CSC 211 A5: Data + Methods = Classes: Making a playing card

Background

Today you're going to learn how to write a C++ class from scratch by modeling a traditional playing card.

Classes are an essential programming tool; they allow programmers to abstract away complexities and work more efficiently across large problems. So far, most of what you've done has been **using** classes, blindly ignoring the internal workings of such libraries as `string`, `iostream`, etc. Today, that changes.

Getting Started

- Get into your Docker development environment and *change into the data directory.*
- Download the lab framework with `git clone https://github.com/csc211/card-lab.git`.
- Type `cd card-lab` followed by `ls`.
- You will see this `Readme.md` along with a compile script `compile`.
- You will also see a folder `card`, this is where your source code and header file will go.
- If you try to compile the lab immediately you will get an error: `clang: fatal error: no such file or directory: 'card/card.cpp'`, this is expected as we haven't written anything yet!

First Steps

This section will quickly cover how to lay the ground work for the class. **Read with Caution**

1. Create the header file. Header files are typically formatted as follows: `name_of_class.h` (hint: I'd suggest `card.h`)
2. Now open the new header file.
3. There is one thing all header files should have that we're going to add together: a **directive**. To create this directive, we start with an `#ifndef __X__` statement, where `X` is the name of the class. This statement reads as follows: "If ___X___ is not defined process what follows until you reach `#endif`"; it is a basic **if statement** that prevents the compiler from redefining the class multiple times. Directly after that line, add a formal definition for your class: `#define __X__`, to tell the compiler, "If ___X___ is **not** defined, define it right here". **Note: All references to `X` should be replaced with the name of the class you are building.** Finally,

we must add the `#endif` at the bottom of our file. All of the header code we write will fall in between the `#define` and the `#endif` statements.

The effect of this is not trivial, and will prevent many frustrating errors and weird compiler issues down the line, in more advanced code.

4. Now that the header file is defined, create a source file. Source files are typically formatted as follows: `name_of_class.cpp`
5. In the new .cpp file, you must **include** the header in order to access the declarations you will put within it. To do this you will write `#include "name_of_class.h"` **KEEP THE QUOTES**. The only difference between including your own files, and that of the standard library are these quotes. They tell the compiler that the file you are looking for is local, rather than in a shared library.

   At this point your code **should** compile properly. Please take a moment to check this, and try to work out any potential errors you may face.

6. Now it's time to start building a class!

The card class

The card class has two private pieces of data, and a few distinct methods:

Private data

Each card must store it's *rank*, *suit*, and it's *color*. They can be stored in the primitive type of your choosing, just note that some types are better suited than others for this task.

Methods

Constructors You should be able to build a card in serveral ways.

- `Card()` The default constructor should generate a card with an arbitrary but reasonable set of parameters.
- `Card(std::string color, std::string suit, int rank)` The parameterized constructor should take a color, a suit, and a rank; ensuring that all of these values are valid. If they are not valid, you should print to `std::cerr` "Card not valid"

These are the only two constructors you must implement.

`std::string get_suit()` Returns the suit of the card in a human-readable format: “Diamond”, “Club”, “Heart”, “Spade”

`std::string get_color()` Returns the color of the card in a human-readable format: “Red”, “Black”

`std::string get_rank()` Returns the rank of the card in a human-readable format: “Ace”, “2”, “3”, . . . , “10”, “Jack”, “Queen”, “King”

Comparators The ability to compare cards based on **rank**, with ace being *low* (represented as 1). You should be able to compare whether cards are less than, equal to, greater than, or not equal to eachother based on their rank.

This will mean overloading certain **operators**. Remember from lecture that we can overload an operator as follows. Suppose we wanted to overload `==` for cards, to instead compare based on **suit**:

```
C++ bool operator==(Card c1, Card c2) { return c1.get_suit() ==
c2.get_suit(); }
```

Though remember that we want to compare on **rank**, not suit. So your code will be different.

Requirements

- Define a class called `Card`
- Implement all described methods, including comparators.
- Build 3 test cases within your main, checking your comparators, constructors, and methods.

Hints

Remember that every method gets **declared** in the `card.h` file within the `card` directory, and **defined** in the `card.cpp` file within the `card` directory.

When you **declare** a method, it goes inside the `class Card {}` block.

When you **define** a method, in `card.cpp`, it gets the class name prepended before its name, so the definition of get_rank() should look something like this:

```
C++ std::string Card::get_rank() { // actual contents go here }
```

Mapping rank to name

Consider the following:

“‘C++ #include #include #include

using namespace std;

int main() { vector years = {“Freshman”, “Sophomore”, “Junior”, “Senior”}; cout << years[0] << endl; } “‘

This will print out “Freshman”. Now, what would we want to do differently if we wanted “Freshman” to be year 1, and not have year 0 represent anything?

For more details about C++ classes refer to cplusplus docs.

For a better way to do testing, feel free to check out Catch2 a unit-testing framework for C++.

Submitting

You will submit your code to Mimir, once you are done.