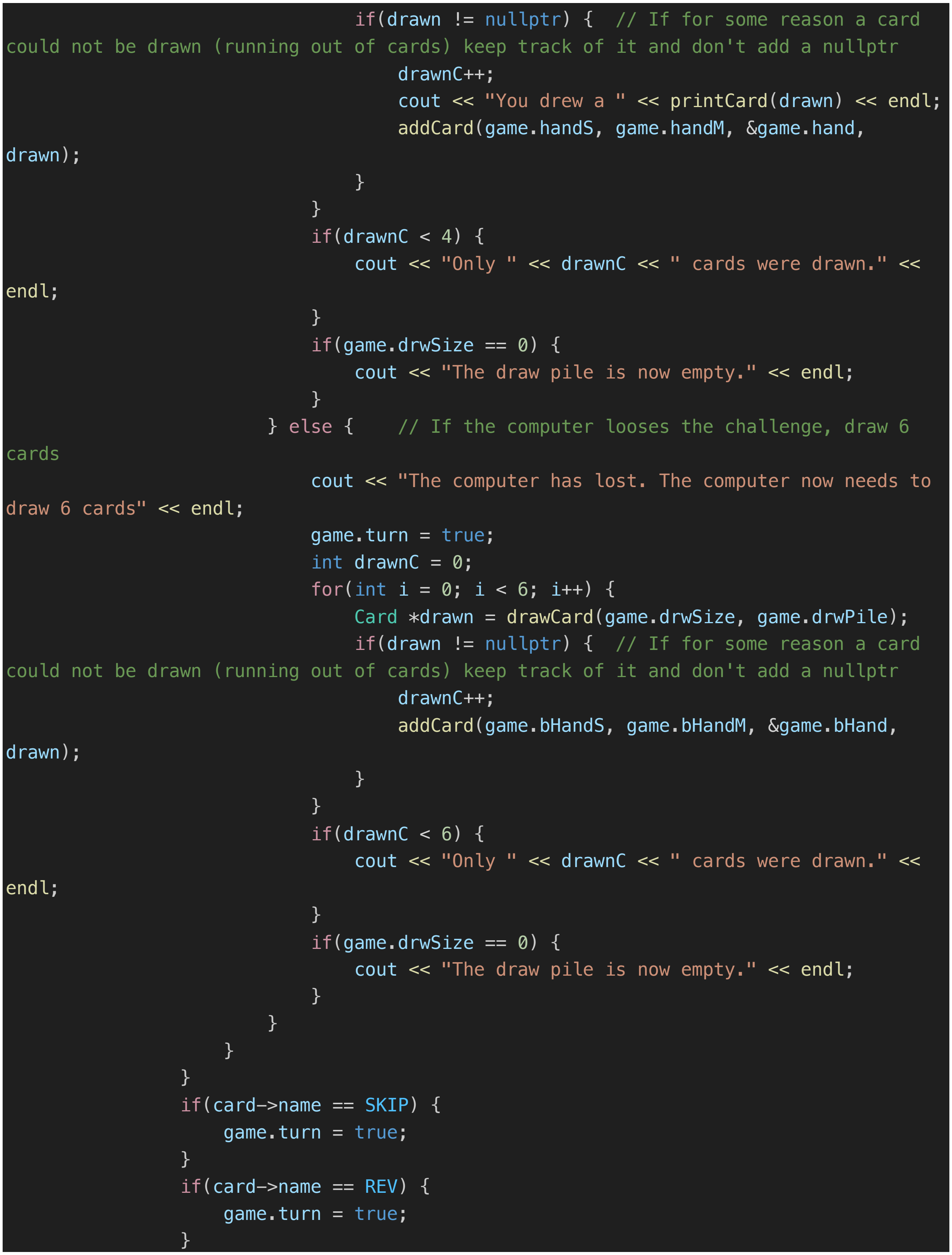
```
Card *drawCard(int &size, Card **draw) {
    if(size == 0) {
        return nullptr;
    }
    int index = rand() % size;
    Card *card = draw[index];
    removeCard(size, draw, card);
    return card;
}

Card **genColor() {
    Card **colors = new Card*[4];
    colors[0] = new Card{RED, CARD};
    colors[1] = new Card{GREEN, CARD};
    colors[2] = new Card{YELLOW, CARD};
    colors[3] = new Card{BLUE, CARD};
    return colors;
}

void copy(int size, Card **arr1, Card **arr2) {
    for(int i = 0; i < size; i++) {
        arr2[i] = arr1[i];
    }
}

void addCard(int &size, int &maxSize, Card ***hand, Card *card) {
    if(size >= maxSize) {   // If the array is not large enough, replace it with a
bigger one
        maxSize += 10;
        Card **handN = new Card*[maxSize];
        copy(size, *hand, handN);
        delete [](*hand);
        *hand = handN;
    }
    (*hand)[size] = card;
    size++;
}

void setupGame(Game &game) {
    const int start = 7;    // The number of cards each player should start with
    game.handS = start;
    if(game.hand == nullptr) {          // Prevents accidental memory leaks when
called after a previous game
        game.handM = start;
        game.hand = new Card*[start];
    }
    game.bHandS = start;
    if(game.bHand == nullptr) {         // Prevents accidental memory leaks when
called after a previous game
        game.bHandM = start;
        game.bHand = new Card*[start];
    }
    for(int i = 0; i < start; i++) {    // Fills the computer's and player's hand with
cards
        Card *card = drawCard(game.drwSize, game.drwPile);
        game.hand[i] = card;
        card = drawCard(game.drwSize, game.drwPile);
        game.bHand[i] = card;
    }
    Card *card = drawCard(game.drwSize, game.drwPile);
    while(card->color == WILD) {        // If the first played card is a wild, return
it to the draw pile and draw another
        addCard(game.drwSize, game.drwMax, &game.drwPile, card);
        card = drawCard(game.drwSize, game.drwPile);
    }
    game.lastPl = card;
    game.rest = card;
    game.turn = true;
}

Card *playCom(Game &game, bool uno, bool &unoed) {
    if(uno) {   // If the player forgot to call uno, there is a 50% change the
computer catches it
        if(rand()%2 == 1) {
            cout << "The computer caught you with one card!" << endl;
            int drawnC = 0;
            for(int i = 0; i < 4; i++) {
                Card *drawn = drawCard(game.drwSize, game.drwPile);
                if(drawn != nullptr) {
                    drawnC++;
                    addCard(game.handS, game.handM, &game.hand, drawn);
                }
            }
            if(drawnC != 4) {
                cout << "You could only draw " << drawnC << " cards." << endl;
            }
            if(game.drwSize == 0) {
                cout << "The draw pile is now empty." << endl;
            }
        }
    }
    Card *usrCard = nullptr;
    int validS;
    Card **valid = getPlayable(game.bHandS, game.bHand, game.rest, validS);
    if(validS != 0 || game.drwSize != 0) {  // If the computer can do something
```



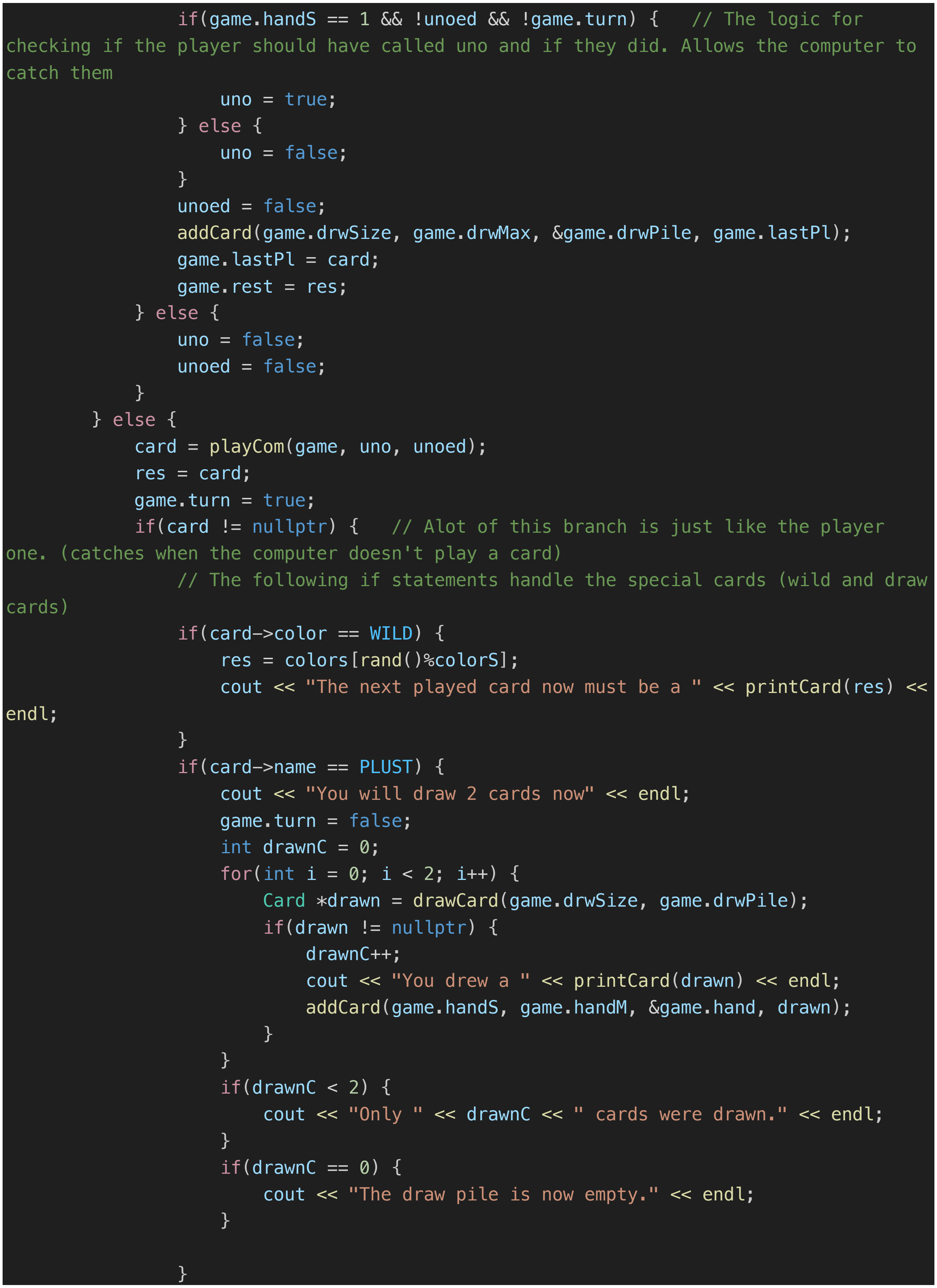
```
        if(validS != 0) {                   // If the computer doesn't have to draw, play a card
