جامـعـة نيويورك أبـوظـبي

NYU ABU DHABI

Computer Programming for Engineers

ENGR-UH 1000

Project proposal : the Moroccan street game “Sota”

Group 6

Spring, 2025

This report is entirely our own work, and we have kept a copy for our own records. We are aware of the University’s policies on cheating, plagiarism, and the resulting consequences of their breach.

Submitted by:

Name	**Net ID**	**Signature**
Alae Hajjaj	ah7020	
Hafsa Ait Gaouzguit	Ha2972	
Ghadi Hanna	gjh7959	

Introduction:

The Moroccan card game ***Sota*** known locally as *Jaban Garae* is a cherished cultural tradition, blending strategy and social interaction through its unique 40-card Spanish/Moroccan deck. In this project, our team bridged tradition and technology by translating ***Sota*** into an interactive C++ application. Our goal was twofold: to preserve the game's cultural legacy in a digital format and to demonstrate mastery of advanced programming concepts. We meticulously recreated **Sota**'s fast-paced mechanics, including card matching (suits or ranks), special card effects (Ace for skipping turns, Two for stacking penalties, and Seven for wildcard suit changes), and dynamic deck reshuffling.

Using object-oriented programming (**OOP**), we designed a class hierarchy `Cards, Player,` and `Game` to encapsulate gameplay logic, while **dynamic memory allocation** ensured efficient handling of decks and player hands. This approach not only honored the game's authenticity but also showcased core C++ principles emphasized in the course. By digitizing ***Sota***, we aimed to breathe new life into a cultural gem, proving how programming can transform traditional activities into accessible, modern experiences.

This project stands as a testament to both cultural preservation and technical rigor, merging the vibrancy of Moroccan heritage with the precision of computational problem-solving.

Project Development:

Software & Tools

- **Programming Language**: C++ (for OOP and dynamic memory management).
- **Libraries**:
  - `<iostream>`: Handles input/output operations (e.g., displaying cards, player interactions).
  - `<cstdlib>` and `<ctime>`: Used for randomization (shuffling the deck) and timing.
  - `<string>`: Manages player names and card suit labels.

Implementation Details

Dynamic Memory Allocation

Two key components use dynamic memory:

1. **`Cards` Class**:
   - **`int** deck`**: A 2D array dynamically allocated to represent the 40-card deck (4 suits × 10 cards each).

In cpp

```
deck = new int*[SUIT];
```

```
for (int i = 0; i < SUIT; ++i) {
   deck[i] = new int[MAXCARDS / SUIT];
}
```

- **Destructor**: Ensures memory is freed to prevent leaks:

In cpp

```
~Cards() {
   for (int i = 0; i < SUIT; ++i) delete[] deck[i];
   delete[] deck;
}
```

2. `Player` **Class**:
    - `int* hand`: Dynamically allocated array to store a player's cards.

In cpp

```
hand = new int[MAXCARDS];
```

- **Destructor**: Releases memory when a player object is destroyed:

In cpp

```
~Player() { delete[] hand; }
```

Object-Oriented Programming (OOP)

Three core classes structure the game:

1. `Cards` **Class**:
    - **Responsibilities**:
        - Initialize and shuffle the deck.
        - Draw cards from the deck.
    - **Key Methods**:
        - `shuffleDeck()`: Implements the **Fisher-Yates algorithm** to randomize card order.
        - `drawCard()`: Removes and returns the top card from the deck.
2. `Player` **Class**:
    - **Responsibilities**:
        - Manage a player's hand (drawing, displaying, and playing cards).
        - Validate moves based on game rules.
    - **Key Methods**:
        - `hasPlayableCard()`: Checks if a player can make a valid move.
        - `playCard()`: Handles card selection and updates the player's hand.
3. `Game` **Class**:
    - **Responsibilities**:
        - Control gameplay flow (turns, win conditions).

        - Apply special card effects (skip, draw two, wildcard suit change).
    - **Key Methods**:
        - `start()`: Initializes the game, deals cards, and runs the turn loop.
        - Handles `skipCount` and `drawTwoCount` logic for stacked penalties.

