Programming Assignment # 3 (part II) Solitaire	
Release Date	**17-Dec-2020 (Wednesday)**
Due Date	**22 -Dec-2020 (Tuesday) 11:59 PM**
Submission Folder	**On Google Classroom**
Total Marks	**TBA**

Solitaire Display

Object Oriented Programming Exercise

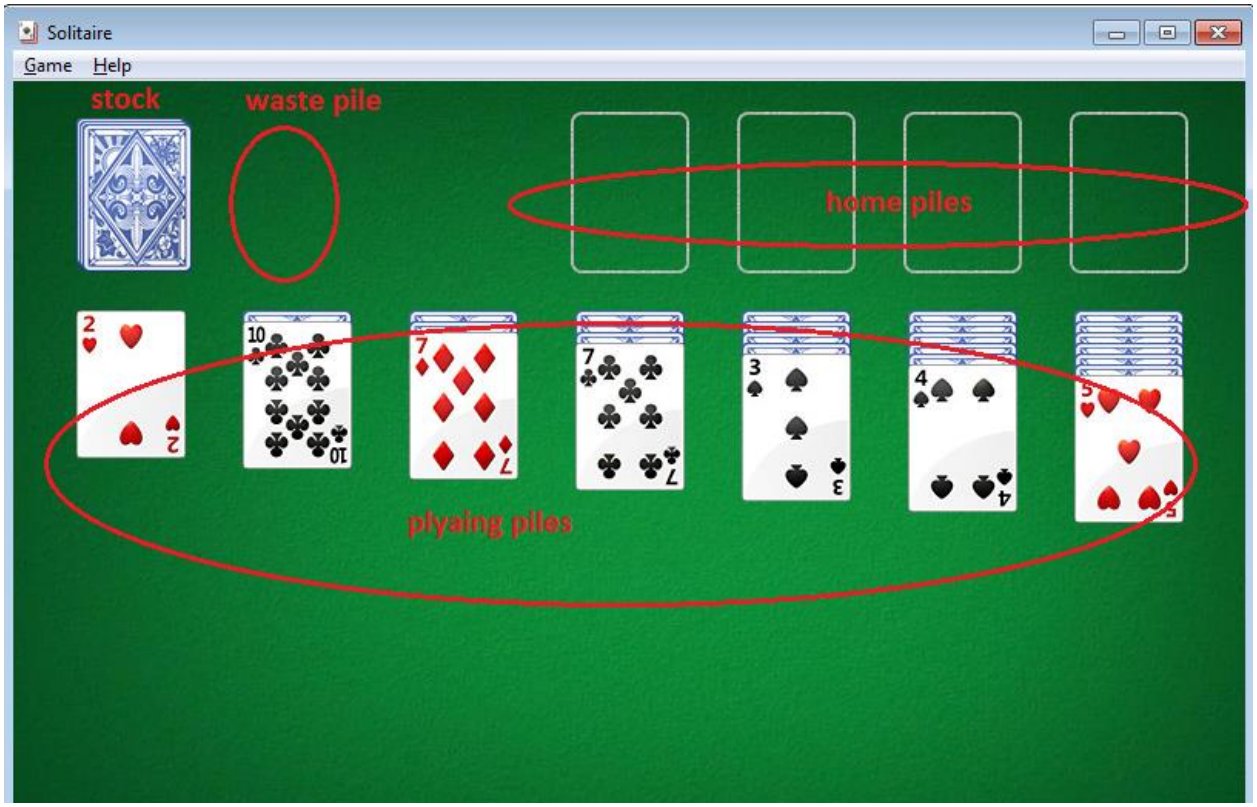
This part of homework-4 covers game display only. In part III you'll be covering the entire game!

Up till now you made simple structures of **PlayingCard**, **PileofCards** and **Deck**. Now using these classes we can build any interesting card game. You will develop a solitaire game that you can find in Microsoft Windows Games as well. Before you start doing coding you must understand the rules and terms in the game!

Rules:

To begin the game, 28 cards are dealt into 7 **playing piles**. The leftmost pile has 1 card, the next has two cards, and so forth up to 7 cards in the rightmost pile. Only the uppermost card of each of the 7 piles is turned face up. The cards are dealt left to right dealing and top card in each pile during dealing is turned face up.

There is another pile called **stock** to which remaining of the cards go. There are five more piles, 1 **waste** pile and 4 **home** piles whose purpose would be clear as you read further. See the image below shown for Windows Solitaire game telling different piles by labeling them.