            int index = rand()%validS;
            usrCard = valid[index];
            removeCard(game.bHandS, game.bHand, usrCard);
            cout << "The computer played the card " << printCard(usrCard) << endl;
        } else {
            cout << "The computer drawed a card" << endl;
            Card *drawn = drawCard(game.drwSize, game.drwPile);
            addCard(game.bHandS, game.bHandM, &game.bHand, drawn);
        }
        if(game.bHandS == 1 && rand()%2 == 1) { // There is a 50% chance the computer remembers to call uno. I know this computer is really dumb
            cout << "The computer called uno!" << endl;
            unoed = true;
        }
    } else {    // If the computer can't play anything and for some reason the draw pile is completly empty, skip their turn
        cout << "The computer couldn't play anything and the draw pile was empty." << endl;
        unoed = true;   // Just incase
    }
    delete []valid;
    return usrCard;
}

void playGame(Game &game, int colorS, Card **colors) {
    bool uno = false, unoed = false, quit = false;  // Used to keep track of things in between turns
    while(game.plScore < 500 && game.cpScore < 500) {   // A game ends when someone has 500 points or more
        Card *card;
        Card *res;
        if(game.turn) {
            cout << "Current card at play: " << printCard(game.lastPl) << endl;
            card = playCard(game, uno, unoed, quit);
            if(quit) {  // Allows the player to exit the game
                break;
            }
            res = card;
            game.turn = false;
            if(card != nullptr) {   // This is to catch when the player draws a card and doesn't play it (or if they couldn't do anything)
                // The following if statements handle the special cards (wild and draw cards)
                if(card->color == WILD) {
                    res = playWild(colorS, colors);
                }
```



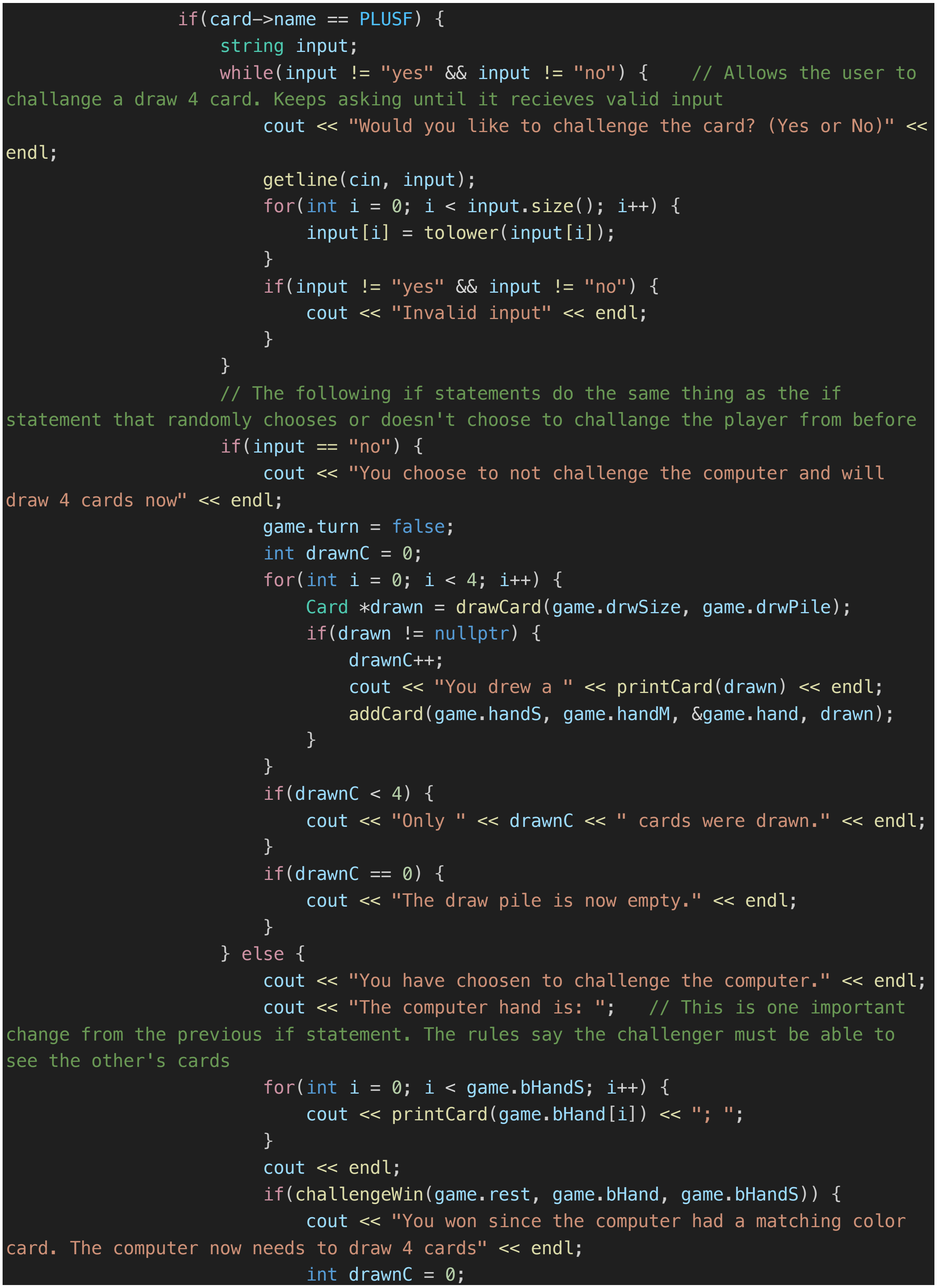
```
                if(card->name == PLUST) {
                    cout << "The computer will draw 2 cards now" << endl;
                    game.turn = true;
                    int drawnC = 0;
                    for(int i = 0; i < 2; i++) {
                        Card *drawn = drawCard(game.drwSize, game.drwPile);
                        if(drawn != nullptr) {
                            drawnC++;
                            addCard(game.bHandS, game.bHandM, &game.bHand, drawn);
                        }
                    }
                    if(drawnC < 2) {
                        cout << "Only " << drawnC << " cards were drawn." << endl;
                    }
                    if(drawnC == 0) {
                        cout << "The draw pile is now empty." << endl;
                    }

                }
                if(card->name == PLUSF) {
                    if(rand()%2 == 1) { // The computer chooses randomly if it wants
to challange you or not
                        cout << "The computer choose to not challenge you and will
draw 4 cards now" << endl;
                        game.turn = true;
                        int drawnC = 0;
                        for(int i = 0; i < 4; i++) {
                            Card *drawn = drawCard(game.drwSize, game.drwPile);
