CS 112 - Homework 7

Deadline

11:59 pm on **Friday, October 21**

Purpose

To answer questions on the "big-3", those additional things a class should include and implement in a class when it involves dynamic memory, and to create a class that includes those "big-3".

Note: you will be using, extending, and modifying the class that you create here in future homeworks.

How to submit

You will complete **Problem 1** on the course Canvas site (short-answer questions on the "big-3", those additional things a class should include and implement in a class when it involves dynamic memory).

For **Problem 2**, you will create the specified `.cpp`, and `.h` 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 am have set up Canvas to *also* accept `.zip` files.

That is,

- you can submit each `.cpp` and `.h` 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 - 12 points

Problem 1 is correctly answering the "HW 7 - Problem 1 - Short-answer questions on the "big-3" a class should include when it involves dynamic memory" on the course Canvas site.

Problem 2 - class `CardPack`

Consider your C++ class **`PlayingCard`** or **`GameCard`** from Homework 4.

(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 4, as long as it still meets Homework 4's minimum requirements. Just make sure that the version you submit with this homework works with the class you create here.)

Create a class **`CardPack`** (in files `CardPack.h` and `CardPack.cpp`) that can be used to create a pack of instances of your card class. It must include at least the following:

- a **`name` private data field**, a name the user assigns to a pack
- a **`size` private data field**, how many card instances are currently in a pack
- a **`cards` private data field**, a pointer to your card class, that will be used to point to a **dynamically-allocated array** of instances of your card class
  - Because an important purpose here is to practice implementing the "big-3" for your `CardPack` class, this must be a dynamically-allocated array -- do **not** use a static array or a vector or a linked list for this data field.

- **at least two constructors**, at least one of which is a **no-argument constructor**, and at least one of which lets them specify a name and a size for the new card pack
  - Make sure each constructor reasonably initializes all of the class' data fields.
  - It is fine if a `CardPack` instance's size cannot be changed after it is created -- but if this is the case, have the no-argument constructor make a pack of a default size (not size 0!) as we did for class `Team`.
  - (It is also fine if a `CardPack` instance's size *can* be changed -- if you make this choice, be sure to carry it through appropriately throughout your class' methods.)
- **a destructor**
- a **copy constructor**
- an **overloaded assignment operator**
- **accessors** for at least getting the calling pack's **name**, the calling pack's **size**, and at least one accessor for **getting a card** from the calling pack based on its position
  - You get to decide whether the accessor for getting a card from the calling pack given a 0-based or 1-based position.
  - You may also have *additional* accessors if you would like (for example, a no-argument accessor that always returns the first or last card in the calling pack)
- **mutators** for at least changing the calling pack's **name**, and for setting a **card** in the calling pack
  - The mutator for setting a card in the calling pack should expect the card to be "put" at that position as well as the position (which you can choose to be either 0-based or 1-based).
  - You may also have *additional* mutators if you would like (for example, if you decide you would like them to be able to change the card pack's size).
- for "other" methods, you need **at least a method `display`** that somehow prints to the screen some readable depiction of the calling card pack's data fields (including the data fields of the cards in the calling pack)
  - IF you want, you may ALSO have an `add_card` method that adds a card to the calling pack, changing the calling card pack's size.

    If you include this method, you get to choose if, along with the card to be added, it also has an argument specifying the position where that new card should be placed.

    (That is, it is OK if you have a one-argument `add_card` method that always adds the specified card to the bottom/end of the calling pack, and it is also OK if you have a two-argument `add_card` method that adds the specified card at a specified position in the calling pack, "shifting" the existing cards appropriately.)

In addition to your files **`CardPack.h`** and **`CardPack.cpp`**, also write a `main` function in a file **`pack-play.cpp`** that includes at least the following actions:

- use each of your constructors at least once in declaring `CardPack` instances
- print to the screen, for each of your `CardPack` **accessors**, at least one result of comparing a call to that accessor to what it should return
  - (For accessor(s) that return a card, you can use compare results of calling your card class' `to_string` method to what the resulting card's `to_string` method should contain, since you probably have not implemented a == operator for your card class.)

  - This would be a good place to compile and run your test-so-far, to see if your constructors and accessors seem to be working!

- write statements calling each of your `CardPack` mutators at least once,

  and print statements to the screen that demonstrate that these mutators did appropriately change their calling `CardPack` object

  - For `set_name`, print the result of comparing the name of the affected card pack to the name it should be if `set_name` worked as it should.
  - For `set_card`, print the result of comparing the results of calling `to_string` for the card at that position in the affected card pack to the value it should be if `set_card` worked as it should.
  - If you add any other mutators, either print the result of a similar comparison that should be true after executing that mutator, or print a message saying what should now be seen if the mutator worked followed by a call of `display` for the affected card pack.
  - This would be a good place to compile and run your test-so-far, to see if your mutators seem to be working!

- To demonstrate your `display` method:

  for each of your `CardPack` instances, print to the screen a message summarizing what should be seen for that card pack, and then call the `display` method for that card pack.

- How can you demonstrate or test your **destructor**, **copy constructor**, and **overloaded assignment operator**?
  - I can't think of a good way to demonstrate or test your **destructor** -- but I will look to see if it appropriately **frees/deallocates** the dynamically-allocated memory in the `CardPack` object passing out of scope.
  - To see whether your **copy constructor** is at least somewhat correct, do the following:

    – write a declaration of a new `CardPack` instance and, **as part of the declaration**, **initialize** it to one of your other `CardPack` instances (so we know this will use the copy constructor).

    – use your mutator for setting a card in a `CardPack` instance in either the new `CardPack` instance or the one you initialized it to

    – then print to to the screen a message noting which card pack has had a card changed,

    and follow that by calling `display` for both of these `CardPack` instances -- they should be mostly the same, except the newly-set card in one should **NOT** also be in the other, if your copy constructor has done its job correctly.
  - To see whether your **overloaded assignment operator** is at least somewhat correct, do the following:

    – write a statement assigning one of your *already-declared* `CardPack` instances to another of your already-declared `CardPack` instances (so we know this will use the overloaded assignment operator)

    – use your mutator for setting a card in a `CardPack` instance in one of these `CardPack` instances

    – then print to to the screen a message noting which card pack has had a card changed,

    and follow that by calling `display` for both of these `CardPack` instances -- they should be mostly the same, except the newly-set card in one should not also be in the other, if your overloaded assignment operator has done its job correctly.

- If you included any other methods, include appropriate demonstrations or tests for those methods. (You can ask me if you are not sure how you might test or demonstrate those.)
- Is there anything else you would like to try with your new class here? Feel free to add that after the above.

Submit your resulting `.cpp` and `.h` files; also include your **`PlayingCard`** or **`GameCard`** `.h` and `.cpp` files.