Programming Assignment # 3 (part III) Playing Cards (Composition)	
Release Date	**25-Dec-2014 (Friday)**
Due Date	**8-Jan-2014 (Friday) 11:59 PM**
Submission	**On Google Classroom**
Total Marks	**50 (complete working)**

Solitaire

Object Oriented Programming Exercise

In this part you're going to implement entire game functionality by adding mouse inputs. In last part you developed the class **Solitaire**. Now you must add further functionalities to this class shown in red below. They're discussed one by one in this document.

```
#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();
	point getClickedPoint();
	PileofCards* getClickedPile(point p, int& a, int& c);
	void moveFromStock();
	void moveFromWasteToHome(int a);
	void moveFromWasteToPlayingPile(int a, int c);
	void moveFromPlayingPileToHome(int a, int b, int c);
	void moveFromPlayingPileToPlayingPile(int a, int b, int c1, int c2);
	void moveFromHomeToPlayingPile(int a, int b, int c);
	void Play();
	~Solitaire();
};
```

Before we move on we must know how user is going to play your game. Refer to the game rules discussed in part II, to know the legal moves. You can also check legal moves in Microsoft Solitaire Game packaged with Windows. Following six cases might appear during the game:

Case 1:
If user makes first click on non-empty **stock** pile, a card should be added from stock pile to waste pile face-up. If stock is empty then all the cards from **waste** pile (if non-empty) are moved to stock face-down.

Case 2:
If user makes first click on a non-empty **waste** pile and second click on a **home** pile, then a card from **waste** pile will be moved to **home** pile if it is a legal move. In case of invalid first or second clicks, the process to detect clicks should be repeated.

Case 3:
If user makes first click on a **waste** pile and second click on top card of a **playing pile**, a card from **waste** pile will be moved to that **playing pile** if it is a legal move. In case of invalid first or second clicks, the process to detect clicks should be repeated.

Case 4:
If a user makes first click on top card of a **playing pile** and second click on **home** pile, the top card from **playing pile** will be moved to **home** pile if it is a legal move. In case of invalid first or second clicks, the process to detect clicks should be repeated.

Case 5:
If a user makes first click on any face up card of a **playing pile** and second click on top card of another **playing pile**, the card or stack of cards from first **playing pile** will be moved to second **playing pile,** if it is a legal move. In case of invalid first or second clicks, the process to detect clicks should be repeated

Case 6:
If a user makes first click on a non-empty **home** pile and second click on top card of a **playing pile**, a card from **home** pile will be moved to **playing pile** if it is a legal move. In case of invalid first or second clicks, the process to detect clicks should be repeated.

Now the newly added functions to **Solitaire** class are discussed below.

point getClickedPoint():
This function runs an infinite loop until a click is made and returns the point of click. (Use console functions to detect clicked point in loop).

PileofCards* getClickedPile(point p, int& a, int& c):
Having a point of click alone is not enough until you know if the click is valid i.e. made on a pile shown on screen or not. Thus this function 'getClickedPile(…)' takes in a clicked 'point p' as parameter and checks by comparing it with **startPt** and **endPt** of each pile on the screen, that on which pile it lies. If a pile is found to have that point, a pointer to that pile is returned. You can see two more parameters.
Let us say that clicked point is on one of the four **home** piles, then 'int& a' should be containing the index of that home pile (range from 0-3).
Likewise if the clicked point is on one of the seven **playing piles** then 'int& a' should be containing index of that playing pile (range from 0-6). As for the parameter 'int& c' it will be needed only in case a playing pile is clicked. Since display of playing pile is cascading, exposing each card partially and top card fully, a click can be made on any card of the playing pile. Thus we need to know which card is clicked! Using the values of **startPt**, **endPt** of the clicked playing pile and the **point p** passed in parameter devise a way to find index of the card clicked. Its value will range from **0-top of the playing pile clicked**.