                            if(drawn != nullptr) {
                                drawnC++;
                                addCard(game.bHandS, game.bHandM, &game.bHand, drawn);
                            }
                        }
                        if(drawnC < 4) {
                            cout << "Only " << drawnC << " cards were drawn." << endl;
                        }
                        if(drawnC == 0) {
                            cout << "The draw pile is now empty." << endl;
                        }
                    } else {
                        cout << "The computer has choosen to challenge you." << endl;
                        if(challengeWin(game.rest, game.hand, game.handS)) {
                            cout << "The computer won since you had a matching color
card. You now need to draw 4 cards" << endl;
                            int drawnC = 0;
                            for(int i = 0; i < 4; i++) {
                                Card *drawn = drawCard(game.drwSize, game.drwPile);
```



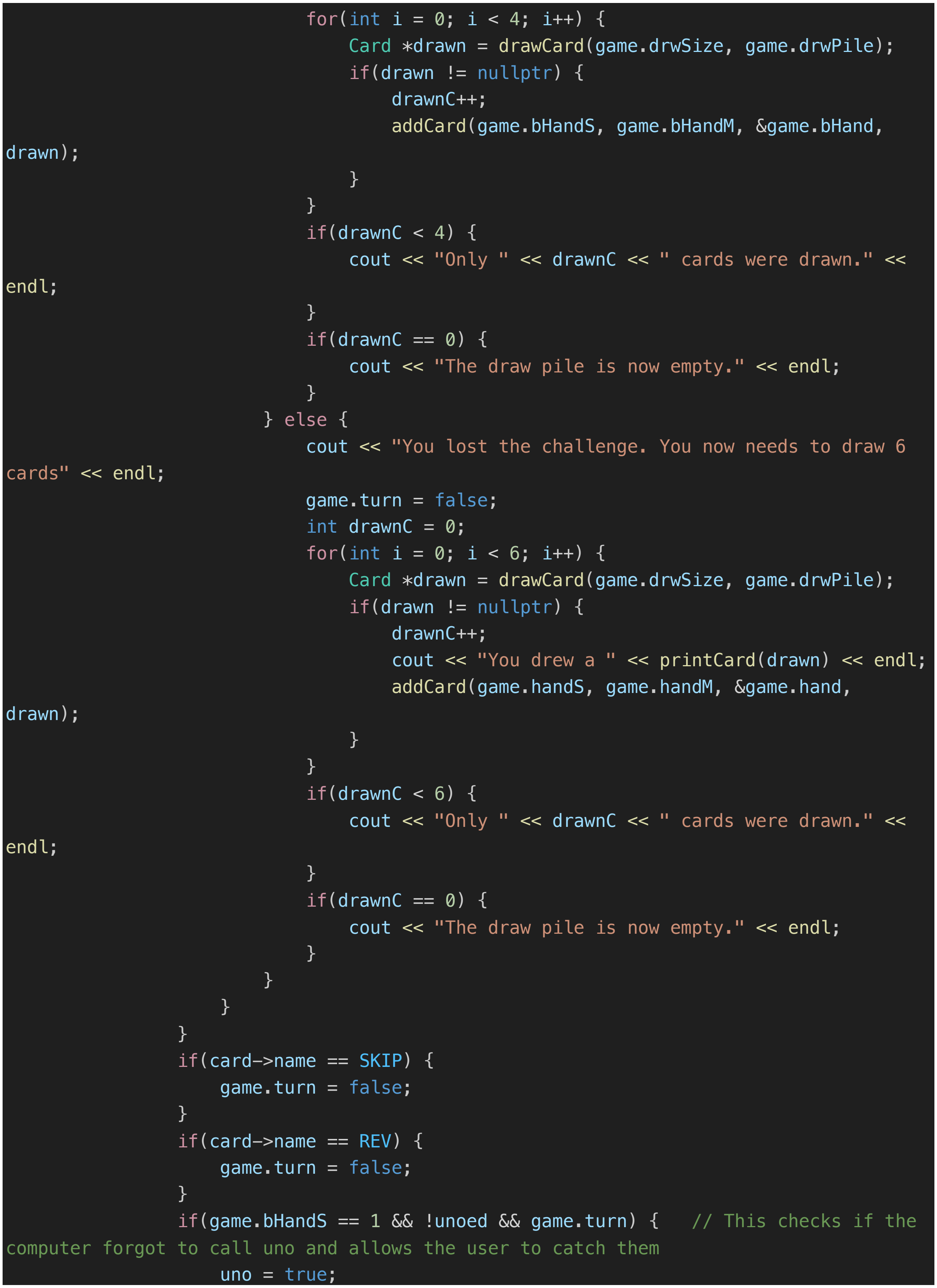
```
                                if(drawn != nullptr) {  // If for some reason a card
could not be drawn (running out of cards) keep track of it and don't add a nullptr
                                    drawnC++;
                                    cout << "You drew a " << printCard(drawn) << endl;
                                    addCard(game.handS, game.handM, &game.hand,
drawn);
                                }
                            }
                            if(drawnC < 4) {
                                cout << "Only " << drawnC << " cards were drawn." <<
endl;
                            }
                            if(game.drwSize == 0) {
                                cout << "The draw pile is now empty." << endl;
                            }
                        } else {    // If the computer looses the challenge, draw 6
cards
                            cout << "The computer has lost. The computer now needs to
draw 6 cards" << endl;
                            game.turn = true;
                            int drawnC = 0;
                            for(int i = 0; i < 6; i++) {
                                Card *drawn = drawCard(game.drwSize, game.drwPile);
                                if(drawn != nullptr) {  // If for some reason a card
could not be drawn (running out of cards) keep track of it and don't add a nullptr
                                    drawnC++;
                                    addCard(game.bHandS, game.bHandM, &game.bHand,
drawn);
                                }
                            }
                            if(drawnC < 6) {
                                cout << "Only " << drawnC << " cards were drawn." <<
endl;
                            }
                            if(game.drwSize == 0) {
                                cout << "The draw pile is now empty." << endl;
                            }
                        }
                    }
                }
                if(card->name == SKIP) {
                    game.turn = true;
                }
                if(card->name == REV) {
                    game.turn = true;
                }
```



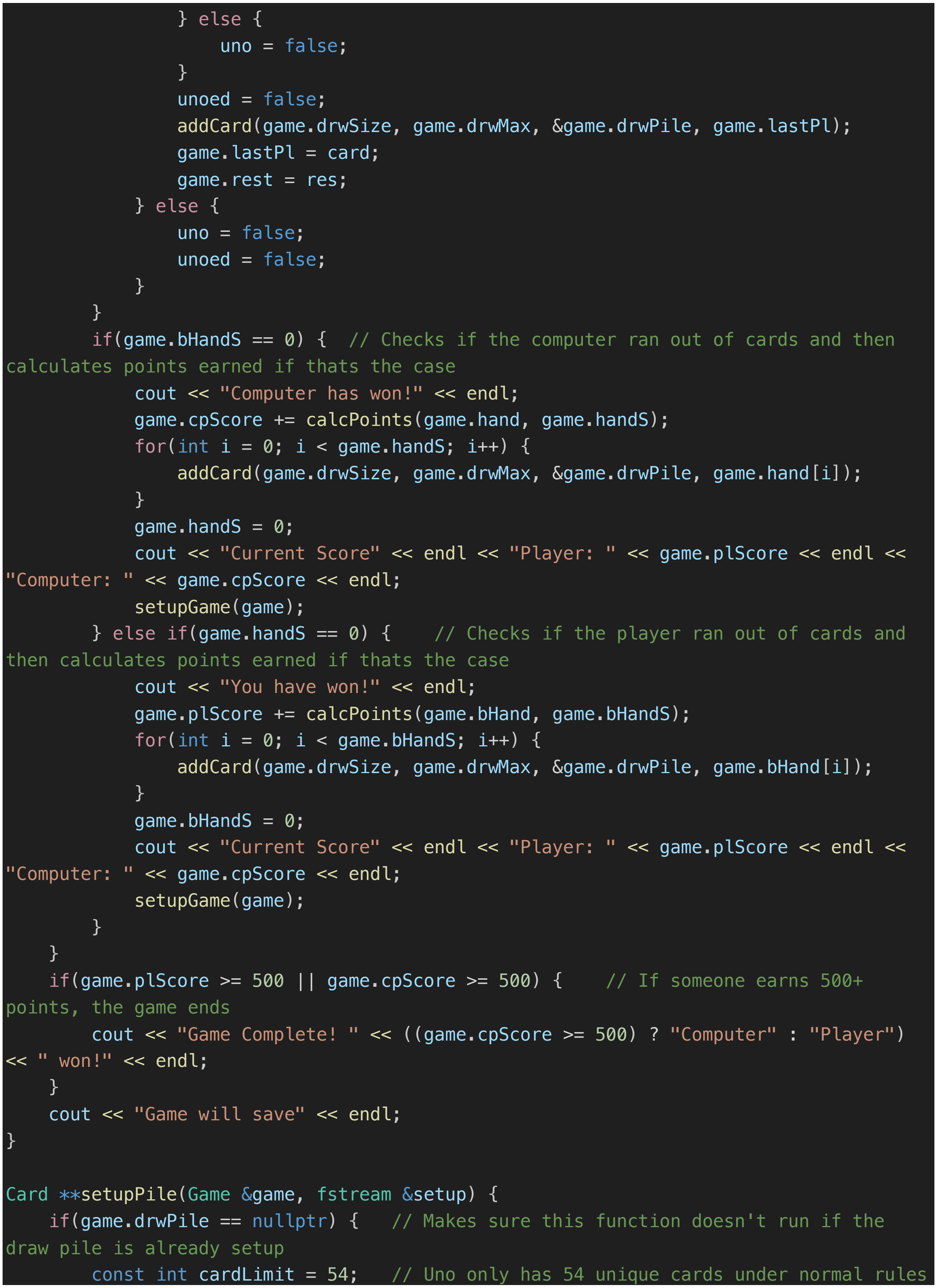
