Programming Assignment # 4 Playing Cards (Composition)	
Release Date	**07-Jan-2014 (Tuesday)**
Due Date	**14-Jan-2014 (Tuesday) 5:00 PM**
Submission Folder	**\\printsrv\Teacher Data\Esha\OOP 2013\Submissions\HW 4\BSSE Afternoon** **\\printsrv\Teacher Data\Esha\OOP 2013\Submissions\HW 4\BSSE Morning**
Total Marks	**TBA**

Playing Cards

Object Oriented Programming Exercise

In this exercise you will implement a program to simulate a game of cards. Read the document carefully and write the code as the steps follow.

Cards

The most obvious object in this game is a card. A card can be represented as follow.

```
class PlayingCard
{
  private:
  int rank;      //integer 1 – 13
  int suit;      //integer 0 – 3
  char color;   // red or black - 'b' for black , 'r' for red
  public:
  PlayingCard(int rank=0, int suit=0);
 };
```

Symbolic Constants

In class **PlayingCard**, a card is represented by two data fields i.e. **rank** and **suit**. The **rank** can have a value between 1 and 13 (1 for Ace). The **suit** is a number from 0 to 3, but how do we know which number represents which suit? We can solve this problem by creating symbolic constants to represent **suit**. A symbolic constant declared for a suit will have meaningful name and have a value between 0 and 3. A symbolic constant can be created by declaring a variable that is both **constant** and **static**.

Modify the class **PlayingCard** by declaring four symbolic constants as **public** data members of the class. The constants should be declared with following specifications.

Variable Name	**Value**
Diamond	0
Heart	1
Spade	2
Club	3

Making these variables static implies that the data values of these variables are associated with the class, not with an object. This is important because there will be many instances of class **PlayingCard**, and it would be very costly if each of them needed to maintain four additional data fields. These variables are made constant so that once the value has been assigned to the variable, it cannot be changed.

The symbolic constants need to be declared as **public** members of the class because values of these variables will be required outside the class. For example, following statement written in main function creates an object of **PlayingCard** representing 3 of spade.

PlayingCard card1(3, PlayingCard::Spade);

The data member **Spade** of class **PlayingCard** is public, so it can be accessed outside the class. Also, Spade is static, so it can be accessed with class name using scope resolution operator.

Here, ask yourself a question that does making symbolic constants **public** causes any problem?

Initialization of Symbolic Constants

Next step is to initialize the symbolic constants with the values specified above. Remember that a constant static variable cannot be initialized in the body of constructor. Therefore you have to write initialization statement at file scope.

Constructor

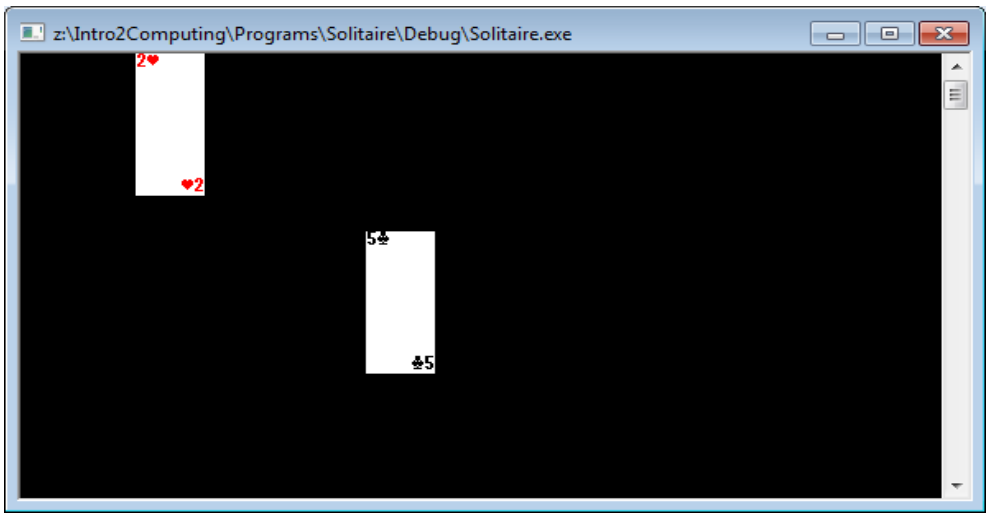
Write constructor for class **PlayingCard**. The constructor should take two integer values in parameters. The object of **PlayingCard** can be created and set to a state in two ways as follow.

```
PlayingCard card1(3, PlayingCard::Spade);

Or

PlayingCard card1(3, 2);
```

In first statement, the suit of the card is expected to be a valid value, because it is referred as *PlayingCard::Spade,* where **Spade** is a constant. But, in second statement, user can give invalid values. So, remember to provide appropriate checks in the body of constructor to make sure that an object is always in a consistent state. The color of the card should also be set to red or black according to its suit. Diamonds and Hearts are red. Spades and Clubs are black.