From the top of the pile called **stock**, cards can be turned up one by one and placed on **waste** pile. You may always play cards off the top of the waste pile, but only one at a time.

On the top-most face up card of each **playing pile** you may build in descending sequences red on black or black on red. For example, on the 9 of spades you may place either the 8 of diamonds or the 8 of hearts. All face up cards on a playing pile can be moved as a unit and may be placed on another **playing pile** according to the bottommost face up card. For example, the 7 of clubs on the 8 of hearts may be moved as a unit onto the 9 of clubs or the 9 of spades.

In a **playing pile** whenever a face down card is uncovered, it is turned face up. If one pile is removed completely, a faceup King may be moved from another **playing pile** (together with all cards above it) or from the top of the **waste** pile.

There are four **home** piles, one for each suit, and the object of the game is to get as many cards as possible into the **home** piles. Each time an Ace appears at the top of a **playing pile** or the top of the **waste** pile it is moved into the appropriate **home** pile. Cards are added to the output piles in sequence, the suit for each pile being determined by the Ace on the bottom.

Note:

You must follow the instructions below thoroughly to complete the task successfully.

Console Functions:

Replace existing "ConsolFunctions.h" and "ConsolFunctions.cpp" files with the new ones provided. In case of not doing so, you'll get errors at later stage that would be hard to resolve. Also a new function `void ConsoleFunctions::clearConsole();` is added. Calling it will erase the entire display from console screen.

Update PlayingCard Class:

Since in Solitaire a card can be face up or face down. Apart from that a card can be the top card of a pile or can be anywhere in the pile. These states of the card affects the way a card should be displayed on the screen. For instance if a card is face up and top-most then it can be displayed as shown in the **Figure 1** below:

However a top most card of a pile with face down should be displayed with it backside printed on screen. We assume our cards are red on the back side so the display of such a card would be as shown in **Figure 2**:

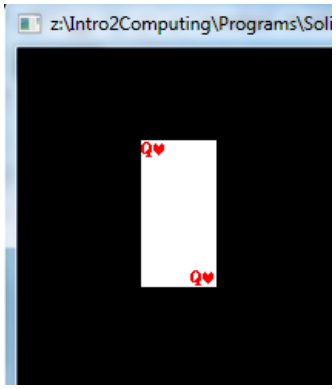


Figure 1

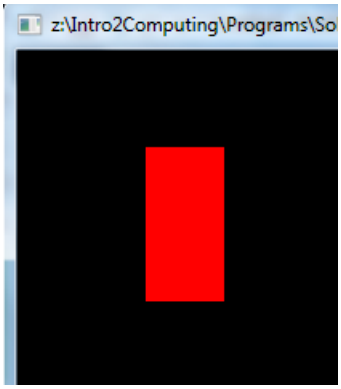


Figure 2

There are situations when a card will be displayed partially e.g. in **playing piles**. See the Figure 3 below of a pile having 5 cards. Pile is displayed in cascading manner with each upper card overlapping card below it and thus only showing it off partially. 4 cards are shown partially. Two face down and two face up. While the topmost card "A 2 of Spades" is shown fully. The horizontal lines that you see are the character underscore '_' printed.

Figure 3

If a card is not the top-most and has face up then it should be shown as below in **Figure 4**. Only two rows of the card are printed. First row is printed as usual, while second row is printed with a brown underscore character "_" on white background thus making a line:

If a card neither top-most nor face up then it should be shown as in **Figure 5**.

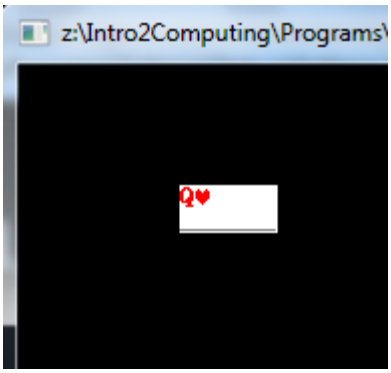


Figure 4

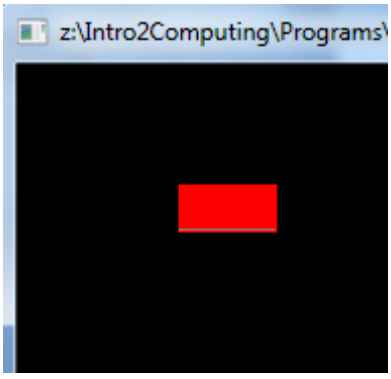


Figure 5

Add following additional private data members to class PlayingCard.

```
bool faceUp;    //tells if a card is face up or not
bool top;       //tells if a card is on top of a pile
```

