Assignment 13: C++ Classes and Data Abstraction

COSC 1437: Programming Fundamentals II

Objectives

- Practice defining and adding our own user defined types to C/C++
- Learn how to declare and use C++ `class` user defined types.
- Learn some of the basics of Object Oriented Design (OOD) and its principles like information hiding.
- Use public and private member variables and functions in a class, and become familiar with why things should be public or private.
- Familiarize yourself with class constructors and member methods.
- More practice with using arrays and other parts of the C++ language.

Description

This assignment in many ways is a continuation of the previous assignment 12 where we used `struct` and `enum` data types to add in a new user defined `Card` type. In that assignment you implemented some regular C++ functions to create a standard deck of playing cards that were held in a regular array, and to shuffle the array that represented the deck of cards.

In the previous assignment you got some practice with defining a new data type, and abstracting its information into a structure. However there are several ways that we would like to improve upon from the previous implementation.

In this assignment you will pick up where you left off before. You will start by converting the `Card` structure into an actual class, and adding in and using constructors and member methods to make the `Card` object abstraction even better. And then we will create a new `Deck` object abstraction to represent a standard deck of playing cards. The `Deck` object abstraction will be able to do things like shuffle the deck, and draw cards from the deck so that we can implement simple card games with it.

Overview and Setup

For all assignments, it will be assumed that you have a working VSCode IDE with remote Dev Containers extension installed, and that you have accepted and cloned the assignment to a Dev Container in your VSCode IDE. We will start each assignment with a description of the general setup of the assignment, and an overview of the assignment tasks.

For this assignment you have been given the following files (among some others):

File Name	Description
`src/assg13-tests.cpp`	Unit tests for the tasks that you need to successfully pass
`include/Card.hpp`	Header include file of the Card data type and the Face and Suit enumerated types
`src/Card.cpp`	Implementation of the Card user defined data type member functions
`include/Deck.hpp`	Header include file of the Deck data type declarations
`src/Deck.cpp`	Implementation of the Deck user defined data type member functions

As usual, before starting on the assignment tasks proper, you should make sure you have completed the following setup steps.

1. Copy the assignment repository on GitHub by accepting the assignment using the provided assignment invitation link for 'Assignment 12' for our current class semester and section.
2. Clone the repository using the SSH url to your local class DevBox development environment. Make sure that you open the cloned folder and restart inside of the correct Dev Container.
3. Confirm that the project builds and runs, though no tests may be defined or run initially for some assignments. If the project does not build on the first checkout, please inform the instructor.
4. You should create the issue for Task 1 and/or for all tasks for the assignment now before beginning the first task. On your GitHub account, go to issues, and create it/them from the issue templates for the assignment.

Assignment Tasks

Lab Task 1 / In-Lab Work : Finish implementation of the `Card` class members

First of all, start this assignment by noticing that the structure of the project has changed from all previous assignments. We now have some different header files and different src files in the `include` and the `src` directories respectively. You might want to take some time to look at and read the comments in the `Card.[hpp|cpp]` and the `Deck.[hpp|cpp]` files.

For task 1 we have given you a mostly complete `Card` class. The declaration of the `Card` class can be found in the `include/Card.hpp` header file. We have copied some of the work you did on the previous assignment here for you. Notice that the `Suit` and `Face` enumerated types are already defined in the `Card` header file.

Take a look at the `Card` class declarations given in the `.hpp` file. Notice that a `Card` has 2 private member variables, its `face` and its `suit`. These member variables are **private**, they can be changed by other members of the `Card` class, but someone using a `Card` cannot directly see or change these private member variables. Likewise you should notice that there are two private member functions declared `suitToString()` and `faceToString()`. These member functions are private to the `Card` class.

The `Card` class has 2 constructors, a default constructor that takes no parameters, and a standard constructor where you have to give the `face` and the `suit` values of the new `Card` to be constructed.

The `Card` class also has a few public member functions declared. Public member functions and the class constructors define the API of the new data type you are adding to the language. They tell how you can create new variables of the new data type, and what kinds of methods you can call for an instance of that type.

Notice that all of the member methods for the `Card` class are defined to be `const` member methods. The `const` keyword at the end of a member function declaration means that if that function is called, it will only return information. So none of these methods can change the `face` or the `suit` of a `Card` if they are called.

Next look at the implementations of the constructors and the member methods in the `src/Card.cpp` file. A member function in this file must have the `Card::` before the name of the member function to indicate it is a member of the `Card` class. Notice that all of the functions in this file are member functions of the `Card` class.

We have already completed some of the member functions for you, but several have been left undone, returning stub results, that you must complete to get Task 1 to pass. As usual, state by `#define` the `task1` in `assg13-tests.cpp` file.

We completed the private `suitToString()` and `faceToString()` member functions that you will find at the bottom of the `.cpp` implementation file. Notice that they basically are the same as what you should have implemented in the previous assignment. However notice that we do not pass in a `suit` or a `face` to these functions anymore. These are member functions, so they are using the private member variables of the `Card` class to lookup and return the appropriate `string` representation of this instance of the `Card` when they are called.