Write a function void PlayingCard::display(int x, int y), that must display a card of 6x8 dimensions using "ConsoleFunctions" class provided to you. The arguments x and y represent the point at which the card should be printed on console. See following example.

```
PlayingCard card2(2, PlayingCard::Heart);  //A 2 of hearts
card2.display(10,0); //prints the card starting from point (10,0) that is the top of console
                     //with 10 positions to the right.

PlayingCard card3(5, PlayingCard::Club);  //A 5 of clubs
Card3.display(30,10); //prints the card starting from point (30,10) that is the 10 positions
                      //downwards and 30 positions to the left of console
```



Making Pile of Cards

In this card game, we need to create Pile of Cards. A Pile of Cards is a collection of **PlayingCard** objects, one placed over the other. A card can be added or removed only to and from the top of the pile. To implement Pile of Cards, define a class as follow.

```
class PileofCards {
  private:
  PlayingCard *pile;
  int top;
  int size;
  public:
  PileofCards (int size );
  ~PileofCards();
  PlayingCard Peek();
  PlayingCard Remove();
  void Add(PlayingCard);
  bool IsEmpty();
  bool IsFull();
 };
```

Constructor

The constructor of the class **PileofCards** takes one arguments to set **size** of the pile. The size of a pile tells the number of cards that can be placed in it. The class also has a private data member **pile,** a pointer to **PlayingCard**. The pointer **pile** will point to an array of **PlayingCard** objects allocated dynamically. The data member **top** will always point to the last element added in the array of **PlayingCard**. The data member **top** will be initialized to value -1 to represent an empty pile.

Write appropriate code for constructor to initialize data members. Dynamically allocate an array of **PlayingCard** of size specified in constructor's argument. The address of the array will be stored in the pointer **pile**.

Destructor

In the lectures, we have discussed that the memory allocated dynamically is not freed automatically. When an object of **PileofCards** is constructed, an array of **PlayingCard** is created dynamically in its constructor. This array space has to be de-allocated explicitly when the object of **PileofCards** destroys. Write code in the destructor function to de-allocate memory area pointed to by the **pile** pointer.

Adding PlayingCard to the Pile

A new PlayingCard is always added at the top of the cards already in the pile. In this class the pile of cards is represented by an array of **PlayingCard**. A Playing card object can be added in the pile by storing it in the first free index of the array. Remember that the data member **top** of class **PileofCards** always points to the last element added in the array. So a new card will always be added to the index next to the **top.** After the addition of a new card, the value of **top** should be updated to point to the element newly added.

Removing a Card

Write code for Remove() function of the class. The function returns a **PlayingCard** object removed from the pile. A card can only be removed from the top of the pile. That is, the last card added in the pile will be removed first. For example if 5 cards A, B, C, D, and E are added in this order, the card on the top of the pile will be E and data member **top** will have value 4 (card E is stored at index 4). If we remove a card from the pile, card E will be removed.

In this operation the card placed in the array does not need to be removed physically. Keeping in mind that the data member **top** always points to the last element of the pile, we can remove an element by just decrementing the value of **top**.

Peeking Top of the Pile

Peek operation does not remove any card from the pile. It just returns the value of the card stored at the top of the pile. The implementation of this operation is given below.

```
PlayingCard PileofCards::Peek()
{    return pile[top];
}
```

Checking if Pile is Empty

A pile is empty if data member **top** has value -1. Write code for IsEmpty() function. The function returns **true** if the pile is empty.

Checking if Pile is Full

A pile is full when there is no index left free in the array and data member **top** points to the last index of the array. Write appropriate Boolean expression to check if pile is full. The function returns true if pile is full.

