Sabancı University

Faculty of Engineering and Natural Sciences
CS204 Advanced Programming
Spring 2022

Homework 5 – Operator overloading for Playing Card Game

Due: 11/05/2022, Wednesday, 21:00

PLEASE NOTE:

Your program should be a robust one such that you have to consider all relevant programmer mistakes and extreme cases; you are expected to take actions accordingly!

You can NOT collaborate with your friends and discuss solutions. You have to write down the code on your own. Plagiarism will not be tolerated!

Introduction

In this homework, you are asked to implement a class for a trading card game. You will implement a class for a *deck of cards* and will overload some operators for it. The details of these operators will be given below. Our focus in this homework is on the class design and implementation. The usage of this class in the main program (i.e. the game implementation) is given to you in the homework pack.

Class Design for Playing Card Deck

You will use a struct called `Card` to model a single card. This struct will contain a data member to store *name* of the card (of type `string`) and another data member to store its *energy* (of type `int`). Card names can only be single words (no spaces in the name; this is an assumption). If you want, you can write constructor(s) for this struct, but that is not needed for the main program. However, you will **not** write any member functions for this struct.

The deck of cards will be implemented as a class named `Deck`. In the `Deck` class, there should be two private data members: (i) the cards inside the deck as a dynamic array of `Card` struct (i.e. `Card` pointer); (ii) the size of this dynamic array, i.e. the number of cards in the `Deck` object. It is a **must** to use such a class structure with a dynamic array of struct in the homework; you cannot use any other data structure.

In the remainder of the document, we will be referring to the beginning and end of the deck with the terms "top" and "bottom". The top of the deck is the beginning of it, i.e. it is the 0^{th} element in the `Card` array, and the bottom of the deck is the last element of the `Card` array. These definitions are demonstrated in Figure 1 below:

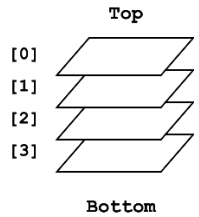

Figure 1: The top of the deck is the card at index 0 of that deck's `Card` array, and the bottom of the deck is the last element in the `Card` array.

Your `Deck` class implementation must include the following basic class functions:

- Default constructor: creates a `Deck` object with zero size. This constructor does not allocate memory for the elements of the deck since there are no cards.
- Deep copy constructor: takes a `Deck` object as const-reference parameter and creates a new `Deck` object as a deep copy of the parameter.
- Destructor: By definition, destructor deletes all dynamically allocated memory and returns them back to the heap.

In this homework, you will overload several operators for the `Deck` class. These operators and the details of overloading are given below. You can overload operators as member or free functions. However, to test your competence in both mechanisms, we require you to have at least three member and three free functions for these operators (the remaining five are up to you).

`<<` This operator will be overloaded to put a `Deck` object on an output stream (`ostream`). See the sample runs for the required output format. You must always print the deck starting from the top card to the bottom card. Since the left-hand side is an output stream, this function must be a free function.

Note that `+` operator will be overloaded in two different ways as (i) addition of a `Card` to a `Deck` and (ii) addition of a `Deck` to another `Deck`.

`+` This operator adds a card to the bottom of a deck. It will be overloaded such that it will add a `Card` item to a `Deck` object. The `Card` item is the **right-hand side (`rhs`)** operand of the `+` operator, and the `Deck` object is the **left-hand side (`lhs`)** operand. Since this is an operator that needs to return a value (not a reference), you must work on a brand-new `Deck` object in the implementation of this function. The content of the deck to be returned will be the cards of `lhs` in the same order as they appear in `lhs`, and after them the `rhs` card. In other words, the new deck will have the cards of the `lhs` deck starting the top of the deck, and the new card will be added to the bottom of the deck.

`+` This operator will be overloaded such that it returns the concatenation of two `Deck` operands. This operator also returns a brand-new `Deck` object (again as value, not reference) without changing the contents of the operands. The new deck will contain the cards of the left-hand side (`lhs`) deck first, and the cards of the right-hand side (`rhs`) deck will come after. To clarify, the cards of the `lhs` deck will be at the top of the new deck (without changing their order), and the cards of the `rhs` deck will be the bottom cards of the new deck (without changing their order). Please check Figure 2 for an illustration.

