

Advanced Object Oriented System Design Using C++

[Dashboard](#) / [My courses](#) / [2020-2021 Academic Year](#) / [COMP3400-30-R-2021W](#) / [31 January - 6 February](#) / [Assignment 4](#)

Assignment 4

Overview

There are many card games. Typically each card game starts with a deck of cards, they are randomly shuffled, and then cards are drawn from the deck. Often card games calculate a score based on the cards one has in his/her hand. This assignment explores using C++ to perform some of these operations.

Task

Your task is to use the provided **card.hxx** (scroll down to closer to the bottom of this page to access such) header file to write code in a **a4.cxx** (or use a .cpp extension if you prefer) as follows (high-level):

1. Generate a full deck of cards in the container `std::deque<card>`.
2. Randomly shuffle the deck of cards.
3. Draw 3 cards from the deck and store in a hand.
4. Calculate the possible low and high score of the cards in the hand.

These steps and what needs to be output are described below.

NOTE: Compile your code as `C++20` as the header file uses a feature from C++ 20, e.g.,

```
1. g++-10.2.0 -std=c++20 -Wall -Wextra -Werror -o a4.exe a4.cxx
```

Starting This Assignment

You are not permitted to modify card.hxx. Do not copy code from other cards programs in this course --concentrate on writing the code in this assignment in a4.cxx as described below.

At the top of your a4.cxx, start by writing the following include directives:

```
1. #include <deque>
2. #include <iterator>
3. #include <random>
4. #include <numeric>
5. #include <iostream>
6.
7. #include "card.hxx"
```

Note, by convention, include files enclosed within `<` and `>` are searched for in the compiler's "include" search paths and files enclosed within double quotation marks are searched for relative to the current directory.

Although using such won't be leveraged in this assignment, you are to store the deck of cards as well as a "hand" in double-ended queues, i.e., in `std::deque<card>` variables. Rather than writing `std::deque<card>` each time you want to say "cards", define a type alias called "cards" that is a synonym for `std::deque<card>` as follows:

```
1. using cards = std::deque<card>;
```

(Type aliases and typedefs are the same: both define a name as a synonym for another type. Unlike typedef, however, type aliases (i) can be templated and (ii) are easier to read and write due to their assignment-like syntax.)

Outputting cards to std::ostream

Given previous assignments and the lectures, by now you should be good at writing `std::ostream` operator `<<()` overloads to output data. Write such an overload to output cards `const&` values to a `std::ostream` with the code inside the function done as follows:

- Call the [`std::copy`](#) algorithm starting at the "beginning" iterator position of cards going up to (but not including) the "end" iterator position of cards, and,
- the third argument to `std::copy` should be an unnamed variable of type [`std::ostream_iterator<card>`](#) whose constructor arguments are (i) the ostream and (ii) a " " (i.e., a space).
- Remember to return the `std::ostream` parameter at the end of the function.

While one can use loops, etc. to process each element of a container, simpler code (without any loss of efficiency) can be written by calling the appropriate algorithms. Clearly it is easier to understand and maintain one line of code (i.e., the `std::copy()` call is one line of code) than multiple lines with variables, test conditions, etc.

Generating the Deck of Cards

In `card.hxx`, there are two special classes provided called `card_faces` and `card_suits`. The purposes of these classes are to provide an easy way to iterate through all possible card faces and all possible card suits. These two classes are very similar to iterators but since understanding such syntax and operator overloads are new to you, the code for generating a deck of cards is given:

```
1. cards gen_deck_of_cards()
2. {
3.     cards retval;
4.     for (card_faces f(card_faces_begin), fEnd; f != fEnd; ++f)
5.         for (card_suits s(card_suits_begin), sEnd; s != sEnd; ++s)
6.             retval.push_back(card{*f,*s});
7.     return retval;
8. }
```

(Place this code in your `a4.cxx` file.)

NOTE: While one could have written the for loops to directly use the "face" values, this would have required hard-coding values in the for loop. Since the suit values are part of an enumeration, the for loop code to loop through all possible suits would have been more complicated. By defining two special "iterator-styled" classes, one can easily obtain all possible values of faces and suits in a way that results in clean, easy-to-understand-and-maintain code.

