**Project 1**

**Solitaire**

**CIS-17C-40369**

**Name: Enrique Villavisencio**

**Date: 4/29/20**

**Introduction**

The game that I decided to code was the game of Solitaire. I chose this game because it is one of the few card games I am familiar with and I have always enjoyed it.

Project Size: About 800 lines

Project Location:

Game Rules:

The game of Solitaire, specifically the Klondike version, is played with a full deck of 52 playing cards which includes the ranks Ace through King for the 4 suites which are Spades, Diamond, Hearts, Clubs. There are four different types of piles in the game. These piles include the Tableau which includes 7 piles, the Foundation which includes 4 piles, the Stock pile and the Waste pile.

At the start of the game the deck shuffled and then distributed through the Tableau piles. You put the first card facing up and then put one card in the rest of the Tableau piles facing down until you reach the end. From there you skip the pile that the face up card and repeat this until the piles have number of cards equal to the pile number with the front card facing up. The rest of the cards you place them in a stack facing down in the Stack pile.

From here the game starts and you start trying to arrange the cards by suite from Ace at the bottom to King at the top in order in the Foundation piles. To do this you have to arrange the face up cards from highest to lowest in the Tableau in an attempt to turn over all the face down cards. Suite doesn’t matter when arranging the cards in the Tableau and instead they need to alternate between the two colors.

When you can’t make any more plays based on the cards you have in the Tableau you can pull a card from the Stock pile and put it face up in the waste pile. You can do this as many times as you want as long as there are cards and only the top card in the waste pile can be played at a time. When the stock pile is empty you can move from cards back from the waste pile and cycle through the cards again.

You repeat all this until you either get all the cards into the foundation piles or there are no more plays left at which point the game ends.

Approach to Development:

My approach to this project was creating base classes for each of the components in the game. From there I started adding things to add more functionality to them. As I was working on adding everything that I needed I didn’t worry about making the code very readable. I would comment out sections I would end up trying different ways to approach. In the end the code was full of random junk I no longer was using. At this point I went through deleting all this commented out code and making the entire thing more readable. As I was cleaning up the code, I found more efficient ways to write certain parts of the code and went through the entire thing changing parts around and moving them to places where they made more sense.

Description of Code:

My code is separated into 7 classes which include a class for Card, Deck, Tableau, Stock, Waste, Foundation, and Game.

|  |  |  |
| --- | --- | --- |
| **Class Card** |  |  |
| **Variable Type** | **Variable Name** | **Description** |
| int | value | The card’s value (1-13) |
| string | rank | The card’s rank (A – K) |
| char | suit | The card’s suit (S, C, D, H) |
| char | color | The card’s color (R, B) |

|  |  |
| --- | --- |
| **Class Card** |  |
| **Functions** | **Description** |
| Card(int, string, char, char) | Card class constructor to initialize the variables. |
| int getValue() | Return the value variable. |
| string getRank() | Returns the rank variable. |

|  |  |  |
| --- | --- | --- |
| **Class Deck** |  |  |
| **Variable Type** | **Variable Name** | **Description** |
| const string | RANK[13] | All the different card ranks |
| const char | SUIT[4] | All the different card suits |
| const char | COLOR[2] | The two different ca |
| string | valRank[52] | Value and rank combinations for all 52 cards |
| pair<char, char> | SuitColor[4] | Pairs the suit and Color together for the 4 different combinations |
| deque<int>\* | deckIndx | The container for the deck you will be playing . |
| map<string, int> | inputRef | Holds the valRank and an index for the deck of cards |
| map<int, Card> | deckRef | The deck of cards used as reference to look up the card’s information |

|  |  |
| --- | --- |
| **Class Deck** |  |
| **Functions** | **Description** |
| Deck() | Deck class constructor to initialize the variables. |
| card getCard(int) | Takes in an index value and returns a card |
| deque<int>\* getDeck() | Returns the deckIndx container to access it’s functions in another class |
| void displayCard(int); | Takes in an index value and display the correct card. |
| void displayCard(pair<int, bool>) | Takes in an int and bool pair and displays the correct card up or down based on bool. |
| bool validateCardInput(string) | Checks if the card the player inputs is a valid card. |
| Int getCardIndxFromInput(string) | Takes in the card the player input to move and returns the correct index for it. |