```
                if(game.handS == 1 && !unoed && !game.turn) {   // The logic for checking if the player should have called uno and if they did. Allows the computer to catch them
                    uno = true;
                } else {
                    uno = false;
                }
                unoed = false;
                addCard(game.drwSize, game.drwMax, &game.drwPile, game.lastPl);
                game.lastPl = card;
                game.rest = res;
            } else {
                uno = false;
                unoed = false;
            }
        } else {
            card = playCom(game, uno, unoed);
            res = card;
            game.turn = true;
            if(card != nullptr) {   // Alot of this branch is just like the player one. (catches when the computer doesn't play a card)
                // The following if statements handle the special cards (wild and draw cards)
                if(card->color == WILD) {
                    res = colors[rand()%colorS];
                    cout << "The next played card now must be a " << printCard(res) << endl;
                }
                if(card->name == PLUST) {
                    cout << "You will draw 2 cards now" << endl;
                    game.turn = false;
                    int drawnC = 0;
                    for(int i = 0; i < 2; i++) {
                        Card *drawn = drawCard(game.drwSize, game.drwPile);
                        if(drawn != nullptr) {
                            drawnC++;
                            cout << "You drew a " << printCard(drawn) << endl;
                            addCard(game.handS, game.handM, &game.hand, drawn);
                        }
                    }
                    if(drawnC < 2) {
                        cout << "Only " << drawnC << " cards were drawn." << endl;
                    }
                    if(drawnC == 0) {
                        cout << "The draw pile is now empty." << endl;
                    }

                }
```



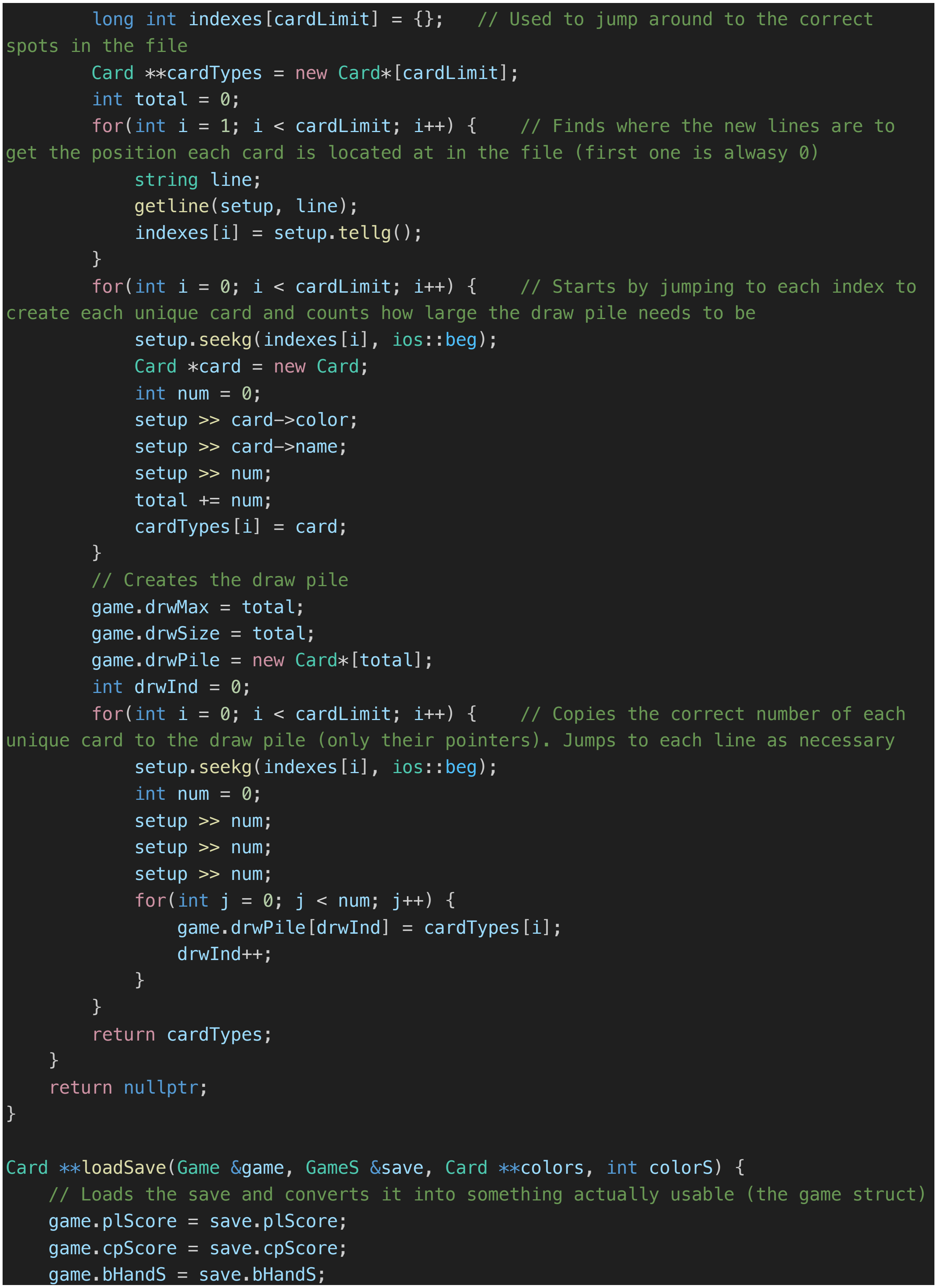
```
                if(card->name == PLUSF) {
                    string input;
                    while(input != "yes" && input != "no") {    // Allows the user to
challange a draw 4 card. Keeps asking until it recieves valid input
                        cout << "Would you like to challenge the card? (Yes or No)" <<
endl;
                        getline(cin, input);
                        for(int i = 0; i < input.size(); i++) {
                            input[i] = tolower(input[i]);
                        }
                        if(input != "yes" && input != "no") {
                            cout << "Invalid input" << endl;
                        }
                    }
                    // The following if statements do the same thing as the if
statement that randomly chooses or doesn't choose to challange the player from before
                    if(input == "no") {
                        cout << "You choose to not challenge the computer and will
draw 4 cards now" << endl;
                        game.turn = false;
                        int drawnC = 0;
                        for(int i = 0; i < 4; i++) {
                            Card *drawn = drawCard(game.drwSize, game.drwPile);
