## Document

I felt like my other document was getting really long, and that I would have a lot of specific learning associated to this project so I made a new one.

I did not comment my code very much for this project. This was on purpose. It was becoming cluttered and I was changing types so frequently as I went that it was slowing me down greatly. I was also going to have so many “leanings” that it was going to be unreadable.

Instead I have decided to type of the things I learned here, and let the code speak for itself. That is not to say I have no comments, I made them where I felt further explanation was needed, but they are not to the level I know you prefer to have for your assignments and I wanted to note that I did this on purpose.

## Credit due

I want to mention that I developed a lot of these functions by declaring an stl list and inspecting its behavior. FO the most part I did not look at the underlying code however to avoid the tendency to straight copy it. I do however copy their form, as that was part of the exercise, to produce my own container that functioned as an stl list. I looked at both MS visual c++ and the gcc g++ source.

## The power of basic\_node

This allows me to not template some of the node and node iterator portions of my nodes! Cool! Imagine the fractions of kb this will save! It saves space because the iterator code will not be regenerated for each template that utilizes it.

Basic node and basic node iterator only know about one thing, the next thing. Within the List class I “upcast” the basic node to reveal the templated item field. This pushes all of the knowledge about item into list node and out of basic node so a template is not needed.

I didn’t really get the idea from one place so much as several sources on the internet, but I implemented the code from scratch after reading. I also inspected the g++ source libraries on my computer so I guess that would be a source to cite as well.

## Iterator and Node as part of the List class

The iterator proved to be challenging, certainly more so than I expected. I was repeatedly faced with issues that arose form primarily mis-declarations of the constructor and improper use of constructor chaining.

I had also originally declared Node as a separate class but this caused issues when trying to access the Node pointers form within list so I pushed it into list.

## “Belt Buckle” versus NULL

I switched to a “belt buckle” implementation after running into issues with the NULL terminated list and inserting at the head. The goal was to implement a single insert statement that would work for all insertion locations without having to test for isHead(). By implementing insert using iterators and properly defining end and begin to skip the “buckle”, and adding an accessor for the iterator node by reference I was able to accomplish this (finally).

## Const-ness and const pointers

You can only invoke functions declared as const on const members of that class. It took me forever to figure this out when fixing the operator=(const& target) to actually use a const reference. In my previous submission I had forced my way around this with a const\_cast. Now I know how to do it properly!

It finally clicked when I read this post about a size() member method someone was having issues with:

<http://stackoverflow.com/questions/12708689/cannot-convert-this-pointer-from-const-containert-to-containert>

Further, I found a great breakdown on const pointers here:

<http://stackoverflow.com/questions/4949254/const-char-const-versus-const-char>

type\* p: I can change the item to which p points, and I can modify the item at which it points.

const type\* p: I can change the item to which p points, but I cannot modify the item at which it points.

type\* const p: I cannot change the item to which p points, but I can modify the item at which it points.

const type\* const p: I cannot change the item to which p points, nor can I modify the item at which it points.

## nullptr

There is a constant in the std call nullptr. It has implications when doing type checking. One of the more useful explanations I found was this post:

<http://stackoverflow.com/questions/13816385/what-are-the-advantages-of-using-nullptr>

It makes a lot of sense to include this. I come from a C background and we did not the std, so it is obvious to me that NULL is as a pre-compiler directive and hence will resolve as an int because I had to write the #DEFINE statement, but I don’t think that is the norm, and I think it is easy to forget, so introducing a type to handle null pointers was a smart move.

## Auto

C++11 adds support for an auto, which is similar to var in c#. It allows the compiler to deduce what type to use for your variable based on how you initialize it. IT’s not really “needed” but is one of those things that newer language are adding because it makes it “easier” to code.

“Specifies that the type of the variable that is being declared will be automatically deduced from its initializer. “

From <http://en.cppreference.com/w/cpp/language/auto>

I didn’t use it though, because it’s for c++11 only.

## Explicit constructors and conversion constructors

Iw as having a lot of trouble decalring my delegate contrsuctors from list iterator, down through node, and into basicnode and at somepoint I came across a post on the explicit keyword for cosntructors.

I didn’t use them in anything, but I still found it very interesting.

<http://en.cppreference.com/w/cpp/language/converting_constructor>

## Doubly linked list and insert times

My implementation of list is singly linked. As a result my insertions are O(N) on the base list because I have to iterate from begin to the current node to find the previous node with which to perform the insert. This means that my push for the priority Q is O(N^2) because I first must sort the insert, which is worst case O(N) and then the actual insert is O(N).

## Insertions using iterators

We have discussed that insertions go in front of an iterator, however the stl for VS++ appears to insert BEHIND the iterator. As a result I have implemented my class to mirror this behavior.

template<class... \_Valty>

void \_Insert(\_Unchecked\_const\_iterator \_Where,

\_Valty&&... \_Val)

{ // insert element at \_Where

\_Nodeptr \_Pnode = \_Where.\_Mynode();

\_Nodeptr \_Newnode =

this->\_Buynode(\_Pnode, this->\_Prevnode(\_Pnode),

\_STD forward<\_Valty>(\_Val)...);

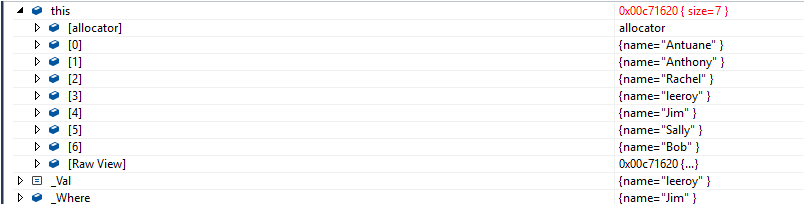
\_Incsize(1);

this->\_Prevnode(\_Pnode) = \_Newnode;

this->\_Nextnode(this->\_Prevnode(\_Newnode)) = \_Newnode;

}

Here is the result of the insert statement, it actually proves that the value is in fact inserted before the iterator. \_where describes the iterator value for the insert, and leeroy is inserted before jim, with the head pointing to Antuane.



This is supported by discussions in the stl insert here as well

<http://www.cplusplus.com/reference/list/list/insert/>

I also found a great way to crash VS debugger by declaring a temporary object and then trying to view the value of the list in the debugger. This causes the debugger to try and access the value that you placed into the list, that no longer exists and BOOM.

//insert into the middle of the list

std::list<people>::iterator stlListIterator = stlList->begin();

while (stlListIterator->name != "Jim")

{

++stlListIterator;

}

stlList->insert(stlListIterator, people("leeroy"));

(NOTE that we discussed this in class, I simply left it because I had typed it up)

## DRY Principle and Constructor Delegation

Don’t Repeat Yourself. I saw this primarily applied to constructor delegation WRT c++. Use as little code as possible, and chain them all together so that one calls the other. Then share that code with the copy constructor, and the assignment operator if possible.

The acronym I found here:

<http://www.thundernet.com/alanpartis/articles/constructor_chain.shtml>

But it appears to be common:

<https://en.wikipedia.org/wiki/Don%27t_repeat_yourself>

I used it with constructor delegation which your list code for this assignment introduced (sneakily). It is worth noting that this is new to c++11 so understanding what it is we are saving ourselves form typing is still important. Delegation allows you to not repeat constructor code to initialize arguments with default values by delegating “lesser” constructors to perform that work in the signature.