CS 112 - Homework 10

Deadline

11:59 pm on **Friday, December 2**

Purpose

To answer questions related to exception handling, inheritance, and types, to practice a bit with exception handling, to modify the `Node` class so that its `NodeDataType` uses your (base) card class, and to create a card collection class that, under the hood, is implemented using a linked list of your resulting `Node` class (also providing more practice with the "big-3", the destructor, copy constructor, and overloaded assignment operator methods).

Note: you *may* be using, extending, and/or modifying the classes that you create here in the next homework.

How to submit

You will complete **Problems 1 and 2** on the course Canvas site (short-answer questions on exception handling, inheritance, and types).

For **Problems 3 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 - 18 points

Problem 1 is correctly answering the "HW 10 - Problem 1 - Short-answer questions on C++ exception handling" on the course Canvas site.

Problem 2 - 8 points

Problem 2 is correctly answering the "HW 10 - Problem 2 - Short-answer questions on inheritance and types" on the course Canvas site.

Problem 3 - 17 points

Just to lightly actually program a *little* using exceptions:

1. Look at this list of C++ standard derived exception classes, derived from the C++ standard library `exception` class:

   https://en.cppreference.com/w/cpp/error/exception

2. Select at least TWO of these, being careful to note if one of your choices happens to be derived from the other! (That is, keep in mind the types of an instance of a derived class!)
3. Write a `main` function in a file `exc-play.cpp` that meets the following requirements:

- Ask the user to enter something from among at least three options of your choice.

  - (For example, enter 1, 2, or any other integer; enter a, b, or any other character; etc.)
- Within a `try` block, check what they entered.
  - For one of the options, it should throw an instance of one of your selected derived exception classes from Step 2, calling its constructor with a message of your choice.
  - For another of the options, it should throw an instance of the other of your selected derived exception classes from Step 2, calling its constructor with a different message of your choice.
  - (and so on, for as many selected derived exception classes from Step 2 as you'd like)
  - And for the other/any remaining options, it should simply print a message of your choice saying that it has been reached.
- Follow this `try` block with at least two `catch` blocks catching at least the two specific derived exception classes you chose from Step 2, passing their catch parameters **by reference**,

  carefully ordering them so that you don't put a `catch` block for a base or ancestor class of the other class as the *first* catch block.
  - Each `catch` block should include a message saying that an exception of that derived class was caught, and also print the result of the calling the `what` method for the thrown exception.
- Finally, after the last `catch` block, print a message to the screen saying that it is AFTER the last `catch` block.
- (And as long as you meet the above requirements, please feel free to try additional exception-handling experiments as you'd like!)

Try out your program, and make sure the exceptions are indeed thrown and caught as intended, based on the user's input when prompted.

Submit your resulting `exc-play.cpp` (and any other files it needs, if your additional experiments used any).

Problem 4 - changing `Node`'s `NodeDataType` to your card class - 17 points

Now consider: what if we wanted a linked list of instances of your card class?

We could change the `typedef` of `NodeDataType`, near the beginning of `Node.h`, to have as its type your card class. (Here, use the card class you added overloaded == to in Homework 9 - Problem 4 and used as the base class in Homework 9 - Problem 5, **not** your derived card class from Homework 9 - Problem 5.)

But -- some other changes will now be needed, because in `Node.cpp` and `Node-test.cpp`, we did some things that we could do with `int` expressions that we cannot necessarily do with expressions whose type is a class. And, since the class in this case is one we wrote, we need to make that class visible to the `Node` class.

Start with the posted `Node.h/Node.cpp/Node-test.cpp` from Week 10 Lecture 1 (`Node.h/Node.cpp` should have last-modified dates of 2022-10-25), and `Node-test.cpp` should have a last-modified date of 2022-10-27).

- In **all three** of your `Node.h/Node.cpp/Node-test.cpp` for this homework:
  - make sure **your** name is in all of these files -- add an `adapted by:` line in their opening comments.
  - add a new first date next to `last modified:` , and next to that date note that you are changing `typedef` of `NodeDataType` to your card class (in the same style as we have been doing as we add

functions to the `linked-list-functs.h/linked-list-functs.cpp` files)

Then, in addition:

Node.h

- *After* the `#include <string>` line and *before* the `using namespace std;`, add the `#include` needed to use your card class here.
- change the `typedef` of `NodeDataType` from `int` to your card class.

Node.cpp

- Before the existing `#include` for the `Node` class, add the `#include` needed to also be able to use your card class here.
- Reminder/Fun fact!