We also completed the default constructor of the `Card` already for you. Notice that by default, if not given any parameters, a new `Card` will be created to have a `face` value of `ACE` and a `suit` value of `DIAMONDS`.

However all of the remaining functions do nothing or return some made up stub value. You have to complete the rest of the functions to get the task 1 tests to pass.

1. Complete the standard `Card` constructor

The other constructor currently does nothing, which is not correct. You need to initialize the `face` and `suit` member variables to the given constructor input parameters.

2. Complete the `toString()` member method.

This member methods is similar to the `cardToString()` that you completed before, and also similar to the already completed private `suitToString()` and `faceToString()` member methods below. You need to reuse the two private member methods to construct a string representation of the `Card` instance and return it. Make sure you replace the `return` statement, you should not return an empty string when this function is complete.

3. Complete the `getFace()` and the `getSuit()` accessor methods

We also need these two accessor methods for testing the `Card` class. Notice that, like most getter methods, these are `const` member methods. This means that if someone calls them they guarantee not to change the `Card` state, by for example changing the face value of the card to something else. For both of these methods you need to return the member variable being asked for instead of returning a hard coded value as it currently does.

If you successfully complete the function, and save and compile and rerun the tests, you should find that all of the unit tests for Task 1 pass successfully. When you are satisfied your code is correct, create a commit and push your commit to your GitHub classroom repository. It is always a good idea to check the autograder after pushing a commit to ensure it is also compiling and passing the tests for the task you just completed on GitHub.

Task 2: Declare a Deck of Cards and Construct it

For the second task, we are going to create a new data type, a `Deck`, using a C++ class to represent a standard deck of 52 cards. In the previous assignment you used an array of 52 `Card` items to represent a deck of cards. We will do the same thing here but encapsulate the array of cards inside of the `Deck` class.

This time we have not declared the `Deck` class in the `include/Deck.hpp` header file for you yet. There are a lot of steps you will need to do to get the task 2 tests to run once you `#define` them.

1. Declare a class named `Deck` in the `include/Deck.hpp` header file. You can look at the declaration of the `Card` class for help on the syntax. Don't forget the `;` after the closing curly brace.
2. You need to have a private array of `Card` types. So create a `private:` section in your class and declare an array called `deck` of `Card` objects. This array should be `STANDARD_DECK_SIZE` in size, so that you will have 52 cards in the array and thus valid indexes will be from 0 to 51 for this member variable.
3. Create a default constructor for the `Deck()`. You need to declare the constructor in the `.hpp` file. But also you need to add in the constructor to the `.cpp` implementation file. The constructor will do the same thing as your `createDeckOfCards()` from the previous assignment, so you can copy the code from your previous assignment 12. You need a loop over all of the face enumerations and all of the suit enumerations so that you can initialize each index of your 52 cards in the array member variable to a distinct separate card.
4. Finally in order to test your function we need an accessor method we can call that will return a card at a particular index in your member array of Card items. Here is the declaration of the public member function you need

```
Card getCardAtIndex(int index) const;
```

This should be a constant member function that returns a Card. You give it an index, which will need to be a number from 0 to 51 to correctly index your array of 52 `Card` items. The getter method should simply lookup the indicated card in your private array and return it.

If you get all 4 of those correctly done, you should be able to `#define` your `task2` tests, and they should compile and run. If you look at the `task2` tests, they create a `Deck` of cards, which is where your default constructor will be called. All of the tests use the `getCardAtIndex()` member function to get a card and test it is the expected card. If your default constructor initializes the cards in the array in the correct order, the tests should find the expected cards at the expected indexes and pass the tests.

When you are satisfied your function is working, the project still compiles, and you can run and pass the tests, perform the usual to create and push a commit to the `Feedback` pull request.

Task 3: Shuffle the Deck of Cards

Finally you will implement the ability to shuffle the Deck of cards. Add a public member method to your `Deck` class called `shuffle()`. This member method should not be a `const` member method, because when you call it it will randomly shuffle the array of cards, so the state of your `Deck` will be changed by calling it.

This public member method doesn't take any input and should be a `void` function as it does not return an explicit result.

Shuffle the cards the same way you did in the previous assignment. In fact your code here should be the same as you did before, just as a member method of the `Deck` class now. As a reminder, the pseudocode to perform the shuffle given before was:

We will use the `rand()` function from the C standard library to perform some random shuffling in this function. The `<cstdlib>` is already included for you in the `assg12-library.cpp` file so that you can call this function. We are going to shuffle the deck of cards by doing the following:

1. Iterate over all of the cards in the array from index 0 to index `STANDARD_DECK_SIZE - 1`
    - Randomly generate a number in the range from 0 to 51 to serve as the index of another card in the deck.
    - Swap the card at the index we are iterating with the card at the index we randomly generated.