Algorithms & Key Functions

- Constants

```
DEFINE MAXCARDS ← 40
DEFINE SUIT ← 4
```

- Utility Functions

```
FUNCTION printLine()
   PRINT "----------------------------------------"
END FUNCTION

FUNCTION skipLines(n ← 3)
   FOR i FROM 0 TO n - 1
      PRINT NEWLINE
   END FOR
END FUNCTION

FUNCTION isValidMove(prevCard, newCard)
   RETURN (prevCard MOD 10 = newCard MOD 10) OR ((prevCard - 1) DIV 10 = (newCard - 1) DIV 10)
END FUNCTION

FUNCTION isValidMovePow(newCard, suit)
   RETURN (newCard MOD 10 = 7) OR (suit = (newCard - 1) DIV 10)
END FUNCTION

FUNCTION isExceptionCard(card)
   val ← card MOD 10
   RETURN val = 1 OR val = 2 OR val = 7
END FUNCTION

FUNCTION printCardName(k)
   suits ← ["Clubs", "Cups", "Swords", "Coins"]
   suit ← (k - 1) DIV 10
   val ← k MOD 10
```

```
  IF val = 0 THEN val ← 10
  IF val = 8 THEN PRINT "The Jack of " + suits[suit]
  ELSE IF val = 9 THEN PRINT "The Horse of " + suits[suit]
  ELSE IF val = 10 THEN PRINT "The King of " + suits[suit]
  ELSE PRINT "The " + val + " of " + suits[suit]
END FUNCTION

FUNCTION announceSuit(suit)
  suits ← ["Clubs", "Cups", "Swords", "Coins"]
  PRINT "The chosen suit is now: " + suits[suit]
END FUNCTION
```

- Class: Cards

```
CLASS Cards
  PRIVATE deck: 2D ARRAY OF INTEGER

  CONSTRUCTOR()
    deck ← 2D ARRAY SUIT x (MAXCARDS / SUIT)
    FOR i FROM 0 TO SUIT - 1
      FOR j FROM 0 TO (MAXCARDS / SUIT) - 1
        deck[i][j] ← i * 10 + j + 1
      END FOR
    END FOR
  END CONSTRUCTOR

  METHOD shuffleDeck()
    flat ← ARRAY OF MAXCARDS
    idx ← 0
    FOR i FROM 0 TO SUIT - 1
      FOR j FROM 0 TO (MAXCARDS / SUIT) - 1
        flat[idx] ← deck[i][j]
        idx ← idx + 1
      END FOR
    END FOR
    FOR i FROM MAXCARDS - 1 DOWNTO 1
      j ← RANDOM(0, i)
      SWAP flat[i], flat[j]
    END FOR
    idx ← 0
    FOR i FROM 0 TO SUIT - 1
      FOR j FROM 0 TO (MAXCARDS / SUIT) - 1
        deck[i][j] ← flat[idx]
        idx ← idx + 1
      END FOR
    END FOR
  END METHOD

  METHOD drawCard()
    FOR i FROM 0 TO SUIT - 1
      FOR j FROM 0 TO (MAXCARDS / SUIT) - 1
        IF deck[i][j] ≠ 0 THEN
          card ← deck[i][j]
          deck[i][j] ← 0
          RETURN card
        END IF
      END FOR
    END FOR
    RETURN -1
  END METHOD

  DESTRUCTOR()
    DELETE deck
  END DESTRUCTOR
END CLASS
```