|  |  |  |
| --- | --- | --- |
| **Class Foundation** |  |  |
| **Variable Type** | **Variable Name** | **Description** |
| Stack<int>\* | foundation | The container for the Foundation piles. |

|  |  |
| --- | --- |
| **Class Foundation** |  |
| **Functions** | **Description** |
| Foundation() | Foundation class constructor to initialize the stack. |
| ~Foundation() | Class destructor to delete the stack allocated memory. |
| Stack<int>\* getFoundation() | Returns the Foundation pile container. |

|  |  |  |
| --- | --- | --- |
| **Class Stock** |  |  |
| **Variable Type** | **Variable Name** | **Description** |
| Stack<int>\* | stock | The container for the stock pile. |

|  |  |
| --- | --- |
| **Class Stock** |  |
| **Functions** | **Description** |
| Stock() | Tableau class constructor to initialize the stack. |
| ~Stock() | Class destructor to delete the stack allocated memory. |
| stack<int>\* getStock() | Returns the Stock container. |

|  |  |  |
| --- | --- | --- |
| **Class Waste** |  |  |
| **Variable Type** | **Variable Name** | **Description** |
| list<int>\* | waste | The container for the waste pile. |

|  |  |
| --- | --- |
| **Class Waste** |  |
| **Functions** | **Description** |
| Waste() | Waste class constructor to initialize the list. |
| ~Waste() | Class destructor to delete the stack allocated memory. |
| list<int>\* getWaste() | Returns the Waste container. |

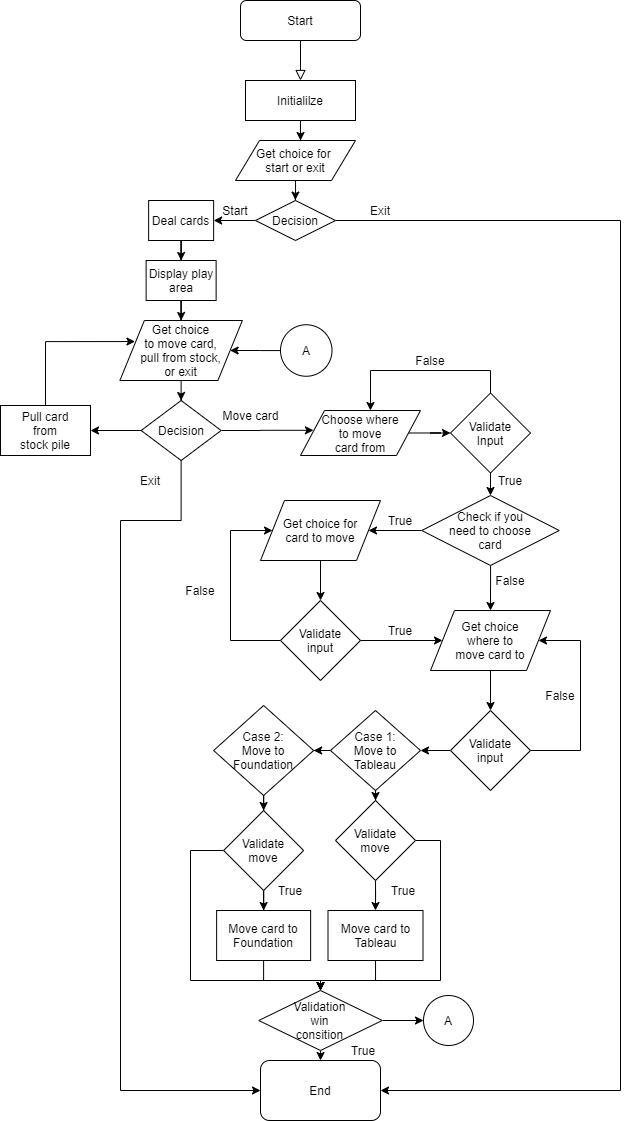
|  |  |  |
| --- | --- | --- |
| **Class Tableau** |  |  |
| **Variable Type** | **Variable Name** | **Description** |
| int | startCards | Number of cards the pile can hold at the start of the game when distributing the cards. |
| list<pair<int, bool>>\* | tableau | The container for the Tableau piles. |