                            if(drawn != nullptr) {
                                drawnC++;
                                cout << "You drew a " << printCard(drawn) << endl;
                                addCard(game.handS, game.handM, &game.hand, drawn);
                            }
                        }
                        if(drawnC < 4) {
                            cout << "Only " << drawnC << " cards were drawn." << endl;
                        }
                        if(drawnC == 0) {
                            cout << "The draw pile is now empty." << endl;
                        }
                    } else {
                        cout << "You have choosen to challenge the computer." << endl;
                        cout << "The computer hand is: ";   // This is one important
change from the previous if statement. The rules say the challenger must be able to
see the other's cards
                        for(int i = 0; i < game.bHandS; i++) {
                            cout << printCard(game.bHand[i]) << "; ";
                        }
                        cout << endl;
                        if(challengeWin(game.rest, game.bHand, game.bHandS)) {
                            cout << "You won since the computer had a matching color
card. The computer now needs to draw 4 cards" << endl;
                            int drawnC = 0;
```



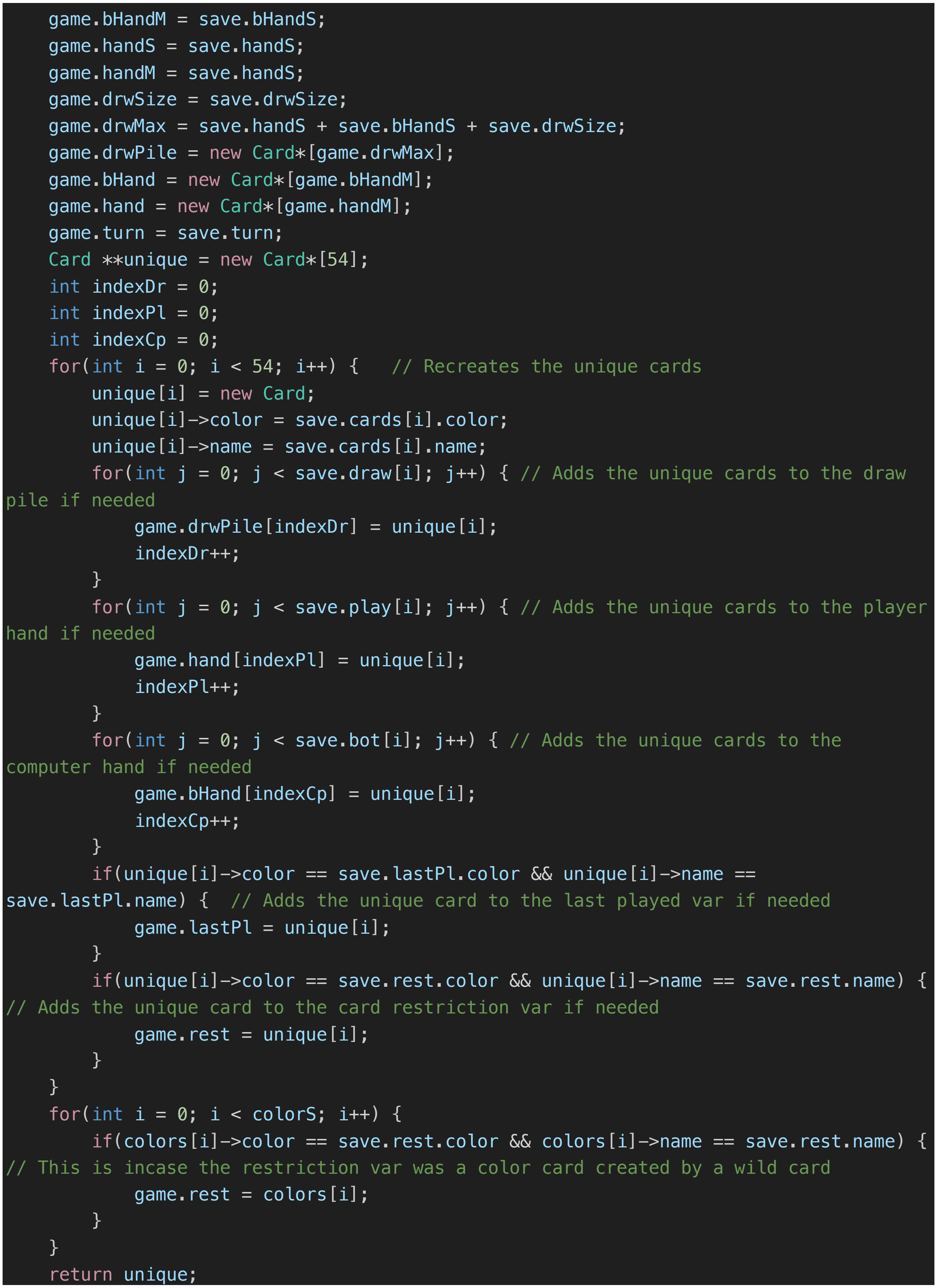
```
                            for(int i = 0; i < 4; i++) {
                                Card *drawn = drawCard(game.drwSize, game.drwPile);
                                if(drawn != nullptr) {
                                    drawnC++;
                                    addCard(game.bHandS, game.bHandM, &game.bHand,
drawn);
                                }
                            }
                            if(drawnC < 4) {
                                cout << "Only " << drawnC << " cards were drawn." <<
endl;
                            }
                            if(drawnC == 0) {
                                cout << "The draw pile is now empty." << endl;
                            }
                        } else {
                            cout << "You lost the challenge. You now needs to draw 6
cards" << endl;
                            game.turn = false;
                            int drawnC = 0;
                            for(int i = 0; i < 6; i++) {
                                Card *drawn = drawCard(game.drwSize, game.drwPile);
                                if(drawn != nullptr) {
                                    drawnC++;
                                    cout << "You drew a " << printCard(drawn) << endl;
                                    addCard(game.handS, game.handM, &game.hand,
drawn);
                                }
                            }
                            if(drawnC < 6) {
                                cout << "Only " << drawnC << " cards were drawn." <<
endl;
                            }
                            if(drawnC == 0) {
                                cout << "The draw pile is now empty." << endl;
                            }
                        }
                    }
                }
                if(card->name == SKIP) {