  You can assign to an *already-declared* object the result of calling its no-argument constructor by calling that constructor with an empty set of parentheses after its name. That is,

```
PlayerChar bob("Bob", 1, 1.4, "elf", 9);
...
bob = PlayerChar();
```

  - Knowing this, in `Node`'s no-argument constructor, modify the statement setting `data` to 0 to instead set it to the result of calling your card class' no-argument constructor.
- In `Node`'s `display` method, it displays its `data` field using <<:

```
cout << "  data: " << data << endl;
```

  But the << operator does not know how to handle expressions whose type is your card class!

  The << can handle `string` expressions, though! And your card class' `to_string` method conveniently returns a `string`.

  - So, in `Node`'s `display` method, display the result calling `data`'s `to_string` method instead:

```
cout << "  data: " << data.to_string() << endl;
```

- Now that you know how the `display` method needed to be changed to handle a `NodeDataType` of your card class, that should make it reasonable to revise the tests in the `display` method's opening comment block to also involve instances of your card class.

Node-test.cpp

- Note that the compile command for `Node-test.cpp` now needs to include your card class' `.cpp` file! Modify the compile command in the opening comment here to now include this.
- Before the existing `#include` for the `Node` class, add the `#include` needed to also be able to use your card class here.
- The example `Node` objects here now need to have expressions using your card class for their `data` data field, instead of `int` values.
  - So, before each declaration of a `Node` using a multi-argument constructor, declare an instance of your card class to use for that node's `data` data field, and replace the expression in that constructor call with an appropriate argument.
- Because you added an overloaded == operator to your card class, I am *hoping* that replacing the current

expressions in the `get_data` tests with expressions that are the appropriate instances of your card class will work!

- Remember to replace the current expressions in the `set_data` tests with instances of your card class, also.
- And, modify the `cout` statements describing the output for the `Node` class' `display` tests so they describe the expected card data that should be seen.

Make sure the tests in your `Node-test.cpp` program pass!

Submit your resulting versions of `Node.h`, `Node.cpp`, and `Node-test.cpp`, and also your card class' `.h` and `.cpp` files.

Problem 5 - class `CardPile` - 40 points

What if... we were trying to model, rather than a pack of cards, a *pile* of cards? (As in a pile of cards on a table during a game, and players add to or remove from the *top* of that pile only?)

Create a class **`CardPile`**, this particular kind of collection of cards. The cards in this case should be your (base) card class. And, "under the hood", your class will implement this pile of cards using a linked list of your `Node` class that has its `NodeDataType` set to your card class.

A new `CardPile` always starts out empty -- the user may or may not specify a name for a new `CardPile` object.

- The private data fields for `CardPile` must include at least the following:
  - `name`, the name for that card pile
  - `size`, the number of cards currently considered as being in that card pile (really the number of `Node` objects currently in a linked list of such `Node` objects)
  - `cards`, represents the cards currently in that card pile, implemented as a pointer to the first of a linked list of `Node` objects whose data fields contain the card objects currently considered as being in that card pile
    - (but of course, when the `CardPile` is empty/has no cards/has `size` of 0, `cards`' value with be `NULL`!)
  - (You may have additional data fields, but you are expected to have at least the three data fields as named and described above.)
- You are expected to have at least two **constructors** for `CardPile`, each of which appropriately initializes all of the new `CardPile` object's data fields:
  - a no-argument constructor
  - a one-argument constructor in which the user can specify an initial name for the new `CardPile` object
  - (Remember that we're assuming that a new `CardPile` object starts out as an empty `CardPile`, with no cards yet in it.)
- We're going to deliberately limit `CardPile`'s **accessors** a bit, based on the idea of how a pile of cards used in a game might work: frequently, you only access cards from the top of that pile of cards.

  `CardPile` should thus include at least the following accessor methods:
  - accessor methods `get_name` and `get_size` to get the pile's name and size, respectively
  - a method `get_top_card` that expects nothing, and returns the card at the top of the `CardPile`

(although it does not actually remove it from the card pile -- this is an accessor, after all, which should not change the state of the calling object)

- (you may add additional accessor methods if you would like)

- We're going to limit `CardPile`'s **mutators** as well -- the user should not be able to change the pile size data field directly (it should be changed as other methods affect the `CardPile` object's size).

  So, `CardPile` only needs a mutator method `set_name` to allow the user to change a `CardPile`'s name.