Following functions simply move the cards after verifying them as legal moves. These functions are called after the two clicked piles, the index of the pile (in case of home and playing piles) and the index of card (in case of playing pile) are identified using above functions. These values can be passed as parameters to following functions to perform desired move.

void moveFromStock():
This function moves a card from top of the stock pile to waste pile also making it face up. In case stock is empty, then all cards from waste are moved face down to stock. If waste pile is empty too then no move is made. (This function will be used in case 1 discussed above).

void moveFromWasteToHome(int a):
In this function parameter 'int a' is the index of home pile.
It moves a card from **waste** to pile **home[a]** after checking it as a legal move. (This function will be used in case 2 discussed above).

void moveFromWasteToPlayingPile(int a, int c):

In this function parameter 'int a' is the index of playing pile and 'int c' is the index of card in that playing pile.
It moves a card from **waste** pile on to the card, at index 'int c', of the **playingPile[a]**. You must check if move is legal. Also you can only move from waste pile to playingPile[a] if card at index 'int c' is the top-most card of playingPile[a]. (This function will be used in case 3 discussed above)

void moveFromPlayingPileToHome(int a, int b, int c):

In this function 'int a' is the index of playing pile and 'int b' is the index of home pile to which card is to be moved. Parameter 'int c' is the index of the card in playing pile.
It moves a card at index 'int c' from **playingPile[a]** to pile **home[b]** after checking it as a legal move. Remember you can only move card from playingPile[a] to home[b] if card at index 'int c' is top most card of playingPile[a]. (This function will be used in case 4 discussed above).

void moveFromPlayingPileToPlayingPile(int a, int b, int c1, int c2):

In this function 'int a' is the index of first playing pile and 'int b' is the index of playing pile to which card is to be moved. Parameter 'int c1' is the index of the card from first playing pile that is to be moved, and 'int c2' is index of the card in second pile onto which cards are to be moved.
It moves a card/stack of cards from index 'int c1' of **playingPile[a]** on to the card, at index 'int c2' of the playingPile[b]. You must check if the move is legal. Remember you can only move card from first pile **playingPile[a]** to second pile **playingPile[b]** if card at index 'int c2' is top-most card of the second **playinPile[b]**. This is not necessary for 'int c1' as an entire stack of face-up cards can be moved from playingPile[a]. (This function will be used in case 5 discussed above).

void moveFromHomeToPlayingPile(int a, int b, int c):

In this function 'int a' is the index of home pile from which card will be moved and 'int b' is the index of the playing pile to which card is to be moved. Parameter 'int c' is the index of card in the playing pile.
This function moves a card from pile **home[a]** on to the card, at index 'int c', of the playingPile[b]. You must check if move is legal. Also you can only move from home[a] to playingPile[b] if card at index 'int c' is top most card of playingPile[b]. (This function will be used in case 6 discussed above)

void Play():

This function runs and infinite loop and handles the 6 cases discussed above.
In loop it finds a clicked point by following function call:

```
point p1 = getClickedPoint()
```

Then clicked pile is identified by passing clicked point p1 as parameter to following call:

```
int a = -1; int c = -1;
PileofCards* firstPile = getClickedPile(p1, a, c);
```

To identify which pile the pointer firstPile is pointing at, you must compare it with each pile's address. Fo instance to check if it is pointing at stock pile, you can write following statement:

```
if ( firstPile == &stock)
{     cout<<"stock clicked!"  }
```

Likewise you can compare it with addresses of other piles too. In case of home pile and playingPile you know the index 'a' of the pile with which the comparison should be made.

If first clicked pile is stock then there is no need to check for second clicked pile. Otherwise you'll find the second point of click and thus the clicked pile.
Different moves should be made depending on first and second clicked piles, by calling the move functions already defined. Call function `DisplayGame();` after each move to update the game view. In case a move is invalid, print a statement "Invalid Move" by setting cursor at position (0, 0). Since your card display starts from position (1, 1), the displayed statement will not overlap.

Test:

Write following main. If the game works correctly, you've done good job ☺

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