|  |  |
| --- | --- |
| **Class Tableau** |  |
| **Functions** | **Description** |
| Tableau() | Tableau class constructor to initialize the variables. |
| ~Tableau() | Tableau class destructor to delete the list. |
| list<pair<int, bool>>\* getTableau() | Returns the Tableau container. |
| int startCards() | Returns the startCards value. |
| ist<pair<int, bool>> removeCards() | Determines how many cards are going to be moved, puts them in a temp, pops them out of the list and returns the cards. |
| bool only1Up() | Checks how many cards are facing up in the Tableau pile and returns true if there is only one card facing up and false if there is more. |

|  |  |  |
| --- | --- | --- |
| **Class Game** |  |  |
| **Variable Type** | **Variable Name** | **Description** |
| Deck\* | deck | Deck class object. |
| Tableau\* | tableau[7] | Tableau class object array. |
| Foundation\* | foundation[4] | Foundation class object array. |
| Stock\* | stock | Stock class object. |
| Waste\* | waste | Waste class object. |

|  |  |
| --- | --- |
| **Class Game** |  |
| **Functions** | **Description** |
| Game() | Class constructor that holds the majority of the code for this class, including the game loop. |
| ~Game() | Game class destructor to delete all the class objects from memory. |
| void init() | Function to initialize the class objects once the game loop starts. |
| void deal() | Distributes the cards. |
| void displayTable() | Displays the entire game area. |
| void stockPile() | Function to get one card from the stock pile and put it face up on the waste pile to be able to use. |
| list<pair<int, bool>> fromCards(string) | Function to either get cards to move from the waste or tableau pile. |
| void toTableau(string, int, int) | Function that starts the card move to a tableau pile. |
| void toFoundation(string, int, int) | Function that starts the card move to a foundation pile. |
| bool validateCard(string, string) | Validate if the specific card exists in the tableau pile. |
| bool validateToT(string, string) | Validate if the card can move to the Tableau pile. |
| bool validateToF(string, string) | Validate if the card can move to the Foundation pile. |

Flow Chart



Checkoff Sheet:

1.Container classes

1. Sequences

1. List

* I used the list container for my waste class because it required access to the top 3 cards in that pile to display onto the screen and with a list you can easily access the top elements using iterators.

2. Deque

* I also used the deque container for my deck that I played with even though it wasn’t a requirement because I needed a container that I could use random\_shuffle on.

2. Associative Containers

2. Map

* I used the map container for my card deck that I used as a reference because it allowed me to access the information easier and allowed me to validate one of the inputs from the player.

3. Container Adaptors

1.Stack

* I used the stack container for my stock and foundation classes. The stock uses a last in first out method so a stack was perfect. For foundation class you only ever put in cards and only ever need to access the top card to compare so stack was perfect.

2. Iterators

1. Concepts

6. Bidirectional Iterator

* This is the main iterator that I used in my code to iterate through my lists, deques, and maps
* I also dereferenced this iterator to get to assign the value to other variables.

3. Algorithms

1. Non-mutating algorithm

2.Find

* I used the find algorithm in my code with my map container to search for a specific card based on the string input from the player and return the index and validate the input from the player.

2. Mutating algorithms

7.Random\_Shuffle

* I used the random\_shuffle algorithm in my code with my deque container to shuffle the deck before distributing out the cards.

3. Organization

* I didn’t end up using any organization algorithms because the game of Solitaire doesn’t use any. Instead it relies on the player to organize the piles.