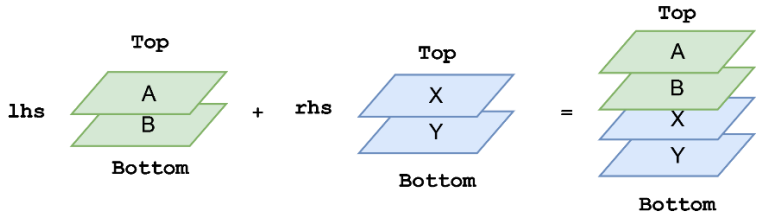

Figure 2: When adding two decks together, the resulting Deck object will have the cards of the left-hand side deck at the top, and the cards of the right-hand side deck at the bottom.

`+=` This operator will be overloaded such that it will add two `Deck` objects (left-hand size and right-hand side) and assign the resulting `Deck` object to the `Deck` object on the left-hand side of the `+=` operator. Addition logic of two decks has been explained above in Figure 2. This function must be implemented in a way to allow cascaded assignments.

Please note that a deck can have multiple instances of the same card, i.e., it can have two or more `Card` elements with the same name and energy values. You do not have to treat such cases in any different way in the addition operators.

- `-` This operator removes one or more cards from the top of a deck. The right-hand side parameter (`rhs`) is an **integer**, while the left-hand side (`lhs`) is a `Deck` object. You will not change the content of the left-hand side deck object. Instead, you will return a brand new deck object but with the top `rhs` cards removed. If `rhs` is greater than or equal to the number of cards in `lhs`, then this function should return an empty `Deck`. This function should also work for empty deck as `lhs`, and in such a case the function should return an empty deck. Please assume that `rhs` is a non-negative integer.

- `=` This operator will be overloaded such that it will assign the `Deck` object on the right hand-side of the operator to the `Deck` object on the left-hand side. This function must be implemented in a way to allow cascaded assignments. Due to C++ language rules, this operator must be a member function (cannot be free function).

- `==` This operator will be overloaded such that it will check that two `Deck` objects whether they have the same total energy. It returns true if same; otherwise, returns false. You can calculate the total energy of a deck by summing the energy values of all the cards in that deck.

- `<` This operator takes two `Deck` parameters. It will be overloaded such that it will check if the `lhs` deck has less total energy than the `rhs` deck. If less, returns true; otherwise returns false. You can calculate the total energy of a deck by summing the energy values of all the cards in that deck.

- `>` This operator takes a `Deck` object (`lhs`) as the left-hand side parameters and an integer (`rhs`) as a right-hand side parameter. It will return true if the deck `lhs` has more than `rhs` cards; returns false otherwise.

- `<=` This operator takes a `Card` element on the left-hand side and a `Deck` object on the right-hand side. It should return true if the card of the left-hand side `Card` element exists (same name and same energy) in the right-hand side `Deck` object. Otherwise, returns false.

- `[]` This operator takes a `Deck` object as a left-hand side parameter (`lhs`), and an integer as a right-hand side parameter (`rhs`). It will return a *reference* to the `Card` element at `rhs` index of the card array of deck `lhs`. While implementing the function, you do not need to deal with the index range issues since it is left to the user of your class to check the index range before using this operator.
  This operator is implemented using the operator function `operator[]` However, while using it in the main program, say to refer to `i`th element of the card array of `Deck` object `d`, as `d[i]`. Such a usage might be confusing for you since the `Deck` object `d` is not an array per se and we normally use such a square bracket operator in order to reach a particular element of an array/vector/string directly. Actually, here we indirectly do the same (via the operator function that you will implement) over the card array of `Deck` object.

In order the see the usage and the effects of these operators, please see **sample runs** and the provided **main.cpp**.

In this homework, you are allowed to implement some other helper member functions, accessors (getters) and mutators (setters), if needed. However, you are **not allowed to use friend functions and friend classes**.

A life-saving advice: Any member function that does not change the private data members needs to be **`const` member function** so that it works fine with const-reference parameters. If you do not remember what "`const` member function" is, please refer to CS201 notes.

Another important advice: First, please learn the return type and parameter structures of the operators (whether they are reference or value, whether they are const or not) and the tricks of developing them by checking out the lecture notes. Then, start designing and coding the homework. Any wrong design decision or wrong approach taken would cause severe bugs that you cannot resolve easily. Do not Google these stuff since online resources may mislead you; please refer to lecture notes.