                    game.turn = false;
                }
                if(card->name == REV) {
                    game.turn = false;
                }
                if(game.bHandS == 1 && !unoed && game.turn) {   // This checks if the
computer forgot to call uno and allows the user to catch them
                    uno = true;
```



```
                } else {
                    uno = false;
                }
                unoed = false;
                addCard(game.drwSize, game.drwMax, &game.drwPile, game.lastPl);
                game.lastPl = card;
                game.rest = res;
            } else {
                uno = false;
                unoed = false;
            }
        }
        if(game.bHandS == 0) {  // Checks if the computer ran out of cards and then calculates points earned if thats the case
            cout << "Computer has won!" << endl;
            game.cpScore += calcPoints(game.hand, game.handS);
            for(int i = 0; i < game.handS; i++) {
                addCard(game.drwSize, game.drwMax, &game.drwPile, game.hand[i]);
            }
            game.handS = 0;
            cout << "Current Score" << endl << "Player: " << game.plScore << endl << "Computer: " << game.cpScore << endl;
            setupGame(game);
        } else if(game.handS == 0) {    // Checks if the player ran out of cards and then calculates points earned if thats the case
            cout << "You have won!" << endl;
            game.plScore += calcPoints(game.bHand, game.bHandS);
            for(int i = 0; i < game.bHandS; i++) {
                addCard(game.drwSize, game.drwMax, &game.drwPile, game.bHand[i]);
            }
            game.bHandS = 0;
            cout << "Current Score" << endl << "Player: " << game.plScore << endl << "Computer: " << game.cpScore << endl;
            setupGame(game);
        }
    }
    if(game.plScore >= 500 || game.cpScore >= 500) {    // If someone earns 500+ points, the game ends
        cout << "Game Complete! " << ((game.cpScore >= 500) ? "Computer" : "Player") << " won!" << endl;
    }
    cout << "Game will save" << endl;
}

Card **setupPile(Game &game, fstream &setup) {
    if(game.drwPile == nullptr) {   // Makes sure this function doesn't run if the draw pile is already setup
        const int cardLimit = 54;   // Uno only has 54 unique cards under normal rules
```



```
        long int indexes[cardLimit] = {};   // Used to jump around to the correct spots in the file
        Card **cardTypes = new Card*[cardLimit];
        int total = 0;
        for(int i = 1; i < cardLimit; i++) {    // Finds where the new lines are to get the position each card is located at in the file (first one is alwasy 0)
            string line;
            getline(setup, line);
            indexes[i] = setup.tellg();
        }
        for(int i = 0; i < cardLimit; i++) {    // Starts by jumping to each index to create each unique card and counts how large the draw pile needs to be
            setup.seekg(indexes[i], ios::beg);
            Card *card = new Card;
            int num = 0;
            setup >> card->color;
            setup >> card->name;
            setup >> num;
            total += num;
            cardTypes[i] = card;
        }
        // Creates the draw pile
        game.drwMax = total;
