Project 2

GOALS

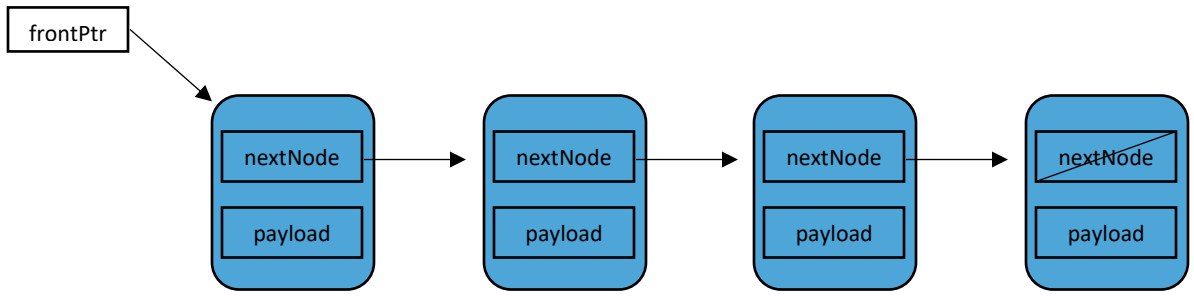
The primary goals of this project are to increase our familiarity with raw pointers in C++ and learn how they may be used to construct an important data structure, the linked list. Implementing a linked list requires careful manipulation of pointers; by the end of this exercise, you may find that you have a much finer-grained feel for their nature and a hard-won acquiantance with some of the related pitfalls. We will also use the opportunity to gain more experience with class design, writing two classes that work together while separating interface from implementation.

This assignment will be due to Canvas by 6 p.m. Central Time on Monday, November 16th, 2020.

INSTRUCTIONS

For this assignment, you will build an ordered, singly-linked list, along with several methods for interacting with the list. A driver program is attached to the assignment posting on Canvas (including the resulting output commented-out beneath it); your code should work with this driver.

You'll recall from your previous programming experience that a singly-linked list is a data structure comprised of separate nodes, where each node contains a pointer to the next node in the list. The list itself must maintain a pointer to the first node in the list:



Prior to any operation on the list, the pointers must be traversed to the desired position. To insert or remove a node, the appropriate pointers must then be manipulated in such a way as to achieve the desired outcome. With any operation, it is important to consider the different possible scenarios (e.g., is the insertion taking place at the front of the list, or somewhere in the middle, or at the end?)

Your program must include an `OrderedList` class and a `Node` class. Like the driver, a copy of a valid `Node` class is also provided for you on Canvas. Instances of the `Node` class will be stored in the `OrderedList`, and each `Node` should contain a payload of type `std::string`.

As you'll see when you examine it, the Node class has the following features:

1. A pointer to the next Node in the OrderedList.

2. A payload of type std::string.

3. A constructor that accepts a std::string representing the payload of that Node.

The Node class also has a couple of items we have not seen yet.

First, it has a "forward declaration" of the OrderedList class, which allows the Node class declaration to refer to the OrderedList class a few lines later without the compiler objecting that it has never heard of an OrderedList. (This is another example of a declaration that is not a definition.)

Second, it declares that the OrderedList is a "friend." In C++, classes can declare that they have certain friends, which means that the specified "friend" class can directly access the private members of the class. Some see this as an end-run around object-oriented principles; some see it as a useful convenience. You can be the judge of how you feel about it, but it's good to be able to recognize how this works. Here, it allows the OrderedList class to directly access the payload and nextNode fields of a Node object, which arguably helps reduce code clutter and avoids the overhead of a method call.

The OrderedList class, which you will write, must have the following features:

1. A pointer to the first Node in the OrderedList.

2. A default constructor, which simply creates an empty OrderedList.

3. An insert() method, which takes in a Node by value (i.e., a copy) and inserts an equivalent Node into the list (i.e., a Node containing the same payload). When a Node is inserted, it should be placed into the correct position, as determined by the ordering of the payloads. Remember that you can compare std::string objects simply by using the <, <=, >, >=, and == operators.

4. A remove() method, which takes in a Node by value, finds the equivalent Node in the list (if it exists), removes it and returns a copy. Remember that in order to avoid memory leaks, there must be a call to delete aimed at any memory previously allocated with new.

5. A removeFront() method, which removes the first Node in the list and returns a copy.

6. A clear() method that removes all nodes from the list.

7. An absorb() method, which takes in another OrderedList by reference, removes its nodes, and inserts them into the caller.

8. A printOrderedList() method that prints the payloads of each Node in the OrderedList.

9. A destructor, which removes all nodes from the list. For this assignment, you should make a "noisy" destructor: it should announce that it has been invoked, print out the payload of each Node it is about to destroy, and announce when it has finished.

Note that the `OrderedList` class must be implemented as a linked list – it should not be an array, vector or other such container (and you may not just use a linked list class from the Standard Library). **All Node objects must be allocated on the heap (i.e., using the `new` operator).**

You must have all of the above methods in your `OrderedList`, but you may include additional methods if you wish (for example, you could make an `isEmpty()` method to test whether a list is empty).

If an attempt is made to insert a node that has a payload matching a node that is already in the `OrderedList` (i.e., a duplicate node), that second node should be ignored and not inserted into the list. In other words, duplicate nodes should just be ignored (as shown in the driver program).

SUBMITTING YOUR WORK:

Once your program is able to compile and run, producing correct results without logic errors, you should submit your files (source code and headers) to the course's Canvas page. In order to grade your submission, I will need to be able to open up your source code and examine it.

As described on the Canvas page for this assignment, this is a team project. You may of course collaborate with your teammate. Only one submission is required per team.

Again, make sure your `OrderedList` is able to work with the provided driver file, producing the same results.