You have to have separate header and implementation files for your class, in addition to the main.cpp. Make sure that the header file name that you submit and `#include` are the same; otherwise your program does not compile.

main.cpp

In this homework, **main.cpp** file is given to you within this homework package and we will test your codes with this main function with different inputs. You are not allowed to make any modifications in the main function (of course, you can edit to add `#include`s, etc. to the beginning of the file). All class related functions, definitions and declarations (including free operator functions) must be done in class header and implementation files. Inputs are explained with appropriate prompts so that you do not get confused, also you can assume that there won't be any wrong inputs in test cases; in other words, you do not need to do input checks. We strongly recommend you to examine main.cpp file and sample runs before starting your homework.

We have not explained the input files and the rules of the game in this document. Thus, reading and understanding main.cpp is very important since it is the only way to understand the file inputs and how the game is played.

Rules and Submission

General rules and the submission guidelines are the same as previous homework assignments, please refer to them. You have to submit both .cpp and .h files of the class that you will develop (after properly renaming) together with the provided main program file (again after properly renaming and updating the class header file `#include` line).

Good Luck!
Albert Levi and Amro Alabsi Aljundi

Sample Runs

Some sample runs are given below, but these are not comprehensive, therefore you have to consider **all possible cases** to get full mark. Especially try different deck contents (other than the ones given in the sample runs) and extreme cases. Inputs are shown in **bold** and *italic*.

Please do not try to understand the game by checking out the sample runs only. They may give you an idea, but in general, sample runs are not that helpful to understand how the game ends, how the winner is determined and what the winner wins. Instead, you have the check out the provided main.cpp. We have extensively commented the provided code.

Sample Run 1:

ex1_d1.txt

```
Python 2
C 10
Rust 7
```

ex1_d2.txt

```
Java 0
JavaScript 6
C++ 11
```

Output

```
SU Trading Card Game
Please enter name of the first deck's file: ex1_d1.txt
Please enter name of the second deck's file: ex1_d2.txt
First deck:
1: Python - 2
2: C - 10
3: Rust - 7

Second deck:
1: Java - 0
2: JavaScript - 6
3: C++ - 11


Deck 1 has more total energy than Deck 2!

Deck 1 has a total of 3 cards and 3 of them are unique.
Deck 2 has a total of 3 cards and 3 of them are unique.

Cutting deck 1...
Deck 1 after cutting:
1: C - 10
2: Rust - 7
3: Python - 2

Cutting deck 2...
Deck 2 after cutting:
1: JavaScript - 6
2: C++ - 11
3: Java - 0

P1: C @ 10 Vs.  P2: JavaScript @ 6
Player 1 wins the round
P1: Rust @ 7 Vs.  P2: C++ @ 11
Player 2 wins the round
P1: Python @ 2 Vs.  P2: Java @ 0
Player 1 wins the round
P1: C @ 10 Vs.  P2: C++ @ 11
Player 2 wins the round
P1: Python @ 2 Vs.  P2: C++ @ 11
Player 2 wins the round
Player 2 wins the game!

The winning deck:
1: JavaScript - 6
2: Rust - 7
3: Java - 0
4: C - 10
5: Python - 2
6: C++ - 11
```

Sample Run 2:

ex2_d1.txt

```
Dog 2
Cat 3
Mouse 1
Horse 4
```

ex2_d2.txt

```
Dragon 1000
```

Output

```
SU Trading Card Game
Please enter name of the first deck's file: ex2_d1.txt
Please enter name of the second deck's file: ex2_d2.txt
First deck:
1: Dog - 2
2: Cat - 3
3: Mouse - 1
4: Horse - 4

Second deck:
1: Dragon - 1000

Deck 1 has less total energy than Deck 2!

Deck 1 has a total of 4 cards and 4 of them are unique.
Deck 2 has a total of 1 cards and 1 of them are unique.

Cutting deck 1...
Deck 1 after cutting:
1: Mouse - 1
2: Horse - 4
3: Dog - 2
4: Cat - 3

Cutting deck 2...
Deck 2 after cutting:
1: Dragon - 1000

P1: Mouse @ 1 Vs.  P2: Dragon @ 1000
Player 2 wins the round
P1: Horse @ 4 Vs.  P2: Dragon @ 1000
Player 2 wins the round
P1: Dog @ 2 Vs.  P2: Dragon @ 1000
Player 2 wins the round
P1: Cat @ 3 Vs.  P2: Dragon @ 1000
Player 2 wins the round
Player 2 wins the game!

The winning deck:
1: Mouse - 1
2: Horse - 4
3: Dog - 2
4: Cat - 3
5: Dragon - 1000
```