        game.drwSize = total;
        game.drwPile = new Card*[total];
        int drwInd = 0;
        for(int i = 0; i < cardLimit; i++) {    // Copies the correct number of each unique card to the draw pile (only their pointers). Jumps to each line as necessary
            setup.seekg(indexes[i], ios::beg);
            int num = 0;
            setup >> num;
            setup >> num;
            setup >> num;
            for(int j = 0; j < num; j++) {
                game.drwPile[drwInd] = cardTypes[i];
                drwInd++;
            }
        }
        return cardTypes;
    }
    return nullptr;
}

Card **loadSave(Game &game, GameS &save, Card **colors, int colorS) {
    // Loads the save and converts it into something actually usable (the game struct)
    game.plScore = save.plScore;
    game.cpScore = save.cpScore;
    game.bHandS = save.bHandS;
```



```
    game.bHandM = save.bHandS;
    game.handS = save.handS;
    game.handM = save.handS;
    game.drwSize = save.drwSize;
    game.drwMax = save.handS + save.bHandS + save.drwSize;
    game.drwPile = new Card*[game.drwMax];
    game.bHand = new Card*[game.bHandM];
    game.hand = new Card*[game.handM];
    game.turn = save.turn;
    Card **unique = new Card*[54];
    int indexDr = 0;
    int indexPl = 0;
    int indexCp = 0;
    for(int i = 0; i < 54; i++) {   // Recreates the unique cards
        unique[i] = new Card;
        unique[i]->color = save.cards[i].color;
        unique[i]->name = save.cards[i].name;
        for(int j = 0; j < save.draw[i]; j++) { // Adds the unique cards to the draw pile if needed
            game.drwPile[indexDr] = unique[i];
            indexDr++;
        }
        for(int j = 0; j < save.play[i]; j++) { // Adds the unique cards to the player hand if needed
            game.hand[indexPl] = unique[i];
            indexPl++;
        }
        for(int j = 0; j < save.bot[i]; j++) { // Adds the unique cards to the computer hand if needed
            game.bHand[indexCp] = unique[i];
            indexCp++;
        }
        if(unique[i]->color == save.lastPl.color && unique[i]->name == save.lastPl.name) {  // Adds the unique card to the last played var if needed
            game.lastPl = unique[i];
        }
        if(unique[i]->color == save.rest.color && unique[i]->name == save.rest.name) { // Adds the unique card to the card restriction var if needed
            game.rest = unique[i];
        }
    }
    for(int i = 0; i < colorS; i++) {
        if(colors[i]->color == save.rest.color && colors[i]->name == save.rest.name) { // This is incase the restriction var was a color card created by a wild card
            game.rest = colors[i];
        }
    }
    return unique;
```

```
}

void createSave(Game &game, GameS &save, Card **unique) {
    // Saves everything
    save.plScore = game.plScore;
    save.cpScore = game.cpScore;
    save.bHandS = game.bHandS;
    save.handS = game.handS;
    save.drwSize = game.drwSize;
    save.turn = game.turn;
    save.lastPl.color = game.lastPl->color;
    save.lastPl.name  = game.lastPl->name;
    save.rest.color = game.rest->color;
    save.rest.name  = game.rest->name;
    for (int i = 0; i < 54; i++) {  // Searches for each card type, counts them, and then saves them to the correct variable
        save.draw[i] = 0;
