CS 112 - Homework 9

Deadline

11:59 pm on **Friday, November 18**

Purpose

To answer questions related to inheritance, to write two more linked list functions, and to practice writing a derived class.

How to submit

You will complete **Problem 1** on the course Canvas site (short-answer questions on 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 - 12 points

Problem 1 is correctly answering the "HW 9 - Problem 1 - Short-answer questions on inheritance basics" on the course Canvas site.

Setup for Problems 2 - onward

- FIRST: in the CS50 IDE, in your folder for this homework create copies of the following:
  - `Node.h` and `Node.cpp` from Week 10 - Lecture 1 or 2 (they should be identical)
  - the versions of `linked-list-functs.h`, `linked-list-functs.cpp`, and `linked-list-tests.cpp` from your Homework 8.
- At the beginning of **each** of `linked-list-functs.h`, `linked-list-functs.cpp`, and `linked-list-tests.cpp`, in their opening comment block:
  - add `and` *Your Name* to the end of the `by:` line
  - for `last modified:`, add a new FIRST line (moving the existing original date to the next line) listing your Homework 9's last-modified date and the functions you are implementing for Homework 9.

Problem 2 - another linked-list insertion function

Below are **two** choices for more functions that insert a node into a linked list.

Choose at least **one** of these, and add it to your `linked-list-functs.h` and `linked-list-functs.cpp`.

- **`insert_at_end`**: a function that expects a pointer to the beginning of an **ORDERED** linked list of `Node` instances PASSED BY REFERENCE and a desired new data value to be ADDED to that list, has the side-

effect of adding a new dynamically-allocated node with that desired new data value as its data at the END of this list, and returns the new **size** of the list (as it should **increase** the size of the linked list by one).

- Use the type `NodeDataType` for the desired data value to insert.
- Hint: you need to correctly handle the case when the linked list is empty when this is called.

- **`insert_at`**: a function that expects a pointer to the beginning of a linked list of `Node` instances PASSED BY REFERENCE, a desired new data value to be ADDED to that list, and the relative position in the list where that node should be ADDED, has the side-effect of adding a new dynamically-allocated node with that desired new data value as its data at that position within the list, and returns `true` if it succeeded, and `false` otherwise. (Note that, if successful, this should increase the size of the linked list by one.)
  - Use the type `NodeDataType` for the desired data value to insert.
  - (Hmm; for linked lists, I *think* it makes sense for the positions to be 1-based (head pointer points to the node at position 1, the node at position 1 points to the node at position 2. etc.)
  - The assumption here is that, if you add a new element at position 3, the former element at position 3 becomes the element now at position 4.

    (That is, `insert_at` ADDS a new node AT that position, inserting it such that the node that was previously at that position now follows the new node.)
  - What if the position is "out of range" for the given list? (Although note that it COULD be for a new "last" item! It could be a new 3rd item in a 2-item list, for example.)
    - Do you see that a call to Homework 8's `get_size` function could let you easily check to make sure that the given position is in the bounds of the list? Then you can return `false` and be done if the position is not "in bounds"...
    - ...(but, again, be sure to remember that a position of (size + 1) SHOULD be allowed for `insert_at`, so that one could add to the END of the list.)
  - Hint: you need to gracefully handle several cases here:
    - What if the node to be inserted will be the new first node in a formerly-empty list?
    - What if the node to be inserted will be the new first node in a non-empty list?
    - What if the node to be inserted will be the new last node in a non-empty list?
    - What if the node to be inserted will be neither the first nor the last node in the list?

- In `linked-list-tests.cpp`, add at least the following after the `cout` printing that you are testing your selected function (put your selected function's name in that `cout`!):
  - **`insert_at_end`: IF** you implemented **`insert_at_end`**, then, for **EACH** of the following cases:

    (1) print to the screen the result of comparing the value of a call to `insert_at_end` to what it should return,

    (2) AND THEN, after **each** of those calls, ALSO print a message to the screen listing what values should now be in your list, followed by a call to `print_list` to show that these are indeed now the values in your list:
    - the case where the list is empty
    - the case where the list contains exactly one item
    - the case where the list contains more than one item

- **`insert_at`: IF** you implemented **`remove_at`**, then, for for **EACH** of the following cases,

  (1) print to the screen the result of comparing the value of a call to `insert_at` to what it should return,

  (2) AND THEN, after **each** of those calls, ALSO print a message to the screen listing what values should now be in your list, followed by a call to `print_list` to show that these are indeed now the value in your list:

  - the case where the given position is NOT in the (non-empty) list (nor just one larger)
  - the case where the list is empty and the given position is 1
  - the case where the given position is for a new LAST position, but not the only position, in the list (so, the case where the given position is one more than the list's current size)
  - the case where the given position is for a new FIRST position, but not the only position, in the list (so, the case where the given position is 1)
  - the case where the given position is neither the first nor the last in the list

- Call `delete_list` to free/deallocate the memory for your resulting linked list(s). (Note: you can shift this call to `delete_list` *after* your other function's tests below if you'd like to use your list for testing it/them, also.)