Testing

Test the classes you have implemented so far. You can use following sample code to test the classes. In order to debug your code properly, you need to write print statements at different points of execution in member functions to view the state of objects.

```
int main()
{
   PlayingCard A(3,PlayingCard::Spade);
   PlayingCard B(2,PlayingCard::Spade);
   PlayingCard C(4,PlayingCard::Heart);
   PlayingCard D(5,PlayingCard::Club);
   PlayingCard E(3,PlayingCard::Diamond);

   PlayingCard temp (3,PlayingCard::Heart);

   PileofCards pile1(5);

   /****** Adding cards in pile ********/
   pile1.Add(A);
   pile1.Add(B);
   pile1.Add(C);
   pile1.Add(D);
   pile1.Add(E);

   /*Note for Testing:
     The 6th card temp, in the statement below, should not be added to the
     pile, as at this point pile is full.
     Write a statement in IsFull function to display an error message if pile is
     full.
     */

  pile1.Add(temp);

 /****** Removing Cards from pile *****/

   temp = pile1.Remove();

   /* Here you will need to see which card is removed from the pile.
        For this you need to write display function for class PlayingCard to
        view a card. */

   temp.display(0,0);
```

```
   /* According to this scenario, the above statement should display the card
        3 of Diamond, as this is the card removed from the top of the pile*/

   temp = pile1.Remove();  //5 of Club should be removed
   temp.display(0,10);

   temp = pile1.Remove();  //4 of Heart should be removed
   temp.display(0,20);

   temp = pile1.Remove();  //2 of Spade should be removed
   temp.display(0,30);

   temp = pile1.Remove();  //3 of Spade should be removed
   temp.display(0,40);

   temp = pile1.Remove();  //Now the pile is empty

 /*Write a print statement in IsEmpty funtion to display an error message when
     pile is empty*/

  return 0;
}
```

Deck of Cards

In any game of cards we start with a deck. A deck contains 52 cards placed in order. At this point you must have discovered that a deck of cards is also a core object required in this game. This object will contain **PlayingCard** objects in it. So we need to define a class to represent deck of cards as follow.

```
class Deck
{
  private:
  PlayingCard *deck[52];
  int size;
  public:
  Deck();
  int getSize();
  bool IsEmpty();
  PlayingCard Remove(int i);
  ~Deck();
};
```

The relationship of composition can be established by placing an instance of class in another class as a data member. A Deck object will be composed of 52 cards as the class Deck contains in its data members 52 instances of class **PlayingCards**.

In this class, we establish composition relationship among objects by defining pointers to point to the **PlayingCard** objects created dynamically. Note that private data members of class Deck include an array of size 52. The type of array tells that it is an array of Pointers to **PlayingCard**. That is, each index of array is a pointer pointing to an object of type **PlayingCard**.

Construction of a Deck

Remember that the contained objects are constructed along with the Container object. The **PlayingCards** objects will be constructed dynamically using **new** operator in the constructor of class Deck.

For example, following statement dynamically creates a PlayingCard object and assigns its address to a pointer at index j of the array of pointers.

```
deck[j]= new PlayingCard(i,PlayingCard::Spade);
```

where 'i' is any number between 1 and 13.

Note that the constructor of Deck class takes no input to initialize contained objects, because the contained object have specific values that we already know and do not need to take from the user. For example, in the constructor of class deck, 13 cards of spade can be constructed as

```
for (int i = 1; i<= 13; i++)
 {
          deck[j]= new PlayingCard(i,PlayingCard::Spade);
           j++;
 }
```

Destroying an object of class Deck

We have studied in class that if an object or array of objects is created dynamically, it has to be destructed explicitly using **delete** operator. The class Deck contains an array of pointers, each pointing to a dynamic object. Keep in mind that the array **deck** itself is a static array of a specific size. So when an object of type Deck is destroyed the array will be destroyed automatically. But, what will happen to the objects created dynamically that were being pointed to by the indices of array **deck?**

You will need to write a code in destructor of class Deck to explicitly destroy each of the **PlayingCard** objects pointed to by each of the indices of array **deck**.

Testing

Test your class Deck for correctness of the constructor function by declaring an object of this class. Provide a member function in the class to display all the cards of the deck. This will help you to verify that correct object of class Deck is constructed with correct values of contained objects.

Removing a Card from the Deck

Write function to remove a card from the deck from any index. When a card is removed from index k, all the cards from index k+1 to last index are shifted to left.
Write appropriate code to shift the cards from a specific index towards left. Also remember to decrement the **size** data member of deck object after removing a card from that deck.

Again there is a challenge for you. If you just remove the object at index k by shifting the values from index k+1 towards left (as we do with integer array), it will not work. Because each index of the array is a pointer having an address of an object created dynamically. By shifting the values from index k+1 to the left, the old value at index k is overwritten. So, by overwriting a value of an index, we lose the address of the dynamic object space pointed to by that indexed variable. Therefore, you will have to physically remove the object from the dynamic space before shifting the remaining elements.

Also, Make sure that Card cannot be removed from an empty Deck.

Testing

Following is a sample code to test the working of Remove function of class Deck.

```
Deck D1;
PlayingCard A(0,0);
int y = 0;
A = D1.Remove(0);
A.display(0,y);
y=y+10;
```

```
A = D1.Remove(0);
A.display(0,10);
y=y+10;

A = D1.Remove(0);
A.display(0, 20);            // three cards removed
y=y+10;

while(!D1.IsEmpty())
{
        A = D1.Remove(0);
        A.display(0, y);
        y=y+10;
}
```