**Code:**

Card.h

class Card

{

private:

int value;

string rank;

char suit;

char color;

public:

Card(int, string, char, char);

int getValue() { return value; }

string getRank() { return rank; }

char getSuit() { return suit; }

char getColor() { return color; }

};

Card::Card(int v, string r, char s, char c)

{

value = v;

rank = r;

suit = s;

color = c;

}

**Deck.h**

#include "Card.h"

using namespace std;

class Deck

{

private:

const string RANK[13] = { "A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K" };

const char SUIT[4] = { 'S', 'D', 'C', 'H' };

const char COLOR[2] = { 'B', 'R' };

string valRank[52];

pair<char, char> suitColor[4];

deque<int>\* deckIndx;

map<string, int> inputRef;

map<int, Card> deckRef;

public:

Deck();

Card getCard(int);

deque<int>\* getDeck() { return deckIndx; }

void displayCard(int);

void displayCard(pair<int, bool>);

bool validateCardInput(string);

int getCardIndxFromInput(string card) { return inputRef.find(card)->second; }

};

Deck::Deck()

{

//Setting up CARD[52] for player card input reference

for (int i = 0; i < 4; i++)

{

for (int j = 0; j < 13; j++)

{

valRank[(13 \* i) + j] = RANK[j] + SUIT[i];

inputRef.insert(pair<string, char>(valRank[(13 \* i) + j], (13 \* i) + j));

}

}

//Pair the suits with the correct colors

for (int i = 0; i < 4; i++)

{

suitColor[i] = pair<char, char>('1', '2');

if (SUIT[i] == 'C' || SUIT[i] == 'S')

suitColor[i] = make\_pair(SUIT[i], COLOR[0]);

else

suitColor[i] = make\_pair(SUIT[i], COLOR[1]);

}

//Create Deck Index using Deque

deckIndx = new deque<int>;

for (int i = 0; i < 52; i++)

deckIndx->push\_back(i);

//Create the deck of card reference

deque<int>::iterator itr = deckIndx->begin();

for (int i = 0; i < 4; i++)

{

for (int j = 0; j < 13; j++)

{

Card\* card = new Card(j, RANK[j], suitColor[i].first, suitColor[i].second);

deckRef.insert(make\_pair(\*itr, \*card));

itr++;

}

}

}

Card Deck::getCard(int n)

{

map<int, Card>::iterator itr = deckRef.find(n);

return itr->second;

}

void Deck::displayCard(int n)

{

map<int, Card>::iterator itr = deckRef.find(n);

cout << "[" << itr->second.getRank() << ":" << itr->second.getSuit() << ":" << itr->second.getColor() << "]";

}

//Prints out the specific card

void Deck::displayCard(pair<int, bool> p)

{

if (p.second == false)

cout << "[ ? ]";

else

{

map<int, Card>::iterator itr = deckRef.find(p.first);

cout << "[" << itr->second.getRank() << ":" << itr->second.getSuit() << ":" << itr->second.getColor() << "]";

}

}

bool Deck::validateCardInput(string card)

{

for (int i = 0; i < 52; i++)

{

if (card == valRank[i])

return true;

}

return false;

}

**Tableau.h**

class Tableau

{

private:

int startCards;

list<pair<int, bool>>\* tableau;

public:

Tableau(int);

~Tableau();

list<pair<int, bool>>\* getTableau() { return tableau; }

int getStartCards() { return startCards; }

list<pair<int, bool>> removeCards(int);

bool only1Up();

};

Tableau::Tableau(int n)

{

tableau = new list<pair<int, bool>>;

startCards = n;

}

Tableau::~Tableau()

{

delete tableau;

}

list<pair<int, bool>> Tableau::removeCards(int n)

{

//Create a temp list to hold the cards that are removed

list<pair<int, bool>> temp;

//Move cards from the tableau pile to the temp list to return

while (!tableau->empty() && tableau->front().second == true)

{

temp.push\_front(tableau->front());

if (tableau->front().first == n)

{

tableau->pop\_front();

if (tableau->empty())

return temp;

else if (tableau->front().second == false)

tableau->front().second = true;

break;

}

else

tableau->pop\_front();

}

return temp;

}

bool Tableau::only1Up()

{

int count = 0;

for (list<pair<int, bool>>::iterator itr = tableau->begin(); itr != tableau->end(); itr++)

{

if (itr->second == true)

count++;

}

if (count <= 1)

return true;

else

return false;

}

**Stock.h**

class Stock

{

private:

stack<int>\* stock;

public:

Stock();

~Stock() { delete stock; }

stack<int>\* getStock() { return stock; }

};

Stock::Stock()

{

stock = new stack<int>;

