

Recitation 12: To make a linked list class

Topics:

- Linked lists
- Packaging a data structure as a class
- Iterators
- Copy control

Design decisions.

Our goal is to make a linked list class.

Singly or doubly linked? We will go with **doubly linked**. Why? It will simplify issues like how or where to add or remove a given item.

Some of our considerations will be to make it compatible with various C++ features, e.g. the ranged for and the STL (Standard Template Library) algorithms library. Providing `begin()` and `end()` methods that return iterators will buy us a lot.

However, we will also provide some functionality that the STL list class does not. After all, this is *our* class.

Sentinels: I am not specifying whether or not to use sentinel(s). I will leave that up to you. Likely you will choose to do whatever you did back in your data structures course. Please note, that your code will generally be easier to write if you do use sentinels.

Nodes: The client of the List class will never directly interact with the Node type. Therefore none of the public methods will return or be passed a Node or Node pointer.

Task One

This class will naturally have a private struct Node. If we are constructing a doubly linked list, then the Node type will have not only data and next fields, but also a prior field. Be sure that all fields get properly initialized.

We will start by providing a simple class that supports:

- `push_back(int data)`
- `pop_back()`

- `front()`. Remember that `front()` allows you to change the value stored there, if the list is not `const`.
- `back()`. Like `front()` should allow modifying a non-`const` list.
- `size()`. This should, of course, be a constant time operation. This is one of the advantages to having the list represented by a class.
- An output operator

Modifying the list, whether adding or removing, will take a little more work than with a singly linked list. Don't be surprised if you have some segmentation faults...

Test out your functions. You should create a few lists, adding and removing items verifying that the contents are correct. (I have attached some test code. But perhaps you can come up with additional tests!)

Task Two

Now that you have a reliable list class, add the following methods and test them:

- `push_front(int data)`
- `pop_front()`

Task Three

Naturally it is necessary for a list to have clear method. Implement a clear method and test it.

Task Four

It is nice that we can see what's in the list, using the output operator, but we still can't access any of the items, other than the first and the last.

Let's solve that first with an index operator. With that and our size method, we can write a loop to print out the contents of a list, Write another loop to modify the value of each element and finally print out the list again.

Of course those loops are *horrendously* expensive!

Task Five

Ok, but wouldn't it be cool to be able to use a ranged for? We did that fairly easily when we implemented the Vector class. It will take a bit more work with a list. Why?

First, remember that we need to provide methods `begin()` and `end()`, and that their return values have to support, among other things, `operator++`. That was easy with `Vector`'s because we could just return the address of the first element for `begin()` and the address just past the last element for `end()`. However with the `Node` objects in our list, they are not laid out contiguously as the items in a `Vector` are. That has the result that incrementing the address of one `Node` does not take us to the next.

What can we do?

[Heads up, this is a bit of a big one...]

We will create [yet] another type, called an iterator.

Iterators can be really pretty simple. They will need a single field, a `Node*`. It is possible to also have them "know" what list instance they belong to, but we will leave that out for now.

In addition, iterators should support a few operations:

- `++`
You can stick with just a pre-increment operator, if you like.
- `--`
Again, you can stick with just a pre-decrement operator.
- `==`
Iterators will be equal if they are pointing to the same `Node`.
- `!=`
- `*`
The *dereference* operator.

The Iterator also needs to provide suitable constructor(s).

Copy constructor: While discussing constructors, we should consider the copy constructor. Our iterator type has a pointer to an item, a `Node`, that exists on the heap. What does that suggest? *Not* that we need copy control. Why not? Because the iterator does not *own* the `Node`. So we will *not* be doing a deep copy, and in fact do not need to implement a copy constructor. The one the system provides is good enough.

We can't really test or use this class unless we have a way to get hold of an iterator for a list. The main way to do that will be through two methods in our list class `begin()` and `end()`. [You may remember that the STL's vector type supports those methods.]

The iterator that the begin method returns will obviously have a pointer to the first Node. What about the iterator returned by end? Given the *use cases* we have discussed so far, a simple design that can work would be to have the end iterator have a nullptr. We may see later that there could be useful reasons to do otherwise, but this will certainly work for now.

What can we do with our list class now? Repeat the activities you did with the index operator, but this time using iterators.

Task Six

We still haven't done much to justify defining a list, let alone a list class. What's missing?

Provide an insert method for the list class. What should it be passed? The data item, certainly. And how do we know where to put the data item? Also pass in an iterator! Given an iterator for the class it should be easy enough to insert an item. But we first need to agree on where we are going to put something, given the provided iterator. Do we put it before or after?

Well, how will we insert something at the beginning of the list? The only way we have to get there is the method begin() which hands us an iterator pointing at the first item. So if we want to add something to the head of the list using these iterators, we will need to have our insert method insert *before* the iterator.

Insert should return an iterator to the new item.

One last point!!! The insert code will need to access the Node that the iterator is pointing at. Should iterators have some method to return the Node? Remember that iterators are used by code outside the List class and that code doesn't want to / shouldn't have to know about Nodes. But the List sure does! What's the point? List methods need to be able to know what Node an iterator "points" to. How? One way is to have the iterator class make the List class a friend. Yes, **you can make a whole class a friend!**

Task Seven

Add an erase method to the class. What should it be passed? Again, an iterator! This time, there isn't any question about "where" to remove, as we will remove the Node that the iterator is pointing to.

Erase should return an iterator to the item that came after the one you erased.

Task Eight

Copy control! Yes, it we should be able to pass a list by value and have that result in a copy. We should also not have to worry about all those Node objects getting lost and creating a memory leak. And it should be possible to assign one list, the source, to another, the target, causing the target to be replaced by a copy of the source.

Implement the usual big three and demonstrate that they work.

Observations



We now have a fairly useful linked list class. However there are still a few things it would be nice to have. You've done enough for today, but you should take some time later to consider the following points:

- `const`
You may have noticed that we did not discuss `const` anywhere along the line. Ideally, we want to have a notion of iterators that can be safely used on constant lists, e.g. lists that were passed by constant reference. Basic C++ notation does not give us a way to indicate whether an instance of iterator type would be able to modify the contents of the list. For that we will need an additional type, say, `const_iterator`, that could move over a list but not be used to change anything.
- Our representation of the iterator that `end` returns doesn't work quite the same way that the one that the STL provides does. Can you identify any differences?
- Our list is not templated. How much effort will it be to turn this into a generic class?
- We have not yet experimented with the `<algorithm>` library. How well does our code work with them?

Attached

I am attaching some test code to help you get started.

Additional resources for assignment

-  [output.txt](#) (5 KB; Jan 2, 2018 2:39 pm)
-  [rec12-test.cpp](#) (5 KB; Jan 2, 2018 2:39 pm)