Problem 3 - another linked-list removal function

Below are **two** choices for more functions that remove a node from a linked list.

Choose at least **one** of these, and add it to your `linked-list-functs.h` and `linked-list-functs.cpp`.

- **`remove_at`**: a function that expects a pointer to the beginning of a linked list of `Node` instances PASSED BY REFERENCE and the relative position in the list where a node should be removed and deleted, has the side-effect of removing that node (if there is one at that position) from the list and freeing/deallocating that node's memory, but in such a way that it returns the value in the `data` field of the removed-and-deleted node.

  (Hmm; for linked lists, I *think* it makes sense for the positions to be 1-based (head pointer points to the node at position 1, the node at position 1 points to the node at position 2. etc.)

  - What should be returned if the position is "out of range", if it is not a position in the given list? Since we haven't covered exception-handling in C++ yet, just return a `0` in this case.
  - Do you see that a call to Homework 8's `get_size` function could let you easily check to make sure that the given position is in the bounds of the list? Then you can return `0` and be done if the position is not "in bounds".
  - Use a return type of `NodeDataType` for `remove_at` -- and note that this is a function that will likely need modification if `NodeDataType` is ever changed to a type that does not have `0` as a reasonable value!
  - Hint: you need to gracefully handle several cases here:
    - What if the node to be removed is the first one in the list?
    - What if the node to be removed is the last one in the list?
    - What if the node to be removed is neither the first nor the last in the list?

- **`remove_instance`**: a function that expects a pointer to the beginning of a linked list of `Node` instances PASSED BY REFERENCE and a desired data value to remove, has the side-effects of trying to remove the FIRST node in that linked like that contains the given data value and then freeing/deallocating that node's memory, and returns `true` if it found and deleted a node with that data value and returns `false` otherwise.
  - This one is less awkward in the empty-list case -- it can simply return `false` for that case!
  - Use the type `NodeDataType` for the desired data value to remove.
  - Hint: you may want TWO "temporary" pointers for this function, one pointing to the "current" node you are looking at and one pointing to the "previous" node.
  - Hint: you need to gracefully handle several cases here:
    - What if the node to be removed is the first one in the list?
    - What if the node to be removed is the last one in the list?
    - What if the node to be removed is neither the first nor the last in the list?
- In `linked-list-tests.cpp`, add at least the following after the `cout` printing that you are testing your selected function (put your selected function's name in that `cout`!):
  - **EVERYONE: FIRST**: Print to the screen the result of comparing the value of a call to your function trying to remove from an **empty list** to what it should return.
  - THEN, either use your list from testing Problem 2's function, or create a new linked list of at least the size specified below for the function you chose, and:
  - **`remove_at`: IF** you implemented **`remove_at`**, then, for a list of **at least FOUR** items, for **EACH** of the following cases,

    (1) print to the screen the result of comparing the value of a call to `remove_at` to what it should return,

    (2) AND THEN, after **each** of those calls, ALSO print a message to the screen listing what values should now be in your list, followed by a call to `print_list` to show that these are indeed now the value in your list:
    - the case where the given position is NOT in the (non-empty) list
    - the case where the given position is neither the first nor the last in the list
    - the case where the given position is the LAST position, but not the only position, in the list
    - the case where the given position is the FIRST position, but not the only position, in the list
    - the case where the given position is the ONLY position in the list
  - **`remove_instance`: IF** you implemented **`remove_instance`**, then, for a list of **at least FIVE** items, for **EACH** of the following cases,

    (1) print to the screen the result of comparing the value of a call to `remove_instance` to what it should return,

    (2) AND THEN, after **each** of those calls, ALSO print a message to the screen listing what values should now be in your list, followed by a call to `print_list` to show that these are indeed now the value in your list:
    - the case where the value to remove is NOT in the (non-empty) list

  - the case where the value to remove appears ONCE, but is neither the first nor the last value in the list
  - the case where the value to remove is one that appears at least TWICE in the list
  - the case where the value to remove is the LAST, but not the only, value in the list
  - the case where the value to remove is the FIRST, but not the only, value in the list
  - the case where the value to remove is the ONLY value in the list
- (Your list is probably now empty -- but if you perhaps used multiple lists for your tests instead of just one, call `delete_list` as needed to deallocate any remaining dynamically-allocated memory.)

Submit your resulting files `linked-list-functs.h`, `linked-list-functs.cpp`, and `linked-list-tests.cpp`.

Problem 4 - adding overloaded == to your card class

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.)

We added an overloaded == operator to the class `Point`, and also to its derived class `ColorPoint`, during Week 12.

Now, add an overloaded == operator to your card class, making the appropriate changes in both your card class' `.h` and `.cpp` files.