In the tests for `task3` we perform the same tests as we did for the previous assignment. We create a `Deck` of your cards and we call the `shuffle()` member function to shuffle the deck. Before shuffling we set the random seed. So if you shuffled as described, the cards will be shuffled into the expected random order, which is what is tested in the task 3 tests.

Once you are satisfied with your implementation and are passing the tests, create and push a commit and check that the tasks are passing in the autograder on GitHub.

Bonus / Extra Credit

If you are interested and have time, and if the beginning of the `Card` and `Deck` interest you, it would be easy to add in some methods to start playing some simple games. For example, for a bit of extra credit, you could add in an example of having two players draw a card each at random from your deck of cards, with the high card being the winner. You might want to add in a member function that compares two cards. It would be best to have a member function of the Card class that can check if a card is less than, equal to, or greater in value than `this` card. You would want to handle comparison in some way. For example, usually an ACE is considered as a high card, but in our enumerated type it is defined first so it has an implicit value of 0 in the enumerated type. You cannot change the order of the enumerated type and still get the tests to pass. But you could write a comparison function that does the right thing if one or both cards is an ACE. Also some card games will use the `suit` as a tiebreaker when the `face` values are equal. So you could write a version where you cannot have a tie, and equal face values are settled by looking at the suit to determine the winner.

You could also maybe instead try dealing say a hand of 5 cards, and maybe identifying a hand type. For example could you write a function to identify what type of poker hand you have from 5 dealt cards (a pair, three of a kind, straight, etc.)

If you do extend the `Card` or `Deck` class to try and do some card game tasks, leave a specific comment on GitHub about this for me to consider for extra credit.

Assignment Submission

For this class, the submission process is to correctly create pull request(s) with changes committed and pushed to your copied repository for grading and evaluation. For the assignments, you may not be able to complete all tasks and have all of the tests successfully finishing. This is OK. However, you should endeavor to have as many of the tasks completed before the deadline for the assignment as possible. Also, try and make sure that you only push commits that are building and able to run the tests. You may loose points for pushing a broken build, especially if the last build you submit is not properly compiling and running the tests.

In this assignment, up to 50 points will be given for having completed at least the initial Task 1 / In Lab task. Thereafter 25 additional points are given by the autograder for completing tasks 2 and 3 successfully.

Program Style

At some point you will be required to follow class style and formatting guidelines. The VSCode environment has been set up to try and format your code for some of these guidelines automatically to conform to class style requirements. But not all style issues can be enforced by the IDE/Editor. The instructor may give you feedback in your pull request comments and/or create issues for you for the assignment that you need to address and fix. You should address those if asked, and push a new commit that fixes the issue (or ask for clarification if you don't understand the request). In general the following style/formatting issues will be required for programs for this class:

1. All programs must be properly indented. All indentation must be consistent and lined up correctly. Class style requires 2 spaces with no embedded tabs for all code indentation levels. The editor style checker should properly indent your code when you save it, but if not you may need to check or correct this if code is misaligned or not properly indented.
2. Variable and function names must use `camelCaseNameingNotation`. All variable and function names must begin with a lowercase letter. Do not use underscores between words in the variable or function name. Often function names will be given to you, but you will need to create variables, and maybe some functions, that conform to the naming conventions.
   - Global constants should be used instead of magic numbers. Global constants are identified using `ALL_CAPS_UNDERLINE_NAMING`.
   - User defined types, such as classes, structures and enumerated types should use camel case notation, but should begin with an initial upper case letter, thus `MyUserDefinedClass`.
3. You are required to use meaningful variable and function names. Choosing good names for code items is an important skill. The code examples and starting code tries to give examples of good and meaningful names. In general, do not use abbreviations. Single variable names should be avoided, except maybe for generic loop index variables `i`, `j`, etc. Make your code readable, think of it as writing a document to communicate with other developers (and with your instructor who will be evaluating your code).
4. There are certain white space requirements. In general there should usually never be more than 1 blank line in a row in your code. Likewise there should usually not be more than 1 blank space on a line. There should be 1 blank space before and after all binary operators like `+`, `*`, `=`, `or`.
5. Function documentation is required for all regular functions and all class member functions. You need to follow the correctly formatted Doxygen function documentation format. We will use function documentation generation, and you should be sure your documentation can be built without emitting warnings or errors. Likewise all files should have a file header documentation at the top. You should edit the file header of files where you add in new code (not simply uncommenting existing code). Make sure the information has your correct name, dates, and other information.
6. Practice using proper Git commit messages. You should refer to issues and tasks correctly in commit messages.

Additional Information

The following are suggested online materials you may use to help you understand the tools and topics we have introduced in this assignment.

- Git Tutorials
- Git User Manual
- Git Commit Messages Guidelines
- Test-driven Development and Unit Testing Concepts
- Catch2 Unit Test Tutorial
- Getting Started with Visual Studio Code
- Visual Studio Code Documentation and User Guide
- Make Build System Tutorial
- Markdown Basic Syntax