EECE.3220: Data Structures

Spring 2017

Programming Assignment #3: Classes

Due **Wednesday, 3/8/17**, 11:59:59 PM

1. Introduction

This assignment provides an introduction to programming with classes and with composition, an object-oriented programming technique in which one object contains one or more instances of a different type of object. In this program, you will model a deck of cards. You will also write more member functions in this lab that allow you to read and modify data members, as well as manipulate data in more interesting ways.

2. Deliverables

You should submit five files for this assignment:

- ***prog3_main.cpp:*** Source file containing your main function.
- ***Card.h:*** Header file containing the `Card` class definition.
- ***Card.cpp:*** Source file containing definitions of `Card` member functions.
- ***DeckOfCards.h:*** Header file containing the `DeckOfCards` class definition.
- ***DeckOfCards.cpp:*** Source file containing definitions of `DeckOfCards` member functions.

Submit all five files by uploading them to your Dropbox folder. Ensure your file names match the names specified above. Do not place these files in a subdirectory—place them directly in your shared folder. Failure to meet this specification will reduce your grade, as described in the program grading guidelines.

3. Specifications

Class Design: This assignment contains two classes:

- `Card`: a class to model an individual card by storing its rank (A, 2, 3, etc.) and suit (clubs, diamonds, hearts, and spades), each as a single character. I suggest at least designing functions to do the following:
    - Default and parameterized constructors
    - "Get" functions for both rank and suit
    - A single "set" function that changes both rank and suit
    - A function to print the rank and suit to a generic output stream
        - This function could be written as a member function (like the "`display()`" function in the List examples covered in class) or as an overloaded output operator.

3. Specifications (continued)

- `DeckOfCards`: a class containing 52 cards—all 13 ranks in each of the four suits. This class should contain the majority of the functionality for this assignment. I suggest at least designing functions to do the following:
  - Default constructor (A parameterized constructor is not necessary, since you know what the values of all cards in the deck should be.)
  - Return the top card from the deck.
  - Return a card from a specific position within the deck, assuming there are enough remaining cards to access that position.
    - For example, valid positions are initially between 0 and 51; with 5 cards remaining, valid positions are between 0 and 4.
  - Return a card from a random position within the deck.
  - Return the number of remaining cards in the deck.
  - Print the current contents of the deck.
  - Shuffle deck (randomize order after returning all cards to the deck)

Command Line Input and Corresponding Output: Your main program should start by prompting the user for a seed value for the random number generator, as in Program 2.

After reading that value, your program should accept the following commands:

- **'P':** Print the current contents of the deck, starting with the top card.
  - The deck should be printed as 13 cards per row, with the last row potentially containing fewer cards if the deck is not full. For example, the following shows the deck with five cards removed:

```
Current deck (47 cards remaining):
6c  7c  8c  9c  Tc  Jc  Qc  Kc  Ad  2d  3d  4d  5d
6d  7d  8d  9d  Td  Jd  Qd  Kd  Ah  2h  3h  4h  5h
6h  7h  8h  9h  Th  Jh  Qh  Kh  As  2s  3s  4s  5s
6s  7s  8s  9s  Ts  Js  Qs  Ks
```

- **'D':** Remove the top 5 cards from the deck and display them. For example, executing this command with the deck shown above gives the output:

```
Dealt top 5 cards: 6c  7c  8c  9c  Tc
```

- **'R':** Remove 5 random cards from the deck and display them in a similar manner to that shown for the 'D' command.
  - **NOTE:** Both the 'D' and 'R' commands should print an error if the deck holds less than 5 cards, but should not automatically shuffle.
- **'N':** Prompt the user to enter a starting position within the deck, then remove and display 5 cards starting at that position, assuming 5 cards are available after that point. Print an error if that condition is not true.
- **'S':** Shuffle the deck—return all cards to the deck and randomize card order. After shuffling the deck, print its current contents.
- **'X':** Exit the program.