Tip: It is a good idea before proceeding further to add a `main()`, call this function, and output the cards to verify things are working.

Randomly Shuffling the Deck of Cards

Once one has some cards, the next step would be to randomly shuffle them. The C++ standard defines:

- [`std::shuffle`](#) in `<algorithm>` which will randomly shuffle a random-access iterator range `[first,last)` according to a random number generator,
- [`std::random_device`](#) in `<random>` is a special uniformly-distributed integer random number generator that produces non-deterministic numbers using special hardware
- [`std::mt19937`](#) in `<random>` is a Mersenne Twister that generates 32-bit random numbers.

It is best to always put "single" operations in their own functions instead of adding a lot of code to one function / `main()`. thus, start by writing the function to shuffle cards as follows:

```
1. void shuffle_cards(cards& cs)
2. {
3. }
```

Notice the cards function parameter is passed by reference --you will be modifying the cards passed in (obviously!). :-)

To shuffle the cards, there are three steps required:

1. Declare a variable of type `std::random_device` called `rd`.
2. Construct a variable of type `std::mt19937` called `gen` passing in to its constructor `rd()`.
3. Call `std::shuffle` passing in `cs.begin()`, `cs.end()`, and `gen`.

Random number generation in C++ involves the use of types that behave as function objects, i.e., classes which overload `operator()`. This is why "rd" has parentheses after it in step 2. Inside the `std::shuffle()` code, the equivalent of `gen()` is used to generate the next random number.

NOTE: See §17.1: Random Numbers and Distributions in [ICPPSL](#) for more detail on C++ random number generation. :-)

Tip: It is a good idea before proceeding further to hack a but of code in `main()` to check if your cards are being properly shuffled.

Drawing N Cards From the Deck

Given some cards, one may want to request drawing N cards from those cards. In this function, you will draw (i.e., take) at most N cards from the front of the deck, remove those cards from the deck, and return the drawn cards in a cards object. First, start by writing:

```
1. cards draw(std::size_t num, cards& deck)
2. {
3. }
```

You are to determine the N cards to draw using iterators. Since it is possible that `num > deck.size()`, one cannot unconditionally start at the beginning and then ++ five times to get the five cards --the deck might not have enough (or any!) cards. Instead these are the steps you want to perform:

1. In a single line of code, declare a variable called **pos** with its constructed value set to the result of `deck.begin()`. (Tip: Use `auto` to declare this variable.)
2. If `num <= deck.size()` then call `std::advance(pos, num)`; otherwise set `pos = deck.end()`; (Note: This will identify at most num cards --but also handles all cases including when there are too few or no cards.)
3. Declare a variable of type **cards** and pass in to its constructor two parameters: `deck.begin()` and `pos`. (Note: All C++ containers have constructors that accept an iterator range which copies all values in that range in to the container, e.g., <https://en.cppreference.com/w/cpp/container/deque/deque>.)
4. Call `deck.erase()` passing in `deck.begin()` and `pos`. (Note: This erases the cards in the range `[deck.begin(),pos)` from deck.)
5. Return the variable you declared in step 3.

Tip: Check that this function works before continuing further.

Calculating the Score of a "Hand"

C++'s algorithms work on iterators and/or iterator ranges. There are many algorithms in [<algorithm>](#) and [<numeric>](#) in C++. Calculating the score of cards in a container amounts to iterating from the beginning to the "end" of the container summing up their score. (NOTE: In this assignment that each card contributes to the score only on its own.) While this can be manually done with a for loop, there is an algorithm, [std::accumulate\(\)](#), capable of "adding" up elements in an iterator range `[first,last)`.

The score of a container cards, i.e., a "hand", is to be calculated as follows:

- aces are worth 1 or 11,
- jacks, queens, and kings are worth 10,
- `card::invalid` is worth 0 (obviously and such should never be in a deck or hand),
- otherwise the card is worth its face value (i.e., 2 to 10).

(Some might recognize that these are the values of cards in the game of Blackjack.)