- Class: Player

```
CLASS Player
  PUBLIC name: STRING
  hand: ARRAY OF INTEGER
  handSize: INTEGER

  CONSTRUCTOR()
    hand ← ARRAY OF MAXCARDS
    handSize ← 0
  END CONSTRUCTOR

  METHOD draw(deck: Cards)
    card ← deck.drawCard()
```

```
        IF card ≠ -1 THEN
            hand[handSize] ← card
            handSize ← handSize + 1
        END IF
    END METHOD

    METHOD showHand()
        FOR i FROM 0 TO handSize - 1
            PRINT (i + 1) + ") "
            CALL printCardName(hand[i])
        END FOR
    END METHOD

    METHOD hasPlayableCard(prev, suit)
        FOR i FROM 0 TO handSize - 1
            IF (suit = -1 AND isValidMove(prev, hand[i])) OR (suit ≠ -1 AND isValidMovePow(hand[i], suit))
                RETURN TRUE
        END FOR
        RETURN FALSE
    END METHOD

    METHOD playCard(prev, suit)
        LOOP
            PRINT "Choose card to play: "
            READ choice
            IF choice ≥ 1 AND choice ≤ handSize THEN
                c ← hand[choice - 1]
                IF (suit = -1 AND isValidMove(prev, c)) OR (suit ≠ -1 AND isValidMovePow(c, suit)) THEN
                    FOR i FROM choice - 1 TO handSize - 2
                        hand[i] ← hand[i + 1]
                    END FOR
                    handSize ← handSize - 1
                    RETURN c
                END IF
            END IF
            PRINT "Invalid move! Try again."
        END LOOP
```

```
  END METHOD

  DESTRUCTOR()
    DELETE hand
  END DESTRUCTOR
END CLASS
```

- Class: Game

```
CLASS Game
  PRIVATE players: ARRAY OF Player
  numPlayers: INTEGER
  deck: Cards
  topCard: INTEGER
  suitChange: INTEGER
  skipCount: INTEGER
  drawTwoCount: INTEGER

  CONSTRUCTOR(n)
    numPlayers ← n
    players ← ARRAY OF Player[n]
    deck ← NEW Cards
    suitChange ← -1
    skipCount ← 0
    drawTwoCount ← 0
  END CONSTRUCTOR

  METHOD start()
    deck.shuffleDeck()
    FOR i FROM 0 TO numPlayers - 1
      PRINT "Player " + (i+1) + " name: "
      READ players[i].name
      FOR j FROM 0 TO 4
        players[i].draw(deck)
      END FOR
    END FOR

    REPEAT
      topCard ← deck.drawCard()
    UNTIL NOT isExceptionCard(topCard)
```

```
PRINT "Start card: "
CALL printCardName(topCard)

turn ← 0
LOOP
   p ← players[turn]
   skipLines()
   printLine()
   IF skipCount > 0 THEN
      PRINT p.name + " is skipped!"
      skipCount ← skipCount - 1
      suitChange ← -1
   ELSE
      PRINT p.name + "'s turn. Top: "
      printCardName(topCard)
      p.showHand()
      IF drawTwoCount > 0 THEN
         PRINT "Draw " + drawTwoCount + " cards"
         FOR i FROM 0 TO drawTwoCount - 1
            p.draw(deck)
         END FOR
         drawTwoCount ← 0
      ELSE IF NOT p.hasPlayableCard(topCard, suitChange) THEN
         PRINT "No playable cards. Drawing..."
         p.draw(deck)
      ELSE
         played ← p.playCard(topCard, suitChange)
         topCard ← played
         val ← played MOD 10
         IF val = 1 THEN skipCount ← skipCount + 1; suitChange ← -1
         ELSE IF val = 2 THEN drawTwoCount ← drawTwoCount + 2; suitChange ← -1
         ELSE IF val = 7 THEN
            PRINT "You played a 7! Choose a suit:"
            PRINT "0) Clubs\n1) Cups\n2) Swords\n3) Coins"
            READ suitChange
            announceSuit(suitChange)
         ELSE
            suitChange ← -1
```

```
               END IF
               IF p.handSize = 0 THEN
                  PRINT p.name + " wins!"
                  BREAK LOOP
               END IF
            END IF
         END IF
         turn ← (turn + 1) MOD numPlayers
      END LOOP
   END METHOD

   DESTRUCTOR()
      DELETE players
      DELETE deck
   END DESTRUCTOR
END CLASS
```

- Main Program:

```
BEGIN MAIN
   SEED RANDOM WITH CURRENT TIME
   LOOP
      PRINT "MENU\n1) Play\n2) Help\n3) Exit"
      READ ch
      IF ch = 1 THEN
         PRINT "Players(2-4): "
         READ pc
         g ← NEW Game(pc)
         g.start()
      ELSE IF ch = 2 THEN
         showHelp()
      ELSE IF ch = 3 THEN
         BREAK LOOP
      ELSE
         PRINT "Invalid"
      END IF
   END LOOP
   RETURN 0
END MAIN
```

