Project 1: Card Deck

1. Introduction

In this assignment, you will implement two modules, **Card** and **CardDeck**, to represent playing cards in the program. The code skeleton provides you with three files (**Card.hpp**, **CardDeck.hpp**, **test.cpp)** and is under **/comp/15/files/p1** on the CS server. Your jobs are to implement Card class and CardDeck class, as well as to test the functionalities of your modules by writing programs that use your modules.

1. Log in to the CS homework server.
2. Move to your **comp15** directory that you created in Lab 1.
3. Create a directory named **project1** under the comp15 directory.
4. Move to the project1 directory.
5. Copy the code skeleton to the current working directory.
6. Leverage the features of Git to manage your progress toward writing the program.

As the first step, you are encouraged to take a look at the contents of the code skeleton and try identifying what each file is for. Then, spend enough time to think of how you represent a deck of cards in your program. As a part of the program specifications, you are asked to represent a deck of cards as an array of cards in your program.

You will at least need to create **Card.cpp** and **CardDeck.cpp** files by yourself. Note that the file names matter for the grading purpose and are case-sensitive. Note also that the public section of each **Card.hpp** and **CardDeck.hpp** cannot be modified. Finally, you are expected to write your own tests in **test.cpp**.

2. Specifications

Part 1: Card class

`Card(); //default constructor`

This card is initialized to be one that holds (suit) **SPADE** and (number) **1**.

`Card(std::string suit, int number); //non-default constructor`

This card is initialized to be one that holds the given suit and number. You can assume the suit is one of **SPADE**, **HEART**, **DIAMOND**, or **CLUB** (case-sensitive), and the number is between 1 and 13 (inclusive). Note that the Card class cannot represent the joker.

`Card(const Card& other); //copy constructor`
`Card& operator=(const Card& other); //assignment operator`

This card is initialized to be one that holds the same information that the given card holds.

`std::string getSuit() const;`

Return the suit that this card holds. i.e. **SPADE**, **HEART**, **DIAMOND**, or **CLUB** (case-sensitive).

`int getNumber() const;`

Return the number that this card holds.

`bool equals(Card card) const;`

Return true if this card holds the same suit and number that the given card holds; false, otherwise.

`std::string toString() const;`

Return the string representation of this card (case-sensitive). The representation should have the form **sn** where:

- **s** is one single character that uniquely identifies each suit. We use the first letter of each suit for this. i.e. (**S**)PADE, (**H**)EART, (**D**)IAMOND, (**C**)LUB
- **n** is one single character that uniquely identifies each number. We use [A, 2, 3, 4, 5, 6, 7, 8, 9. X, J, Q, K] to represents [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13], respectively.

For example, the string representation of a card holding SPADE and 7 is: **S7**
For example, the string representation of a card holding HEART and 12 is: **HQ**

Part 2: CardDeck class

`CardDeck();`

This card deck is initialized to be one whose capacity is 13 and that holds no card.

`CardDeck(const CardDeck& other);`
`CardDeck& operator=(const CardDeck& other);`

This card deck is initialized to be one that holds the same information that the given card deck holds.

`~CardDeck(); //destructor`

You are supposed to release (memory) resources used for this card deck.

`int getSize() const;`

Return the number of cards that this card deck holds.

`int getCapacity() const;`

Return the capacity of this card deck (how many cards the deck could contain).

`Card at(int index) const;`

Return the card found at the given index location (0-based) in this card deck. You can assume that the given index is within the range from 0 (inclusive) to the number of cards that this card deck is supposed to hold at that time (exclusive), which we refer as in-range.

`bool isEmpty() const;`

Return true if this card deck holds no card; false, otherwise.

`void add(Card card);`

Add a card that holds the same information that the given card holds to this card deck, so the added card becomes the last one in this card deck.

`void insert(Card card, int at);`

Insert a card that holds the same information that the given card holds to at the given index location (at, 0-based, in-range) in this card deck.

`void replace(Card card, int at);`

Replace the card found at the given index location (at, 0-based, in-range) in this card deck with a card that holds the same information that the given card holds.

`void remove(int index);`

Remove the card found at the given index location (0-based, in-range) in this card deck.

`void swap(int i, int j);`

Swap the two cards found at the given index locations (0-based, in-range) in this card deck.

`bool has(Card card) const;`

Return true if this card deck holds a card that has the same information that the given card holds; false, otherwise.

`int count(Card card) const;`

Return the number of cards in this card deck in which the card has the same information that the given card holds.

`CardDeck take(int n);`

Remove the first n cards of this card deck (where n is between 0 and the number of the cards that this card deck holds) and return another card deck that holds cards that hold the same information, including the order, that the removed n cards held.

`std::string toString() const;`

Return the string representation of this card deck (case-sensitive). The representation should have the form such that the string representation of each card in this card deck is followed by a comma which is followed by a space, except for the last card. Note that the order of cards matters.

For example, assume a card deck is holding (SPADE, 1), (HEART, 2), (DIAMOND, 3) and (CLUB, 4) in this order. The string representation of the card deck is: **SA, H2, D3, C4**

Another requirement:

The capacity of this card deck must be increased by 13 at the timing when a new card is being added but there is no available spot for the new card.

For example, assume this card deck, whose current capacity is 13, is holding 13 cards. After a new card is added to this card deck, the number of cards that this card deck is holding is 14 and the capacity of this card deck is 26.

3. Testing

Modify the **test.cpp** by adding your own test cases to ensure the functionalities of your modules. The given **test.cpp** includes one example test case using assert(). You are encouraged to add as many test cases as necessary to make you feel confident about your modules.

4. README

Create the README file that includes the following categories with appropriate section headers.

1. **Name**: The programmer's name.
2. **Date**: The last updated date of the program code, including README.
3. **Summary**: A brief summary of the project. (e.g. What the program does? How to use the modules? and so on.)
4. **Files**: A list of files that are necessary to build and test the program.
5. **Instructions**: A sequence of commands to compile and test the program. Note that you are expected to report procedures without using the make command.
6. **References**: A list of citations to information used to complete the project.
7. **Post evaluation on planning**: Did your project go smooth, as planned? If not, which parts? What were causes? How will you improve your planning for future assignments?

5. Submission

Submit your files listed below using Gradescope.

- Card.hpp

- Card.cpp
- CardDeck.hpp
- CardDeck.cpp
- test.cpp
- README