}

**Waste.h**

class Waste

{

private:

list<int>\* waste;

public:

Waste();

~Waste() { delete waste; }

list<int>\* getWaste() { return waste; }

};

Waste::Waste()

{

waste = new list<int>;

}

**Foundation.h**

class Foundation

{

private:

stack<int>\* foundation;

public:

Foundation();

~Foundation() { delete foundation; }

stack<int>\* getFoundation() { return foundation; }

};

Foundation::Foundation()

{

foundation = new stack<int>;

}

**Game.h**

#include "Deck.h"

#include "Tableau.h"

#include "Foundation.h"

#include "Stock.h"

#include "Waste.h"

class Game

{

private:

Deck\* deck;

Tableau\* tableau[7];

Foundation\* foundation[4];

Stock\* stock;

Waste\* waste;

public:

Game();

~Game();

void init();

void deal();

void displayTable();

void stockPile();

list<pair<int, bool>> fromCards(string, int);

void toTableau(string, int, int);

void toFoundation(string, int, int);

bool validateCard(string, string);

bool validateToT(Card, int, bool);

bool validateToF(Card, int, bool);

};

Game::Game()

{

int choice;

cout << "\t\t\t\t\tSolitaire" << endl;

cout << "============================================================================================================" << endl;

cout << "1. Start \t 2. Exit" << endl;

cout << "Enter choice: ";

cin >> choice;

cin.ignore();

switch (choice)

{

case 1:

{

system("clear");

init();

//Shuffle the deck using random\_shuffle

random\_shuffle(deck->getDeck()->begin(), deck->getDeck()->end());

deal();

//Start the main game loop and continue looping until end equals true

bool end = false;

while (!end)

{

system("clear");

displayTable();

int choice = 0;

bool isValid = false;

while (!isValid)

{

choice = 0;

cout << "1. Move Card 2. Stock Pile 3. Quit" << endl;

cout << " Enter choice: ";

cin >> choice;

if (cin.fail())

{

cout << "Input is invalid. Try aain." << endl;

cin.clear();

cin.ignore();

}

else

isValid = true;

}

//Start the game or exit depending on the choice you made

switch (choice)

{

case 1:

{

string from, to, card;

bool chooseCard = false;

int fromIndx, cardIndx;

cin.ignore();

isValid = false;

//Loop to repeat "Choose where to move from:" if input is invalid

while (!isValid)

{

cout << "Tableau: T1, T2, T3, T4, T5, T6, T7 Waste: W" << endl;

cout << " Choose where to move from:";

getline(cin, from);

//Validate input

if (from == "W")

{

if (waste->getWaste()->empty())

cout << "Waste Pile is empty." << endl;

else

isValid = true;

}

else

{

if (from.size() > 2 || from.at(0) != 'T' || isalpha(from.at(1)))

cout << "Invalid input." << endl;

else

{

fromIndx = (from.at(1) - '0') - 1;

//Check if the tableau index range is within range

if (fromIndx >= 0 && fromIndx <= 6)

{

if (tableau[fromIndx]->getTableau()->empty())

cout << "Tableau " << fromIndx << " is empty." << endl;

else

isValid = true;

}

else

cout << "Invalid input." << endl;

}

}

}

//Determine if you need to choose a card to move

if (from != "W")

{

if (!(tableau[fromIndx]->only1Up()))

{

isValid = false;

//Loop to repeat if card input is invalid or the card doesn't exist in the pile

while (!isValid)

{

cout << " Choose the card to move:(Ex: AS for Ace of Spades): ";

getline(cin, card);

if (!(deck->validateCardInput(card)))

cout << "Invalid input, try again." << endl;

else

{

if (validateCard(from, card))

{

cardIndx = deck->getCardIndxFromInput(card);

isValid = true;

}

else

cout << "That card doesn't exist in that pile. Try again." << endl;

}

}

}

else

cardIndx = tableau[fromIndx]->getTableau()->front().first;

}

else

cardIndx = waste->getWaste()->front();

//Choose where to move the card to

isValid = false;

int toIndx;

while (!isValid)

