CS 112 - Homework 11

Deadline

11:59 pm on **Friday, December 9**

Purpose

To answer questions on dynamic/late binding and multiple inheritance, to practice lightly with dynamic/late binding, and to write a program using collections of your card class.

How to submit

You will complete **Problem 1** on the course Canvas site (short-answer questions on dynamic/late binding and multiple inheritance).

For **Problems 2 onward**, you will create the specified `.cpp`, `.h`, and `.txt` files on the CS50 IDE, and then submit those to the course Canvas site.

NOTE: While I list the files you need to submit for each problem below, I have set up Canvas to *also* accept `.zip` files.

That is,

- you can submit each `.cpp`, `.h`, and `.txt` file to Canvas.
- OR, if you prefer, you may compress your files to be submitted into a single `.zip` file and submit that `.zip` file to Canvas.

Problem 1 - 14 points

Problem 1 is correctly answering the "HW 11 - Problem 1 - Short-answer questions on dynamic/late binding and multiple inheritance" on the course Canvas site.

Problem 2 - modifying your base and derived card classes slightly to allow for some dynamic/late binding

Consider your C++ class **`PlayingCard`** or **`GameCard`** from Homework 9, that now includes an overloaded == (comparison) operator.

(Note: since you also will be submitting its `.h` and `.cpp` files for this homework, it is fine if you have improved your **`PlayingCard`** or **`GameCard`** class since the version you submitted for Homework 9, as long as it still meets Homework 4's and Homework 9's minimum requirements. Just make sure that the version you submit with this homework works with the class you create here.)

And, consider your derived class you created in Homework 9, Problem 5, that has your card class as its base class.

Modify your base card class *and* your derived card class appropriately so that your base card class' methods `display` and `to_string` that are redefined in your derived card class will now be overridden in your derived card class -- that is, make the (small) changes in their `.h` files so that dynamic/late binding for these two methods might be possible.

(Make sure you update the `last modified:` part of your `.h` files for your base card class and your derived card class, noting that you are adding syntax to make it clear that methods `display` and `to_string` are now going to be **overridden** instead of redefined.)

Problem 3 - seeing some dynamic/late binding in action for your card classes

In a `main` function in a file `dynamic-cards.cpp`:

- Create a vector able to hold at least 8 elements, where each element is of type pointer to your (base) card class.
- Initialize your vector so that:
  - at least four of its elements point to dynamically-allocated instances of your (base) card class.
  - at least four of its elements point to dynamically-allocated instances of your derived card class.
- Loop through your vector, calling the `display` method for the object pointed to by each of its pointers, and also printing to the screen the result of calling the `to_string` method for the object pointed to by each of its pointers.
  - If you made the needed changes in Problem 2, and because these are pointers to dynamically-allocated instances of your base card class and your derived card class, dynamic binding *should* happen so that you see the additional data field values for instances of your derived card class (but not, of course, for instances of your base card class).
- (Add any other statements for things you'd like to do with your vector.)
- Loop through your vector, calling `delete` for each of the dynamically-allocated objects being pointed to.

Problem 4 - using `Rankable` a bit more

Unfortunately, I cannot think of a good reason why your derived card class objects should be rankable, but your base card class objects should not...! So I did not think of a reasonable multiple-inheritance problem in time for this homework.

But, here's a fun fact that I verified: if a base class happens to itself be a derived class -- say from a pure abstract class such as `Rankable` -- then (not too surprisingly!) any classes derived from it inherit the method its base class implemented to avoid becoming abstract themselves. Like a typical inherited method, the derived class can simply inherit it, or it can override it if it wishes.

So: consider the `Rankable` pure abstract class from the Week 14 Lab Exercise. Modify your base card class so that it is derived from `Rankable`, and implement method `compute_rank` for your card class in a way that you think is reasonable for your card class. And, declare method `compute_rank` as `virtual` in your base card class.

Then: given your choice for how `compute_rank` was implemented for your base card class: does that same implementation "work" for your derived card class? If so, that's fine -- your derived card class can simply inherit your base card class' `compute_rank`. But if it does not, then provide an appropriate overridden version of `compute_rank` in your derived card class.

And, make a copy of Problem 3's `dynamic-cards.cpp` in **`dynamic-cards-2.cpp`**, and in your loop calling `display` and `to_string` for each vector instance, also add a statement printing to the screen the result of calling the `compute_rank` method for the object pointed to by each of its pointers.

Problem 5 - a small card competition

Consider: if you had two sets of instances of your card class, you could have them "compete" in some way, even if as straightforwardly as comparing their instances' ranks as determined by `compute_rank`.

To wrap up CS 112, then, design and implement **a simple competition using collections of your card class**.

Here are the requirements for your program:

- **Note:** I highly recommend starting with something **simple**, and get that working first! Then you can try expanding it in interesting directions from there if time permits. **Something simple that meets the following requirements will receive full-credit.**
- You can use your card class, your derived class whose base class is your card class, another derived class whose base class is your card class, or a combination of these.
- It needs to use **at least two instances** of **collections** of your card class,

  where each collection contains at least 3 cards, and

  where those collections are implemented as **at least one** of the following:
  - instance(s) of the `CardPack` class from Homework 7
  - instance(s) of the `CardPile` class from Homework 10
  - instance(s) of vectors of card instances or vectors of pointers to dynamically-allocated card instances
  - instance(s) of dynamically-allocated arrays of card instances or dynamically-allocated arrays of pointers to dynamically-allocated card instances
  - (Why am I giving you so many choices here? Because depending on what kind of competition you choose, one or more of the above might be the better choice for your tasks.)
  - (Why at least two? Because in a competition, there need to be at least two competitors... 8-))
- How will you get your cards for your collections of cards?
  - They can be read from a file.
  - You can ask the user to enter them.
  - They can be automatically generated.
    - You are more than welcome to use function `rand_int` posted along with Homework 3, if that might be useful to you for building a random-ish collection of cards.
- The competition can be *very* simple, but needs to be such that it does not have the same results every time the program is run.
  - (That is, for example, you cannot just always declare the first card set as the winner. 8-))
- If your competition's rounds/parts are zipping by on-screen too quickly to be appreciated, a useful technique is to ask the user to "enter anything to continue", then the program has to read what they type in before it goes on.
- Your competition should also **print to the screen descriptive messages** describing what is happening during each step or round.
  - (These can be very simple, also -- for example, saying that this card beat that card, or saying that this card set had a higher average rank than that card set, etc.)

A few of the many possibilities:

- You could compare which collection has the lowest, or the highest, or the best-average, etc., of some aspect of its cards.
- You could compare each card in each collection to each other based on some aspect, and see which collection "wins" more of these comparisons.

- You could create a shared pile of cards, let each player draw a card from the top, and whosever is "better" by some measure you choose, that player could get to add both to their personal pile of cards. When the shared pile is empty, whosever personal pile of cards is bigger could be declared the winner.
- You could create a set of cards for each player, and each turn each player selects a card or removes one from the top or etc., and then the players' card choices "battle" in some way based on their chosen card's data fields. Perhaps the "wins" are tallied, and after all the cards have been played, a winner is determined.