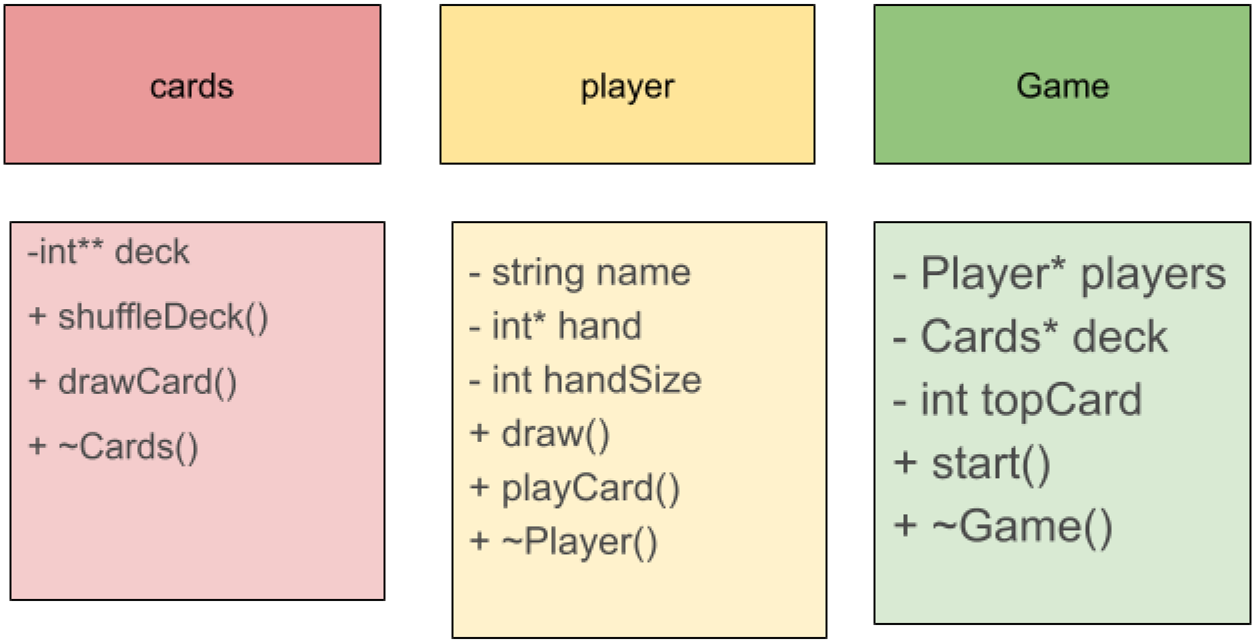
- Function: showHelp:

```
FUNCTION showHelp()
    PRINT "\nHELP:\n"
    PRINT "Match number or suit.\n"
    PRINT "7: Wildcard (choose suit)\n"
    PRINT "1: Skip next\n"
    PRINT "2: Draw two (stackable)\n"
    PRINT "Empty hand wins.\n\n"
END FUNCTION
```

Special Card Logic:

- **Ace (1)**: Skips the next player via `skipCount`.
- **Two (2)**: Forces the next player to draw 2 cards (stackable with `drawTwoCount += 2`).
- **Seven (7)**: Triggers `announceSuit()`, allowing players to dynamically change the active suit.