{

//Display choices

cout << "Tableau: T1, T2, T3, T4, T5, T6, T7 Foundation: F1, F2, F3, F4" << endl;

cout << " Where do you want to move to:";

getline(cin, to);

//Validate the input

//Check if the from location is the same as the to location

if (from == to)

cout << "You can't move a card from the same place." << endl;

else

{

//Check the first character in the string to see if it matches the cases

switch (to.at(0))

{

case 'T':

//Get the index for the container we are moving the card to

toIndx = (to.at(1) - '0') - 1;

//Check if the tableau index is within 0 - 6

if (toIndx >= 0 && toIndx <= 6)

isValid = true;

break;

case 'F':

//Get the index for the container we are moving the card to

toIndx = (to.at(1) - '0') - 1;

//Check if the foundation index is within 0 - 3

if (toIndx >= 0 && toIndx <= 3)

isValid = true;

break;

default:

break;

}

if (!isValid)

cout << "Invalid input." << endl;

}

}

//Start the process to move cards from one pile to another

switch (to.at(0))

{

case 'T':

toTableau(from, cardIndx, toIndx);

break;

case 'F':

toFoundation(from, cardIndx, toIndx);

break;

default:

break;

}

//Winning Condition

for (int i = 0; i < 7; i++)

{

if (!tableau[i]->getTableau()->empty())

break;

if (i == 6)

end = true;

}

break;

}

case 2:

stockPile();

break;

case 3:

exit(0);

break;

default:

break;

}

}

}

break;

case 2:

exit(0);

break;

default:

break;

}

}

Game::~Game()

{

delete deck;

for (int i = 0; i < 7; i++)

delete tableau[i];

for(int i = 0; i < 4; i++)

delete foundation[i];

delete stock;

delete waste;

}

void Game::init()

{

//Create the deck for cards

deck = new Deck;

//Create the 7 Tableau piles and assign the max cards they can hold at the start when the cards are being dealt

for (int i = 0; i < 7; i++)

tableau[i] = new Tableau(i + 1);

//Initialize the 4 foundation piles where you will build up whole sequences based on suit

for (int i = 0; i < 4; i++)

foundation[i] = new Foundation;

//Initialize the stock pile where the extra cards will sit

stock = new Stock;

//Initialize the waste pile

waste = new Waste;

}

void Game::deal()

{

//Deal out the cards from the deck into the 7 tableaus

for (int i = 0; i < 7; i++)

{

for (int j = 0 + i; j < 7; j++)

{

//Copy the top card from the deck to the current tableau

tableau[j]->getTableau()->push\_front(pair<int, bool>(deck->getDeck()->front(), false));

if (tableau[j]->getTableau()->size() == tableau[j]->getStartCards())

tableau[j]->getTableau()->front().second = true;

deck->getDeck()->pop\_front();

}

}

////Put the rest of the cards into the stock pile

for (int i = 0; i < deck->getDeck()->size(); i++)

{

stock->getStock()->push(deck->getDeck()->front());

deck->getDeck()->pop\_front();

}

}

void Game::displayTable()

{

//Game Title and helpful info

cout << "\t\t\t\t\tSolitaire" << endl;

cout << "============================================================================================================" << endl;

cout << "S = Spades, D = Diamonds, C = Clubs, H = Hearts, A = Ace, J = Jack, Q = Queen, K = King, B = Black, R = Red" << endl;

//Display for the foundation info

cout << "============================================================================================================" << endl;

cout << "Foundations:" << endl;

for (int i = 0; i < 4; i++)

{

cout << i + 1 << ": ";

//Check whether the foundation is empty to determine whether to display empty or a card

if (foundation[i]->getFoundation()->empty())

cout << "[Empty]";

else

deck->displayCard(foundation[i]->getFoundation()->top());

cout << "\t";

}

cout << endl;

//Display for the Stock and Waste piles

cout << "============================================================================================================" << endl;

cout << "Stock Pile: ";

if (!stock->getStock()->empty())

cout << "[ ? ] " << endl;

else

cout << "Empty." << endl;

cout << "Waste Pile: ";

if (waste->getWaste()->empty())

cout << "[Empty]";

else

{

list<int>::iterator itr = waste->getWaste()->begin();

list<int> temp;

for (int i = 0; i < 4; i++, i++)

