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 (DLList), 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.

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.

DLList's member variables are provided to you. There are two DLNode pointers (also refered to here as Nodes), head (front) and tail (rear), and an integer to track the total number of nodes in the DLList, size. 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 front must always point to the first Node, or nullptr if the list is empty
- If there are Nodes in the list the rear must always point to the last Node in the list, or nullptr if the list is empty
- size must always be the same as the number of Nodes in the list.

Additionally, since Nodes are referred to through your front and rear 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 is what the user intended through the relevant add functions. For example, if a node is added to the list using the insert_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 front pointer and Nodes only had a next pointer. Some of this may seem obvious on the outset, 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 , or specifically, DLNode, 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. The DLNode's member variables are private, and must remain so, therefore you must use the relevant member functions to either read-from or modify those member variables.

Provided Files:

dlList.h: Templated doubly linked list. None of your implementations should result in undefined behavior, memory leaks or terminate program execution given valid usage (all tests done by me will be valid usage.)

1. DLList(): 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. DLList (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 Items 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. DLList <Type>& operator=(const DLList <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 values in the list in a single line with spaces in between each value. There are no spaces before the first word and after the last word. There is no newline after all values have been printed. For example suppose our list contains the strings "Cash", "Shell", and "Ruby", print will display in the console:

   Cash Shell Ruby
6. bool empty() const: returns boolean value indicating if the list is empty or not.
7. void insert_front(const Type &value): Adds value to a new Node at the Front of the list. Updates front, rear, and size accordingly. Must appropriately handle cases in which the list is empty and if there are nodes already in the list.
8. void insert_rear(const Type &value): Adds value to a new Node at the Rear of the list. Updates front, rear, and size accordingly. Must appropriately handle cases in which the list is empty and if there are nodes already in the list.
9. void insert(int idx, const Type &value): Given an index, this function adds the value to a new Node at the index. Updates front, rear, and size accordingly. If the index is less than or equal to zero add the value 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 value at the index indicated. Indices begin at zero.
10. Type peek_front() const: returns the value 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 peek_rear() const: returns the value 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 peek(int idx) const: returns the value in the Node in the index place of the list. Indices begin at zero.
13. int count() const: returns the number of nodes currently in the list.

14. int count_value(const Type &value): Counts the total number of values in the list that match the given value.
15. int find(const Type &value): Searches the list to see if the value is currently in the list. If it is, the function returns the index of the value, otherwise it returns -1. Indices begin at zero.
16. bool remove_front():Removes the first value in the list, returns true if the value was deleted, false otherwise. Updates front, rear, and size accordingly. Must appropriately manage cases where the list is empty or has one or more values.
17. bool remove_rear() : Removes the last value in the list, returns true if the value was deleted, false otherwise. Updates front, rear, and size accordingly. Must appropriately manage cases where the list is empty or has one value, or has two or more values.
18. bool remove_index(int idx): Removes the value at the index of the list, returns true if the value was deleted false otherwise. Updates front, rear, and size accordingly. If the indices are out of bound remove_index does nothing and returns false. Indices begin at zero.
19. int remove_value(const Type &value): Removes the first occurrence of the value relative to the front of the list, returns the index of the removed value. Return -1 if the value is not found. Indices begin at zero.

Some of these functions have assert statements to check preconditions, these are mostly for your own debugging, my tests will never violate preconditions. You will 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.

Do not add further asserts in any of your DLList member functions implementations, they will most assuredly violate the specification, leading to those function implementations being incorrect.

If you use console printing as part of your informal testing, remove all such statements prior to submission. A good practice is to use cerr as opposed to cout for these purposes.

Other Files:

studentinfo.h: There are only two functions here where you return your name and ID, they are used in the automated testing. Make sure you modify these functions to do what they intend. Replace the entire string so that no special characters remain, e.g. ':' or '#'.

main.cpp: Where you test your work. There will be several different mains demonstrating some usage. Take particular note of the unit test examples. This will not be submitted.

Listtest_s23.h + debugmem.h…: Unit tests DLList and memory tracer code seen in project 2. These will not be submitted.

Submission:

For this project you submit only 2 files:

dllist.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.

Your submission must minimally compile in Visual Studio C++ 2019 or newer. As always, any project that does not compile under the aforementioned conditions, will receive a zero.

Do not submit anything other than what is specified, notably any environment specified files. This excludes and files created by your compression software.

DOUBLE CHECK YOUR SUBMISSIONS.

Tips:

- ***Start On Time!*** 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.
- After you make your final submission, re download that submission and see if that code compiles. Do not just run the project you already have. The point is to see if the upload was corrupted and to make sure you uploaded the files you intended to upload.
- I would recommend starting with the insert_front and print functions, that way you have a way to manipulate the list and view your changes.
- 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.