Programming Project 3: I'll link to like if linking liking move

Overview:

This project will have you implement a templated doubly linked list class, which has both a head and tail pointer, and each node has a pointer to the previous and next nodes. Additionally, we will also keep track of the number of nodes in the list, or the list's size.

list.h:

You will need to implement the member functions for a doubly linked list class. This list (all linked lists for that matter) will be very similar to what we discussed in lecture. So, clearly understanding the how and the why behind the lecture's linked list implementation will go a long way in completing these functions. I cannot emphasize enough the importance of drawing/planning everything out before attempting to implement any of the functions, otherwise you can easily put down erroneous code. Any program with extensive use of pointers can be extremely difficult to debug after the fact, prevention is generally the best approach to anything, but is even truer in this case.

List's member variables are provided to you. There are two Node pointers, head and tail, and an integer, num_nodes. Recall that for any object, the member variables must be consistent with the intended state of the object, for this linked list class that means:

- If there are Nodes in the list the head must always point to the first Node, or nullptr if the list is empty
- If there are Nodes in the list the tail must always point to the last Node in the list, or nullptr if the list is empty
- num_nodes must always be the same as the number of Nodes in the list.

Additionally, since Nodes are referred to through your head and tail member pointers, they too must be consistent with the intended state of the object.

- The value stored in each node is what the user provided.
- The previous pointer points to the previous Node, or nullptr if the node is the first Node.
- The next pointer points to the next Node, or nullptr if the node is the last Node
- The order of the Nodes are what the user intended through the relevant add functions. For example, if a node is added to the list using the push_front member function, then the result of this operation must have that Node before all others.

Your implementations must keep these invariants consistent with the intended state of the list when your functions complete. This is different from the lecture's linked list where there was only a single head pointer and Nodes only had a next pointer. Some of this may seem obvious on the inset, but you will be surprised how often we forget the ensure these consistencies, or even ignore them because we assume what we have is correct.

The Node class is strictly internal to the List and so defined within the list itself, its definition is complete and nothing more is needed in its implementation.

list's functions:
In no particular order:

1. List(): Default constructor. This should construct an empty List, the member variables should be initialized to reflect this state. This function is already fully implemented.
2. List(const List<Type>& other): Copy constructor for the linked list. This should create an entirely new linked list with the same number of Nodes and the Values stored these Nodes in the same order as seen the other list's Nodes. This should not result in any memory leaks or aliasing.
3. List<Type>& operator=(const List<Type>& other): Overloaded assignment operator for the linked list. Causes the already existing linked list to be identical to the other linked list without causing any memory leaks or aliasing.
4. ~List(): Destructor. The list dynamically allocates nodes, that means when we destruct your list we need to ensure we deallocated the nodes appropriately to avoid memory leaks.
5. void print() const: Traverses the list and prints the items in the list in a single line with spaces in between each item. There is no space before the first word and after the last word. There is a newline after all items have been printed. For example suppose our list contains the strings "Cash", "Shell", and "Ruby", print will display in the console exactly:

   Cash Shell Ruby
6. bool empty() const: returns boolean value indicating if the list is empty or not.
7. void push_front(const Type &item): Adds item to a new Node at the Front of the list. Updates head, tail, and size accordingly. Must appropriately handle cases in which the list is empty and if there are nodes already in the list.
8. void push_back(const Type &item): Adds item to a new Node at the Rear of the list. Updates head, tail, and size accordingly. Must appropriately handle cases in which the list is empty and if there are nodes already in the list.
9. void add_at(int index, const Type &item): Given an index, this function adds the item to a new Node at the index. Updates head, tail, and size accordingly. If the index is less than or equal to zero add the item to the front. If the index is greater than or equal to the size of the list then add it to the rear. Otherwise add the item at the index indicated.
10. Type front() const: returns the item in the Node at the front of the list without modifying the list. The function cannot be called if the list is empty.
11. Type rear() const: returns the item in the Node at the rear of the list without modifying the list. The function cannot be called if the list is empty.
12. Type get_at(index item) const: returns the item in the Node in the index place of the list.
13. int size() const: returns the number of nodes currently in the list.
14. int find(const Type &item): Searches the list to see if the item is currently in the list. If it is, the function returns the index of the item, otherwise it returns -1;

15. bool pop_front():Removes the first item in the list, returns true if the item was deleted, false otherwise. Updates head, tail, and size accordingly. Must appropriately manage cases where the list is empty or has one or more items.
16. bool pop_rear() : Removes the last item in the list, returns true if the item was deleted, false otherwise. Updates head, tail, and size accordingly. Must appropriately manage cases where the list is empty or has one item, or has two or more items.
17. bool pop_at(int index): Removes the item at the index of the list, returns true if the item was deleted false otherwise. Updates head, tail, and size accordingly. Must check to see if the index is inbounds.

Many of these functions have assert statements to check preconditions, these are mostly for your own debugging, my tests will never violate preconditions. That is to say you will mostly just leave those alone and add your code after the asserts. In the event that you trip one of the asserts in your development, you will know that you're doing something wrong and will have information on where and why, that way you can go through the process of debugging.

studentinfo.h:

There are only two functions here where you return your name and ID, they are used in the automated testing. Please make sure you modify these functions to do what they intend.

main.cpp:

Where you test your work. This will not be submitted.

Submission:
For this project you submit 2 files:

list.h studentinfo.h

You will have your own main.cpp for testing, but do not include it with your submission. Combine everything into a zip file. Note that when you resubmit on canvas it will postpend your file name with a number indicating its submission order, this is ok. If I take these files, I must be able to compile them using VS2017/2019 without any errors .
Tips:

- ***Start Early!*** Do not wait until the last minute to work on this; you will run out of time. There is relatively little benefit in trying to complete the project in a single sitting. It is better in terms of time management and also more conducive to learning (due to the way the brain works) to pace out your efforts over a period of days rather than cramming it all in one go.
- As always, once you put these files into your project make sure everything compiles without issue before making any changes. For ease of testing and development you can make your member variables public. Of course, don't forget to change it back to private prior to submission.
- Compile and submit. The code that is provided to you should compile without error given an empty main. You should make sure you can do so before doing anything else. When you make significant headway, make sure what you have compiles and then submit it, that way if for any reason when you hit

the deadline and you can't compile you at least have the previous working code to fall back on. Never submit anything that does not compile. Code that does not compile is worth less than code that does but has half the amount of work; in industry it is worth nothing.

- I would recommend starting with the push_front and print functions, that way you have a way to manipulate the list and view your changes.
- Have test cases before executing any code. For any function, in planning how you want to implement that function, spend time jotting down possible ways to test or break that function. Make sure you have tests that visit all areas of code, meaning if you have if-else statements, you have a test case that will enter each of the branches. Not only that, make sure you know ahead of time what it is you expect to happen based on what you understand the function should be doing. That way you can compare "What is actually happening" with "What you intend". An unfortunate fact-of-life for programmers is that these two are seldom the same. It takes considerable practice to be able to effectively delineate between the two notions; we have the tendency to bias one with the other.
- For each specific case you are considering, it is helpful to draw before and after diagrams. That is to say, what the initial state of the linked list looks like and then the resulting state after whatever operation is under consideration. With this you can analyze the steps required to go from the one state to the other. After you have the steps, you can translate into programming syntax.