Since aces can be worth 1 or 11, in this assignment you are to compute the score:

- by computing the lowest possible score,
- by computing the highest possible score, and,
- returning both from this function.

Thus, to calculate the score, you want to **return two values** from a function. You are not permitted to do this indirectly by passing arguments by reference to the function. Instead you will return the results in a single object from the function. This means your `calc_score()` function could be prototyped a one of:

```
1. auto calc_score(cards const& cs) -> std::pair<std::size_t, std::size_t>;
```

or as:

```
1. auto calc_score(cards const& hand) -> cards;
```

(These prototypes are using [C++11](#)'s function suffix notation.)

Returning cards in a type **cards** value would work as there can be 0 or more elements in the container --the caller can check how many results there are. However, in this assignment you will return a [std::pair](#) object for these reasons:

- all of C++'s associative "map" containers store their elements as `std::pair<Key,Value>` where Key and Value are the types of those objects (so it is good to know how to use **std::pair**),
- using **std::pair** demonstrates how to store two values in a single object, and,
- using **std::pair** allows one to learn how to access the elements in **std::pair**.

(ASIDE: **std::pair<A,B>** is legacy (but will remain in the language) and has been made to be essentially compatible with **std::tuple<A,B>**, i.e., a tuple containing two elements of types A and B. Tuples are more general however supporting 0 to N elements of distinct types.)

Write the code in **calc_score()** as follows:

1. **size_t low = accumulate(hand.begin(), hand.end(), size_t{}, LOW_SUM_OP);**
2. **size_t high = accumulate(hand.begin(), hand.end(), size_t{}, HIGH_SUM_OP);**
3. **return { low, high };**

where LOW_SUM_OP and HIGH_SUM_OP are functions, lambda functions, or function objects (of your choosing) to calculate the low and high score in a hand respectively. (Try using lambda functions here!) The signature of LOW_SUM_OP and HIGH_SUM_OP are what [std::accumulate](#) requires when a binary operator is passed in. The first argument is the current sum, and, the second argument is the next element in the container to add to the current sum. Thus, the return value of both LOW_SUM_OP and HIGH_SUM_OP is the first argument + whatever score the current card is. This also means the first argument will be **std::size_t const&** and the second argument is **card const&**.

Tip: Check that this function works before continuing further.

Writing main()

Your main() function should now be easy to write if you wrote test code for the above! The code in main() is to do the following in the order listed:

1. **In a single line of code**, construct a local variable called **deck** of type **cards** passing the result of **gen_deck_of_cards()** to it.
2. **In a single line of code**, call **shuffle_cards()** using **deck** from step 1.
3. Output "**Deck:** " followed by all cards in **deck** (each card is separated by a space) followed by a newline char.
4. Output an empty line.
5. Output "**Drawing 3 cards...\n**".
6. **In a single line of code**, construct a local variable called **hand** passing the result of **draw(3,deck)** to it.
7. Output "**Deck:** " followed by all cards currently in **deck** (each card is separated by a space) followed by a newline char.
8. Output "**Hand:** " followed by all cards in **hand** (each card is separated by a space) followed by a newline char.
9. **In a single line of code**, construct a local variable with the result returned by **calc_score(hand)**.
10. If the first and second score results are the same, then output "**Score:** " followed by the score and then a newline char.
11. Otherwise output "**Possible Score:** " followed by the **low** and **high** scores and then a newline char.

Tip

Don't write the entire program before you test things! Instead write smaller chunks of code that are made of short functions (say 3 to 10 lines long) and check that each function is correct. This will make it a lot easier to write *correct* code that uses those functions --in any programming language! :-)