4. Hints

Design process: We suggest the following approach to this assignment:

1. The `Card` class is relatively straightforward and is required to implement the `DeckOfCards` class. Design this class first, using a simple main program to test your `Card` functions.
2. Note that most of the suggested functions for the `DeckOfCards` class correspond closely to commands that your main program must recognize. Write a program to recognize the specified commands; then, update that program to test each appropriate `DeckOfCards` function as you write it.
3. The manner in which each `DeckOfCards` function works depends largely on how you track which cards have already been dealt from the deck—make sure you consider your approach before writing any functions for this class. One potential solution is described under **"Storing the deck"**.

   We suggest writing the `DeckOfCards` functions in this order:

   a. Default constructor
      i. See the note below about initializing array elements
   b. Print the current contents of the deck
   c. Return the number of remaining cards in the deck
   d. Return the top card from the deck
   e. Return a card from a specific position within the deck, assuming there are enough remaining cards to access that position
   f. Return a card from a random position within the deck
   g. Shuffle deck

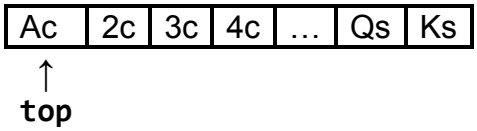
Arrays and constructors: Typically, when using object composition, we use initialization lists in our constructors. However, array data cannot be initialized in one of these lists—you must set each array value in the body of your constructor. The brief example below illustrates this principle:

```
class ExampleClass {
public:
     ExampleClass();
private:
     char c[2];
};

// WRONG CONSTRUCTOR—WON'T COMPILE
ExampleClass::ExampleClass() : c[0]('a'), c[1]('b') {}

ExampleClass::ExampleClass() {      // CORRECT CONSTRUCTOR
     c[0] = 'a';
     c[1] = 'b';
}
```

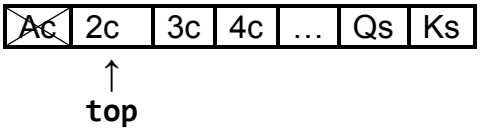
4. Hints (continued)

Storing the deck: An array of `Card` objects is likely the best way to store the deck. (A pointer-based data structure like a linked list might be more appropriate, but ignore that point for now.)

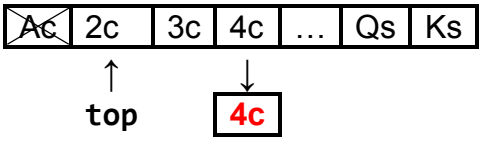
Your `DeckOfCards` class could therefore have a data member that tracks the top card in the deck by supplying the appropriate array index. Note that "top" does not have to be a pointer—this lab does not require pointers at all.



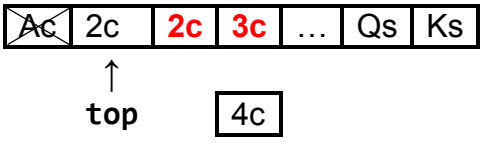
If you deal out the top card in the deck, tracking which cards are still in the deck is as simple as incrementing "`top`." The "Ac" will remain in the array, but you won't actually access that card again until you reshuffle your deck.



Note, however, that removing a card from any position but the top is a little more complex. Let's say we want to remove the "4c." What you'd ideally like to do is the following: first, "remove" that card:



Next, shift the cards "above" that card to fill the newly empty space:



Finally, copy the removed card to the spot that was the top of the deck (thus indicating that it's been dealt, but obviously still exists as a valid card), and increment `top`:

This process of removing a card from a specific position, shifting the other cards to fill that position, and copying the removed card into the old "top" position will be used in several of your `DeckOfCards` functions. You may want to write a private "helper" function to make this task a little easier.

5. Test Cases

Your output should closely match these test cases exactly for the given input values, at least in terms of format and general functionality. I will use these test cases in grading of your assignments, but will also generate additional cases that will not be publicly available. Note that these test cases may not cover all possible program outcomes. You should create your own tests to help debug your code and ensure proper operation for all possible inputs.

Also, note that your output may not match this case exactly, depending on when and where you call the `rand()` function. In particular, your shuffling implementation is likely to differ from mine. Any program that matches the specification above will receive appropriate credit.