Sample Run 3:

ex3_d1.txt

```
Ali 5
Berker 4
```

ex3_d2.txt

```
Ahmed 3
Ahmed 3
Elif 5
Muge 4
```

Output

```
SU Trading Card Game
Please enter name of the first deck's file: ex3_d1.txt
Please enter name of the second deck's file: ex3_d2.txt
First deck:
1: Ali - 5
2: Berker - 4

Second deck:
1: Ahmed - 3
2: Ahmed - 3
3: Elif - 5
4: Muge - 4

Deck 1 has less total energy than Deck 2!

Deck 1 has a total of 2 cards and 2 of them are unique.
Deck 2 has a total of 4 cards and 3 of them are unique.

Cutting deck 1...
Deck 1 after cutting:
1: Berker - 4
2: Ali - 5

Cutting deck 2...
Deck 2 after cutting:
1: Elif - 5
2: Muge - 4
3: Ahmed - 3
4: Ahmed - 3

P1: Berker @ 4 Vs.  P2: Elif @ 5
Player 2 wins the round
P1: Ali @ 5 Vs.  P2: Muge @ 4
Player 1 wins the round
P1: Ali @ 5 Vs.  P2: Ahmed @ 3
Player 1 wins the round
P1: Ali @ 5 Vs.  P2: Ahmed @ 3
Player 1 wins the round
P1: Ali @ 5 Vs.  P2: Elif @ 5
Players tied this round
Game ended with a tie!

The winning deck:
```

Sample Run 4:

ex4_d1.txt

```
```

ex4_d2.txt

```
Mario 10
Luigi 11
Luigi 11
Peach 12
```

Output

```
SU Trading Card Game
Please enter name of the first deck's file: ex4_d1.txt
Please enter name of the second deck's file: ex4_d2.txt
First deck:

Second deck:
1: Mario - 10
2: Luigi - 11
3: Luigi - 11
4: Peach - 12

Deck 1 has less total energy than Deck 2!

Deck 1 has a total of 0 cards and 0 of them are unique.
Deck 2 has a total of 4 cards and 3 of them are unique.

Cutting deck 1...
Deck 1 after cutting:

Cutting deck 2...
Deck 2 after cutting:
1: Luigi - 11
2: Peach - 12
3: Mario - 10
4: Luigi - 11

Player 2 wins the game!

The winning deck:
1: Luigi - 11
2: Peach - 12
3: Mario - 10
4: Luigi - 11
```

Sample Run 5:

ex5_d1.txt

```
Water 1
Fire 1
Earth 1
Air 1
```

ex5_d2.txt

```
Metal 4
```

Output

```
SU Trading Card Game
Please enter name of the first deck's file: ex5_d1.txt
Please enter name of the second deck's file: ex5_d2.txt
First deck:
1: Water - 1
2: Fire - 1
3: Earth - 1
4: Air - 1

Second deck:
1: Metal - 4

Deck 1 and Deck 2 have the same total energy!

Deck 1 has a total of 4 cards and 4 of them are unique.
Deck 2 has a total of 1 cards and 1 of them are unique.

Cutting deck 1...
Deck 1 after cutting:
1: Earth - 1
2: Air - 1
3: Water - 1
4: Fire - 1

Cutting deck 2...
Deck 2 after cutting:
1: Metal - 4

P1: Earth @ 1 Vs.  P2: Metal @ 4
Player 2 wins the round
P1: Air @ 1 Vs.  P2: Metal @ 4
Player 2 wins the round
P1: Water @ 1 Vs.  P2: Metal @ 4
Player 2 wins the round
P1: Fire @ 1 Vs.  P2: Metal @ 4
Player 2 wins the round
Player 2 wins the game!

The winning deck:
1: Earth - 1
2: Air - 1
3: Water - 1
4: Fire - 1
5: Metal - 4
```