Sample Program Runs

Since the program output is random, you will need to run your program multiple times to see the one and two score output variants. Here are two sample runs:

```


1. $ ./a4.exe
2. Deck: QH 1C 4D JS 8D 8C 1S JD 7D KD 5H QC 4H KC AS 8S 3S 3H 5S AC 7C 8H 6D 2C AD 9S KH 6C 2D 4C
   KS 9C QS 3D 2S 5D 3C 5C AH 7H 9D 9H QD JC 4S 7S 6S JH 6H 1H 1D 2H
3.
4. Drawing 3 cards...
5. Deck: JS 8D 8C 1S JD 7D KD 5H QC 4H KC AS 8S 3S 3H 5S AC 7C 8H 6D 2C AD 9S KH 6C 2D 4C KS 9C QS
   3D 2S 5D 3C 5C AH 7H 9D 9H QD JC 4S 7S 6S JH 6H 1H 1D 2H
6. Hand: QH 1C 4D
7. Score: 24
8. $ ./a4.exe
9. Deck: 7D 8S AH 1S 9H 9S JS 5D KC 4D QS JH 5C AC 6D QH 3S 6S QD 1H JD 6C 1D 9C 7H 4C KD 5S 8H 8D
   AS 1C 8C KS 7C 2D 2S AD 3C 4H QC 5H 4S 3D KH 7S 2C 9D 2H 3H JC 6H

```

```
10.
11. Drawing 3 cards...
12. Deck: 1S 9H 9S JS 5D KC 4D QS JH 5C AC 6D QH 3S 6S QD 1H JD 6C 1D 9C 7H 4C KD 5S 8H 8D AS 1C 8C
    KS 7C 2D 2S AD 3C 4H QC 5H 4S 3D KH 7S 2C 9D 2H 3H JC 6H
13. Hand: 7D 8S AH
14. Possible Score: 16 26
15. $
```

Submission

When submitting your assignment only submit your .cxx / .cpp file, i.e., --do not submit the card.hxx file. :-)

-  [card.hxx](#)

24 February 2021, 1:34 PM

Submission status

Attempt number	This is attempt 1.
Submission status	No attempt
Grading status	Not marked
Due date	Friday, 26 February 2021, 11:59 PM
Time remaining	23 hours 12 mins

Grading criteria

This rubric is a general rubric for compiled programming languages.

Normally the marker will adhere to the rubric, but, the marker reserves the right to add/subtract marks based on the quality of the work provided. (Such overrides will be noted in the feedback comments.)

Compiler Warnings	Too many compiler warnings. <i>0 points</i>	No more than two (2) compiler warnings. <i>1 points</i>	No compiler warnings. <i>2 points</i>
Compiler Errors (Deduction)	Code fails to compile due to compiler errors. <i>-5 points</i>		No compiler errors. <i>0 points</i>
Possible Run-Time Errors	Too many run-time errors occur or can occur. <i>0 points</i>	Only a few minor run-time errors occur or can occur. <i>1 points</i>	No run-time errors occur or can occur. <i>2 points</i>
Code Structure, Names, and Comments	Poor code structure, meaningful symbol names, and comments. <i>0 points</i>		Satisfactory code structure, meaningful symbol names, and comments. <i>1 points</i>
Packaging	Not all files are provided and/or not all targets are built cleanly and completely. <i>0 points</i>		All files are provided. All targets are built cleanly and completely. <i>1 points</i>
Program Output	Program output has issues. <i>0 points</i>		Program output is correct (or nearly so). <i>1 points</i>
Assignment Requirements Conformance	Too many instructions were not followed. <i>0 points</i>	Some instructions were followed. <i>1 points</i>	Most instructions were followed. <i>2 points</i>
			All instructions were followed. <i>3 points</i>
Late Penalty (Deduction)	Late. All marks deducted. <i>-10 points</i>	Late but submitted within 24 hours of deadline or late without penalty deadline. <i>-2 points</i>	Not late or late without penalty (as decided by or with permission of instructor). <i>0 points</i>

Last modified

-

Submission comments

► [Comments \(0\)](#)

Add submission

You have not made a submission yet.

[◀ Week 04 Lecture Files](#)[Slides: C++ Objects and Exceptions ▶](#)

You are logged in as Ravi Trivedi (Log out)

COMP3400-30-R-2021W

Links

[School of Computer Science](#)

[University of Windsor](#)

[Blackboard](#)

[Data retention summary](#)

[Get the mobile app](#)

All user-generated content, posts, and submissions are copyright by their respective author(s).

All course materials, activities, etc. are copyright by their respective author(s).

Copyright © 2013-2020. All Rights Reserved.