Class Diagram



cards

-int** deck
+ shuffleDeck()
+ drawCard()
+ ~Cards()

player

- string name
- int* hand
- int handSize
+ draw()
+ playCard()
+ ~Player()

Game

- Player* players
- Cards* deck
- int topCard
+ start()
+ ~Game()

Code Explanation with Snapshots

1. **Deck Initialization**:

```
Cards() {
    deck = new int*[SUIT];
    for (int i = 0; i < SUIT; ++i) {
        deck[i] = new int[MAXCARDS / SUIT];
        for (int j = 0; j < MAXCARDS / SUIT; ++j)
            deck[i][j] = i * 10 + j + 1;
    }
}
```

2. **Turn Cycle in `Game::start()`:**

```
while (true) {
    Player& p = players[turn];
    if (skipCount > 0) {
        skipCount--;
    } else {

        if (drawTwoCount > 0) {
            p.draw(deck);
        } else if (!p.hasPlayableCard(...)) {
            p.draw(deck);
        } else {
            int played = p.playCard(...);

        }
    }
    turn = (turn + 1) % numPlayers;
}
```

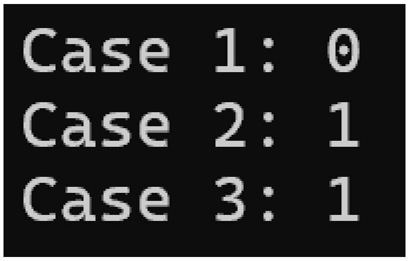
Results and Evaluation:

This section describes how you evaluated the functionalities of this software. Explanations of challenges, errors, and debugging can also be included in this section.

There were many help and class functions in the game that each controlled specific aspects of the game while also taking into account any specific case-by-case scenarios without disrupted the overall flow of the game. This includes functions that validate user input and store temporary cards in either the deck or players hands depending on the situation or round in the game.

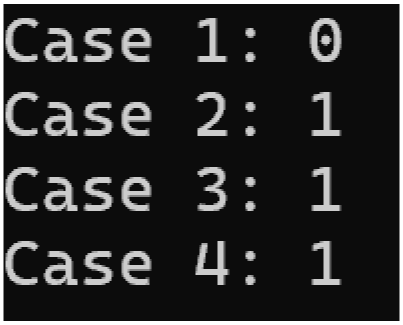
-> The helper functions that are defined outside the class functions were tested via random inputs to check if the desired output came from a predetermined input.

Function	Description	Input	Desired Output	Actual output
isValidMove	Checks to see if the card played matches the current suit or number	1) (13,2) 2) (13,15) 3) (13,33)	0 1 1	0 1 1
isValidMovePow	Checks to see if the card played matches the color determined by a ”7” or is a 7 of any suit	1) (13,0) 2) (36,3) 3) (7,1) 4) (17,2)	0 1 1 1	0 1 1 1
isExceptionCard	Checks to see if the top card is a special card	1) (10) 2) (11) 3) (32)	0 1 1	0 1 1
printCardName	Prints the suit and number of the card	1) (9) 2) (27) 3) (40)	The Horse of Clubs The 7 of Swords The King of Coins	The Horse of Clubs The 7 of Swords The King of Coins



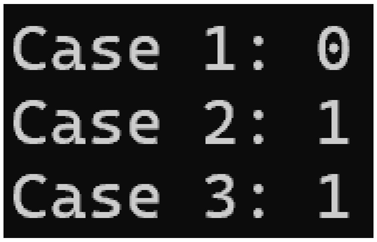
```
Case 1: 0
Case 2: 1
Case 3: 1
```

Figure 1: Code output for isValidMove



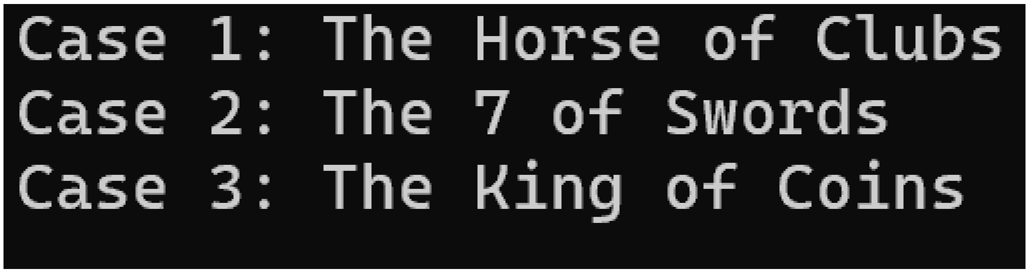
```
Case 1: 0
Case 2: 1
Case 3: 1
Case 4: 1
```