{

if (itr != waste->getWaste()->end())

{

temp.push\_front(\*itr++);

}

}

do

{

deck->displayCard(temp.front());

temp.pop\_front();

} while (!temp.empty());

}

cout << endl;

cout << "============================================================================================================" << endl;

cout << "Tableaus:" << endl;

for (int i = 0; i < 7; i++)

{

list<pair<int, bool>> temp = \*tableau[i]->getTableau();

temp.reverse();

cout << i + 1 << ": ";

for (list<pair<int, bool>>::iterator itr = temp.begin(); itr != temp.end(); itr++)

deck->displayCard(\*itr);

cout << endl;

}

cout << "============================================================================================================" << endl;

}

void Game::stockPile()

{

if (stock->getStock()->empty())

{

while (waste->getWaste()->size() > 0)

{

stock->getStock()->push(waste->getWaste()->front());

waste->getWaste()->pop\_front();

}

}

else

{

waste->getWaste()->push\_front(stock->getStock()->top());

stock->getStock()->pop();

}

}

void Game::toTableau(string from, int cardIndx, int toIndx)

{

bool isValid = false;

Card fromCard = deck->getCard(cardIndx);

if (validateToT(fromCard, toIndx, tableau[toIndx]->getTableau()->empty()))

{

list<pair<int, bool>> temp = fromCards(from, cardIndx);

for (list<pair<int, bool>>::iterator itr = temp.begin(); itr != temp.end(); itr++)

cout << itr->first << " ";

cout << endl;

do

{

tableau[toIndx]->getTableau()->push\_front(temp.front());

temp.pop\_front();

} while (!temp.empty());

}

}

void Game::toFoundation(string from, int cardIndx, int toIndx)

{

Card fromCard = deck->getCard(cardIndx);

if (validateToF(fromCard, toIndx, foundation[toIndx]->getFoundation()->empty()))

{

if (from == "W")

{

foundation[toIndx]->getFoundation()->push(waste->getWaste()->front());

waste->getWaste()->pop\_front();

}

else

{

list<pair<int, bool>> temp = fromCards(from, cardIndx);

foundation[toIndx]->getFoundation()->push(temp.front().first);

}

}

}

list<pair<int, bool>> Game::fromCards(string from, int cardIndx)

{

list<pair<int, bool>> temp;

if (from == "W")

{

temp.push\_front(pair<int, bool>(waste->getWaste()->front(), true));

waste->getWaste()->pop\_front();

return temp;

}

else

{

int i = (from.at(1) - '0') - 1;

//Tableau test for removing cards

list<pair<int, bool>> temp = tableau[i]->removeCards(cardIndx);

return temp;

}

}

//Validate if the card exists in the pile

bool Game::validateCard(string from, string card)

{

int i = (from.at(1) - '0') - 1;

for (list<pair<int, bool>>::iterator itr = tableau[i]->getTableau()->begin(); itr != tableau[i]->getTableau()->end(); itr++)

{

int indx = deck->getCardIndxFromInput(card);

if (itr->first == indx)

return true;

}

}

bool Game::validateToT(Card fromCard, int toIndx, bool empty)

{

if (empty)

{

//Check if the rank of the card we are moving is a king

if (fromCard.getRank() == "K")

return true;

}

//If the top card in the tableau you are moving to has a value of 1 it returns false because you can't put a card above a card of value 1 in the tableau

if (fromCard.getValue() == 1)

return false;

Card toCard = deck->getCard(tableau[toIndx]->getTableau()->front().first);

//Checks if the card being moved is 1 value less than the top card in the tableau we are moving it to

if (fromCard.getValue() == toCard.getValue() - 1 && fromCard.getColor() != toCard.getColor())

return true;

else

return false;

}

bool Game::validateToF(Card fromCard, int toIndx, bool empty)

{

if (empty)

{

//Check if the rank of the card you are moving is an Ace

if (fromCard.getRank() == "A")

return true;

}

Card toCard = deck->getCard(foundation[toIndx]->getFoundation()->top());

if (fromCard.getSuit() != toCard.getSuit())

return false;

}