Also add following member functions:

```
bool PlayingCard::isFaceUp(); //retunrs value of faceUp
void PlayingCard::setFaceUp(bool);  //sets value of faceUp
bool PlayingCard::isTop(); //retunrs value of top
void PlayingCard::setTop(bool);  //sets value of top
int PlayingCard::getSuit();   //getter for suit value
int PlayingCard::getRank();   //getter for rank value
char PlayingCard::getColor(); //getter for color value
```

Also update `void PlayingCard::display(int x, int y);` to display a card according to the values of data members `faceUp` and `top`. You've already done the display for `faceUp == true` and `top == true`. Update it to handle other 3 combinations of their truth values, and display cards according to examples given above.

Update PileofCards Class

Update member function `void PileofCards::Add(PlayingCard P)` such that the newly added card's top value is set to true while top value of the previously on-top card is set to false.

Likewise update member function `PlayingCard PileofCards::Remove()` such that after removing the top card if pile is not empty then the next exposed card's top value is set to true. If the exposed card's faceUp value is previously false then set it to true.

Add following data members in class. Their purpose will be discussed later.

```
point startPt;
point endPt;
```

Write following member functions:

```
int PileofCards::getTop();  //returns value of top
void PileofCards::setStartPt(int x, int y); //sets startPt
point PileofCards::getStartPt();            //returs startPt
void PileofCards::setEndPt(int x, int y);   //sets endPt
point PileofCards::getEndPt();              //returs endPt
PileofCards& PileofCards::operator=(const PileofCards& poc); //overload
                                                    //assignment operator

PlayingCard PileofCards::viewCard(int i); //It is just like peek function but
                                          //returns a card at index i rather
                                          //than top card and no card is
                                          //removed from pile
void PileofCards::SimpleDisplay(); //only displays top card of a pile,and
                                   //starting point of display is startPt, if
                                   //pile is empty a 6x8 dark_green rectangle
                                   //will be printed.
void PileofCards::CascadingDisplay(); //gives cascading display of a pile of
                                      //cards as in Figure 3, and starting
                                      //point of display is startPt, if pile
                                      //is empty, a 6x8 dark_green rectangle
                                      //will be printed.
```

In cascading display, if you have made `PlayingCard::display(int x,int y)` function fully, then all you have to do is call `display(x, y)` function of every card in the pile by providing appropriate values of x and y.

We are interested to know on what range of points, a pile of cards is displayed on screen so that if mouse clicks on screen we will know whether the click is made on pile or elsewhere. We can cover entire range of display just by keeping two points, startPt (top-left corner point) and endPt (bottom-right corner point). **Stock**, **waste** and **home** piles will always be displayed through simpleDisplay(), thus their start and end points of display will be as shown in the image below and can be hard-coded when a pile is created as they are always going to be displayed on same locations!

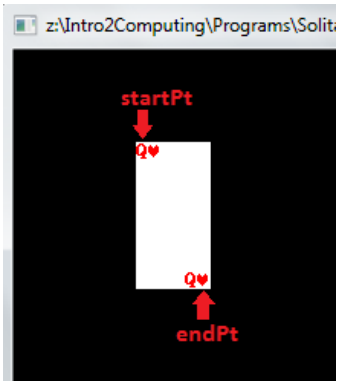


Figure 6

However in **playing piles** the start point will remain same throughout but end point will vary depending on number of cards in it. See the Figure 7 When pile had 4 cards the end point is at the bottom-right of the top-most card. On adding another card (2 of Spades) the end point has changed!

You will update the end point's value in **playing piles** whenever a card will be removed or added to them. (Caution: Do not update it in `PileofCards::Add(…)` and `PileofCards::Remove(…)` functions as they are also used for **stock**, **waste** and **home** piles). You can update it explicitly calling `PileofCards::setEndPt(int x, int y)` whenever add or remove functions for a **playing pile** will be called.