Figure 2: Code output for isValidMovePow



```
Case 1: 0
Case 2: 1
Case 3: 1
```

Figure 3: Code output for Exceptional Card

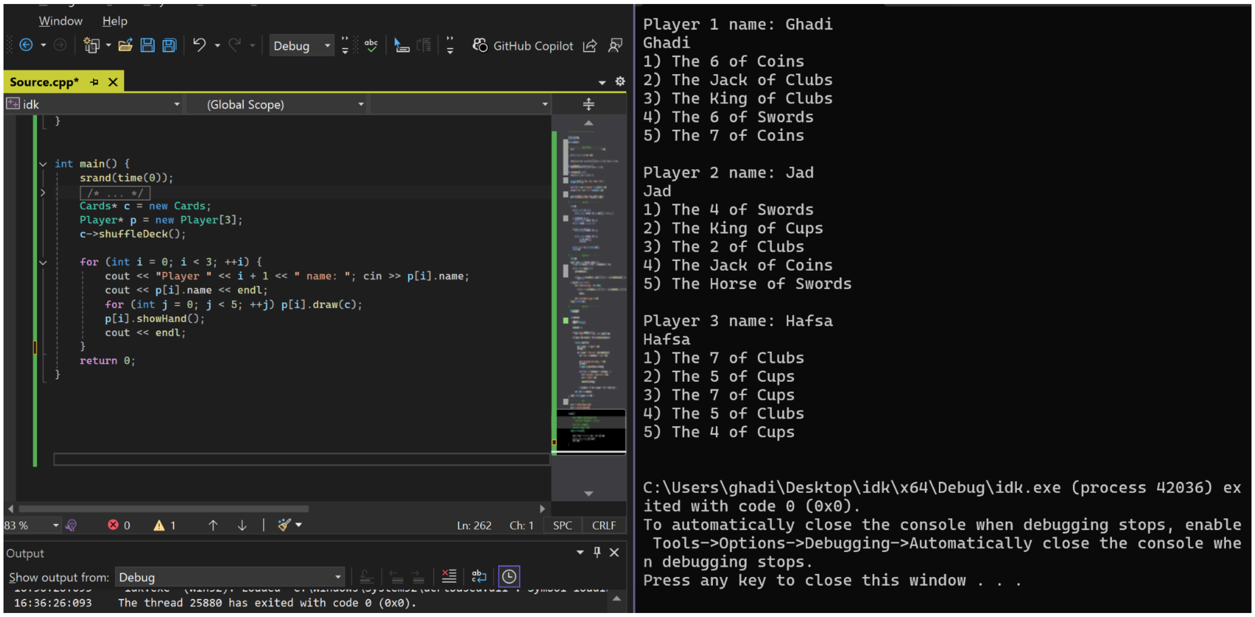


```
Case 1: The Horse of Clubs
Case 2: The 7 of Swords
Case 3: The King of Coins
```

Figure 4: Code output for printCardName

-> To test class functions, we took each respective function and made a class that uses these functions and tested it with main to see if the desired output was met.

- For instance, here we tested the drawcard function, set name function, and show hand function for the player while also testing the functionality of the shuffle function for the Cards class.
- The desired output is the name of each player printed out clearly for the user to see, each with unique sets of cards.



```cpp
}

int main() {
    srand(time(0));
    /* ... */
    Cards* c = new Cards;
    Player* p = new Player[3];
    c->shuffleDeck();

    for (int i = 0; i < 3; ++i) {
        cout << "Player " << i + 1 << " name: "; cin >> p[i].name;
        cout << p[i].name << endl;
        for (int j = 0; j < 5; ++j) p[i].draw(c);
        p[i].showHand();
        cout << endl;
    }
    return 0;
}
```

```
16:36:26:093    The thread 25880 has exited with code 0 (0x0).
```

```
Player 1 name: Ghadi
Ghadi
1) The 6 of Coins
2) The Jack of Clubs
3) The King of Clubs
4) The 6 of Swords
5) The 7 of Coins

Player 2 name: Jad
Jad
1) The 4 of Swords
2) The King of Cups
3) The 2 of Clubs
4) The Jack of Coins
5) The Horse of Swords

Player 3 name: Hafsa
Hafsa
1) The 7 of Clubs
2) The 5 of Cups
3) The 7 of Cups
4) The 5 of Clubs
5) The 4 of Cups


C:\Users\ghadi\Desktop\idk\x64\Debug\idk.exe (process 42036) ex
ited with code 0 (0x0).
To automatically close the console when debugging stops, enable
 Tools->Options->Debugging->Automatically close the console whe
n debugging stops.
Press any key to close this window . . .
```