  - (If you added additional data fields that it would be reasonable for the user to change, you may add mutator methods for those data fields, though!)

- You should also include at least the following **"other methods"**:

  - `display`, which expects nothing, has the side-effect of printing to the screen the data fields of the calling card pile, and returns nothing (and this method should *not* change the state of the calling `CardPile` object).

  - `add_card`, which expects a card object to be added to the top of the calling `CardPile` object, has the side-effect of adding a newly-allocated `Node` containing that card object at the top/beginning of the calling `CardPile` object, and returns nothing

    - (and note that this method should increase the calling `CardPile`'s `size` data field by one, since a card is being added to the card pile!)

  - `remove_card`, which expects nothing, has the side effect of removing the top/beginning `Node` object from the calling `CardPile` object, and returns the card object in that removed `Node`

    - (and note that this method should decrease the calling `CardPile`'s `size` data field by one, since a card is being removed from the card pile,

    - and don't forget to FREE/deallocate the memory for the removed `Node` object (once you have safely copied the card object from it...!)

- Because `CardPile` will involve dynamic memory -- the `Node` objects in its `cards` linked list are to be dynamically-allocated -- you are expected to include appropriate versions of the "big-3":

  - a **destructor** - that appropriately frees the dynamically-allocated `Node` objects linked to data field `cards` when a `CardPile` object passes out-of-scope

  - a **copy constructor** - that appropriately creates an appropriately-deep copy of a `CardPile` when called by the system, that has its own independent `cards` linked list of `Node` objects

    - Note: you need to be careful with the order of your actions in building this `cards` copy!

    - Tip: don't be afraid to create additional helper-pointers!

    - Tip: sketch out PICTURES of what you want to do as you are working on this!

  - an **overloaded assignment operator** - that appropriately implements assignment of a `CardPile`, being sure to include the parts required here as discussed in-class (for example, checking for self-assignment, freeing the dynamic memory using by the calling `CardPile`'s `cards` linked list of `Node` objects before it is assigned its new value, and ensuring that the calling `CardPile`'s `cards` linked list of `Node` objects is independent of that of the right-hand-side expression/overloaded assignment operator argument)

    - So, yes, this includes freeing the "calling"/left-hand-side operand object's `cards` linked list rather

similarly to how you did in the destructor,

– and yes, this includes making an independent copy of the right-hand-side expression's `cards` linked list rather similarly to how you did in the copy constructor!

– So -- if you would like to make private methods for either or both of these that could be called in both, feel free! BUT that is not required.

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

- Use each of your constructors at least once in declaring `CardPile` instances --
  - (Say, `pile1` using the no-argument constructor, and `pile2` using the one-argument constructor, but it is OK if you give them different names than these!)
- Use method `add_card` to add at least three cards to `pile2`.
  - Then print a description of what should be in `pile2` now, and call method `display` to show if this is indeed the case.
- Print to the screen, for each of your accessors, at least one result of comparing a call to that accessor to either what it should return (or to something that should be true about the value it returns, such as comparing a card's `to_string` results to what they *should* be).
- Use `CardPile`'s mutator `set_name` to set the name of `pile1`, and print to the screen the result of comparing its resulting name to what it should be.
- Use method `remove_card` to remove at least one card from `pile2`, and after each such call print the results of:
  - comparing the resulting size of `pile2` to what its size should be
  - comparing the results of calling `to_string` for the card returned to what the results of `to_string` SHOULD be for the card removed
- To test `CardPile`'s copy constructor a little:
  - (Does your `pile2` still have cards in it, after trying out `remove_card`? If not, add a few back in now.)
  - Declare a new `CardPile` object (for example, `pile3`) and immediately initialize it, in the declaration statement, to `pile2`.
  - Add a new, noticeably different card to `pile3`
  - Print to the screen the results of comparing `pile3`'s size to what it should be
  - Print to the screen the results of comparing `pile2`'s size to what it should be
  - Then print a message saying that `pile3` should have that additional card atop its contents than `pile2` does, and then call `display` for each of these
- To test `CardPile`'s overloaded assignment operator a little:
  - Set `pile1` to `pile3`
  - Add a new, noticeably-different new card to `pile1`
  - Print to the screen the results of comparing `pile1`'s size to what it should be

  - Print to the screen the results of comparing `pile3`'s size to what it should be
  - Then print a message saying that `pile1` should have a different card atop its contents than `pile3` does, then call `display` for each of these

- 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 `CardPile.h`, `CardPile.cpp`, and `pile-play.cpp`.