        save.play[i] = 0;
        save.bot[i]  = 0;
        save.cards[i].color = unique[i]->color;
        save.cards[i].name = unique[i]->name;
        for(int j = 0; j < game.drwSize; j++) {
            if((game.drwPile[j]->color == unique[i]->color) && (game.drwPile[j]->name == unique[i]->name)) {
                save.draw[i]++;
            }
        }
        for(int j = 0; j < game.handS; j++) {
            if((game.hand[j]->color == unique[i]->color) && (game.hand[j]->name == unique[i]->name)) {
                save.play[i]++;
            }
        }
        for(int j = 0; j < game.bHandS; j++) {
            if((game.bHand[j]->color == unique[i]->color) && (game.bHand[j]->name == unique[i]->name)) {
                save.bot[i]++;
            }
        }
    }
}
```

Card.h

```
/*
 * File:   Card.h
 * Author: Matthew Madrigal
 * Purpose:  Card Structure
 */

#ifndef CARD_H
#define CARD_H

struct Card{
    int color;
    int name;
};

#endif /* CARD_H */
```

Game.h

```
/*
 * File:   Game.h
 * Author: Matthew Madrigal
 * Purpose:  Game state structure
 */

#ifndef GAME_H
#define GAME_H

#include "Card.h"

struct Game{
    bool turn;      // True is for the player, false is for the computer
    int plScore;    // The score for the player
    int cpScore;    // The score for the computer
    Card *lastPl;   // The last played card
    Card *rest;     // The current restriction on what can be played
    int drwSize;    // Size of draw pile array
    int drwMax;
    Card **drwPile;  // The cards waiting to be drawn that way a game never has more than the max number according to the uno rules
    int handS;      // The size of the array of the cards the player currently has
    int handM;      // Max num of elements the array can hold right now
    Card **hand;     // The cards the player currently has
    int bHandS;     // The size of the array of the cards the computer currently has
    int bHandM;     // Max num of elements the array can hold right now
    Card **bHand;    // The cards the computer currently has
};

#endif /* GAME_H */
```

GameS.h

```
/*
 * File:   GameS.h
 * Author: Matthew Madrigal
 * Purpose:  This structure is used to save the game state to a file
 */

#ifndef GAMES_H
#define GAMES_H

#include "Card.h"

struct GameS{
    bool turn;      // True is for the player, false is for the computer
    int plScore;    // The score for the player
    int cpScore;    // The score for the computer
    int drwSize;    // Size of draw pile array
    int handS;      // The size of the array of the cards the player currently has
    int bHandS;     // The size of the array of the cards the computer currently has
    Card rest;      // The recent restriction on cards to play (useful for wild cards)
    Card lastPl;    // The last played card
    Card cards[54]; // The list of unique cards
    int draw[54];   // The amount of each unique card in the draw pile (index is the
same as the index of the unique card in cards)
    int play[54];   // The amount of each unique card in the player hand (index is the
same as the index of the unique card in cards
    int bot[54];    // The amount of each unique card in the computer hand (index is
the same as the index of the unique card in cards
};

#endif /* Game_S_H */
```