- This shows that the functions shuffleDeck(), and draw() work as intended.

-> In the case of debugging, some instances in the testing phase of loops and other major functions in the code require specific “cout” statements in a certain space to confirm that the code is running correctly through the loop and identify where the loop discontinues or displays the wrong type of information. This method was also used to detect any abnormalities or issues with the output of both helper and class functions.

-> There were three main issues in implementing certain functions:

1) Card Management
2) Power Card Implementation
3) Setting all possible Rules and parameters

1) The card management issue was trying to save all the cards in the played deck, in the main deck, and in the player's hands, ensuring that no duplicate cards were played and that all the cards played or drawn are saved in the program. Dynamic memory allocation helped in storing permanent values in the deck or player’s hands, but with the constant size changes and different cards played every round, it was difficult to track all this progress to each dynamic array whenever it was created.

2) In essence, detecting power cards, no matter the number or suit, was simple; however, putting it in a loop, making sure that the output of the function of the desired power card is reflected in the roun,d was hard to track. An inherent issue was with the loop logic and bool statements not keeping track of previous cards and action cards executions correctly, as the cards constantly changed.

3) This was by far the most difficult part of the code because the rules that governed the normal game of Sota are based on case-by-case scenarios that were either tedious or not thought to be necessary. For instance, updating the same topcard after every round was hard to implement without accidentally accessing corrupted memory or accessing the wrong card since this takes place in a complex loop. Another issue was with the “7” wild card rules that made unnecessarily complex case-by-case scenarios in a very sensitive loop. Implementing rules that are usually taken care of in real life like shuffling the deck and knowing what card is played or drawn is intuitive but challenging to implement in code where it knows nothing about the deck or cards.

Conclusion and Future Work:

This project delivers a console-based card game that faithfully adapts the 40-card Moroccan/Spanish deck. Dynamic memory allocation manages both the deck and each player’s hand, while an object-oriented design comprising `Cards`, `Player`, and `Game` classes ensures clear separation of concerns. Special cards function as intended: aces stack skips, twos impose draw penalties, and sevens act as wildcards for suit changes. The console interface guides 2–4 players with concise prompts and real-time feedback.

Potential Future Developments

- **Graphical Interface:** Transition from console to a lightweight GUI (e.g. SDL or Qt) so cards can be selected visually.
- **Network Play:** Integrate basic TCP/IP communication to enable remote multiplayer sessions.
- **Automated Testing:** Employ a framework like Google Test for unit tests, improving reliability and easing maintenance.
- **AI Opponents:** Introduce rule-based CPU players to support single-player modes with adjustable difficulty.
- **Statistics and Persistence:** Record game outcomes and player statistics in a file or simple database, enabling leaderboards and progress tracking.

These enhancements would elevate the application from a functional prototype to a more polished, user-friendly experience suitable for wider distribution.

Reflection on Learning:

This project deepened understanding of dynamic memory management in C++, highlighting why proper allocation and deallocation is crucial to prevent leaks. Working with the deck and player hands on the heap reinforced mastery of pointers, new/delete, and the RAII principle. Building the Cards, Player, and Game classes improved skills in encapsulation, class cohesion, and designing clear interfaces.

Teamwork proved essential for keeping the game logic consistent. Whenever a rule slipped through, whether stacking skips or reshuffling the deck, debugging alone took too long. By reviewing each other's code, pairing up on tricky sections, and holding short "rule-check" meetings, the group caught edge cases faster and maintained alignment with the specification. This collaborative approach turned a tangled rule set into a reliable, playable game.