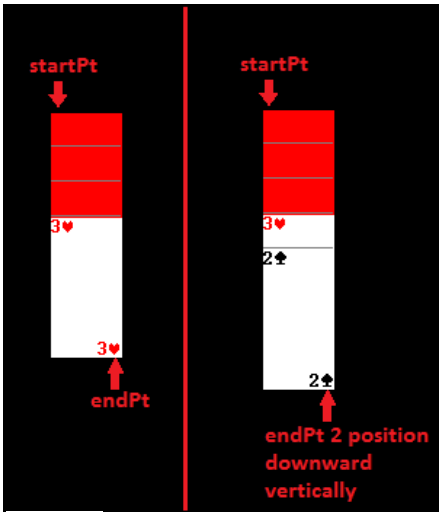


Figure 7

Solitaire:

In this part you will develop the game Solitaire. Define a class **Solitaire** providing all the functionality required to play the game. The initial implementation of this class is as follows.

```
#include "Deck.h"
#include "PileofCards.h"
class Solitaire
{
private:
	Deck deckofCards;
	PileofCards stock;
	PileofCards waste;
	PileofCards home[4];
	PileofCards playingPile[7];
	ConsoleFunctions cf;
public:
	Solitaire();
	void Shuffle();
	void dealToPlayingPiles();
	void DisplayGame();
	~Solitaire();
};
```

To start with we've a **`deckofCards`**, a **`stock`** pile, a **`waste`** pile, 4 home piles **`home[4]`**, 7 playing piles **`playingPile[7]`** and **`cf`** an object of ConsoleFunctions class to handle console display, input and output.

Constructor - Solitaire():

Initialize data members to appropriate values. Stock and waste piles should be able to hold miximum of 52 cards! Each home pile can have at most 13 cards.
A playing pile at first position has 1 card initially and should allow 13 more cards!
The playing pile at second position has 2 cards initially and should allow 13 more cards!
And so on…

Also set startPt and endPt values of each pile. You can hard-code startPt of each pile leaving the first line of console empty.

Initially all these piles are empty.

Shuffle():

To simulate shuffling of cards, cards are picked removed randomly from deck and added to stock pile as follows. All cards are turned face down during the process.

```
void Solitaire::Shuffle()
{
	int i;
	while (!deckofCards.IsEmpty())
	{
		i = rand()%deckofCards.getSize();
		PlayingCard c = deckofCards.Remove(i);
		c.setFaceUp(false);
		stock.Add(c);
	}
}
```

void dealToPlayingPiles():

It deals the cards to playing piles from stock pile. 1 card to first playing pile, 2 cards to 2nd playing pile, 3 to third pile and so on. Only top most cards on each pile are set face up while all the rest are face down.

void DisplayGame():

It displays all the piles in the Solitaire and display should appear as in the Figure 8 below:

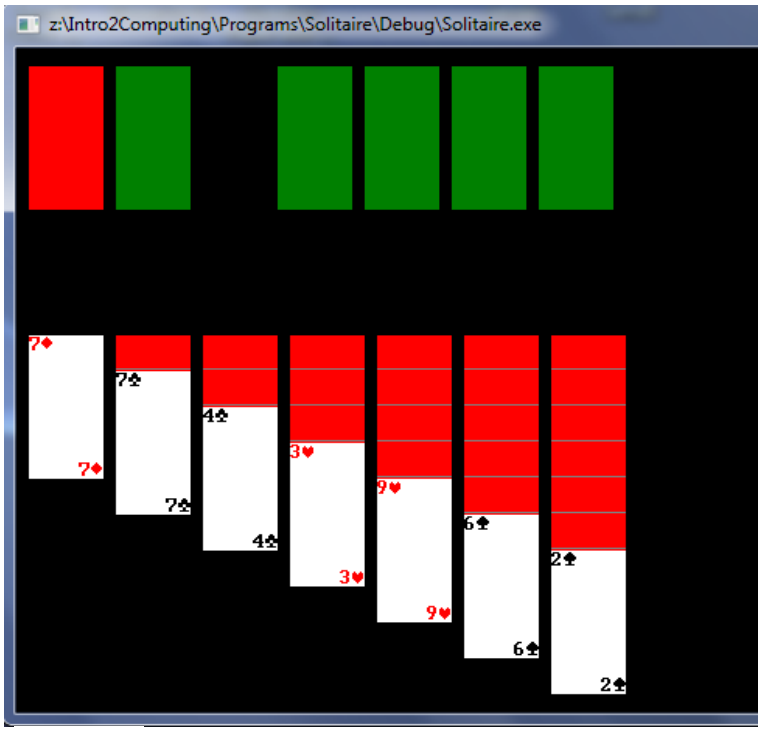


Figure 8

Test:

Write following main. If the above screen is displayed correctly, you're good so far:

```
	int main()
	{
		Solitaire S;
		S.Shuffle();
		S.dealToPlayingPiles();
		S.DisplayGame();
	}
```

In Part III you will implement complete functionality of the game.