- Also change the date next to `last modified:` in both your card class' `.h` and `.cpp` files, and next to that date note that you are adding an overloaded == operator (in the same style as we have been doing as we add functions to the `linked-list-functs.h/linked-list-functs.cpp` files).

And in your card class' `-test.cpp` file, add at least the following tests for your overloaded == operator:

- Print to the screen the result of comparing:
  - (using == to compare a card object to itself) to what that comparison should return
- Print to the screen the result of comparing:
  - (using == to compare two different card objects with identical values for their corresponding data fields) to what that comparison should return
- Print to the screen the result of comparing:
  - (using == to compare two different card objects whose corresponding data field values are *not* the same) to what that comparison should return

Make sure these new tests in your card class' `-test.cpp` program pass before going on to the next part.

Submit your resulting card class' `.h`, `.cpp`, and `-test.cpp` files.

Problem 5 - create a derived card class

Consider your C++ class **`PlayingCard`** or **`GameCard`** from Problem 4, that now also includes an overloaded == (comparison) operator.

Imagine a specialized category of cards, that could inherit the data fields and methods of your existing card class -- as long as it has **at least one additional data field**, it will work for this problem.

For example:

- there could be illustrated playing cards, with a `string` data field containing the URL or file name containing that card's image
- there could be bonus cards, with a data field containing a bonus percentage when that card is played
- there could be cards-with-locations, with a data field containing a `Point` giving its current placement on a game board...!
- there could be special- or limited-edition cards, with a data field saying what edition that card is part of

Decide on such a specialized category of your card class.

Then:

- Add copies of the current version of your C++ class **`PlayingCard`** or **`GameCard`** to your CS50 IDE folder for this problem.
- Create your derived card class in its own `.h` and `.cpp` files, using your card class as its base class, making sure your derived card class includes at least the following:
  - at least one no-argument constructor
  - at least one appropriate multi-argument constructor
  - at least one additional `private` data field
  - at least one `public` accessor method for that additional data field
  - IF appropriate for your particular additional data field, include a `public` mutator method for that additional data field
  - a **REDEFINED** version of `public` method `display`, now also printing the additional data field(s) of this derived card class instance
  - a **REDEFINED** version of `public` method `to_string`, now also including a string depiction of the additional data field(s) of this derived card class instance
  - a version of the == operator that expects an object of this derived card class and returns whether the calling object has the same data field values, including the additional data field(s) for this class, as the given derived card class object
  - (and if you would like additional specialized `public` or `private` methods, or additional overloaded or redefined methods, that is fine and encouraged!)

Submit the `.h` and `.cpp` files both for your base card class and for your derived card class.

Problem 6 - try out your derived card class

Write a `main` function in a file **`derived-play.cpp`** that includes at least the following actions:

- Declare at least two instances of your derived card class, using each of your at-least-two constructors.
- Print to the screen, for each of your derived class' accessors (new *and* inherited!), 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).
- Try out each of your derived class' mutators (inherited *and*, if you added any, new), each followed by

printing to the screen the result of comparing what the changed data field is to what it should be (or to something that should be true about its changed value).

- Print to the screen a message describing what should be seen next, followed by a statement calling **REDEFINED** method `display` on one of your derived card class instances, to show that it includes the additional information for your derived card class.
- Print to the screen the result of comparing a call to **REDEFINED** method `to_string` on one of your derived card class instances to what it should now be, to show that, again, it includes the additional information for your derived class.
  - then just print the result of that call to **REDEFINED** method `to_string` on that derived card class instance to the screen.
- Demonstrate your derived card class' == operator as follows:
  - Create additional instances of your derived card class such that:
    - at least one has **definitely different** data field values than another of your derived card class instances
    - at least one has the **same** values for each of their data fields
    - at least one has the **same** "inherited" data field values but a **different** value for at least one of the data fields particular to this derived class
  - Print to the screen the result of comparing:
    - (using == to compare a derived-class card object to itself) to what that comparison should return
    - (using == to compare two different derived-class card objects with identical values for their corresponding data fields) to what that comparison should return
    - (using == to compare two different derived-class card objects whose corresponding data field values are *not* the same) to what that comparison should return
    - (using == to compare two different derived-class card objects whose corresponding "inherited" data field values are the same, but that have a **different** value for at least one of the data fields particular to this derived class) to what that comparison should return
- THEN -- to show that you can! -- print a message to the screen saying that you are about to call your **base card class' version** of `display` for one of your derived card class instances, and then do so.
  - (remember, this method's result will *not* show the additional information for your derived class)
- AND -- to show that you can! -- print a message to the screen saying that you are about to print the result of calling your **base card class' version** of `to_string` for one of your derived card class instances, and then do so.
  - (again, remember, this method's result will *not* show the additional information for your derived class)
- 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 derived class here? Feel free to add that after the above.

Submit your resulting `derived-play.cpp`. (You should have already submitted your `.h` and `.cpp` files both for your base card class and for your derived card class as part of previous problems.)