Project 1 – Templated Linked List

Overview

The purpose of this assignment is for you to write a data structure called a Linked List, which utilizes templates (similar to Java's generics), in order to store any type of data. In addition, the nature of a Linked List will give you some experience dealing with non-contiguous memory organization. This will also give you more experience using pointers and memory management. Pointers, memory allocation, and understand how data is stored in memory will serve you well in a variety of situations, not just for an assignment like this.

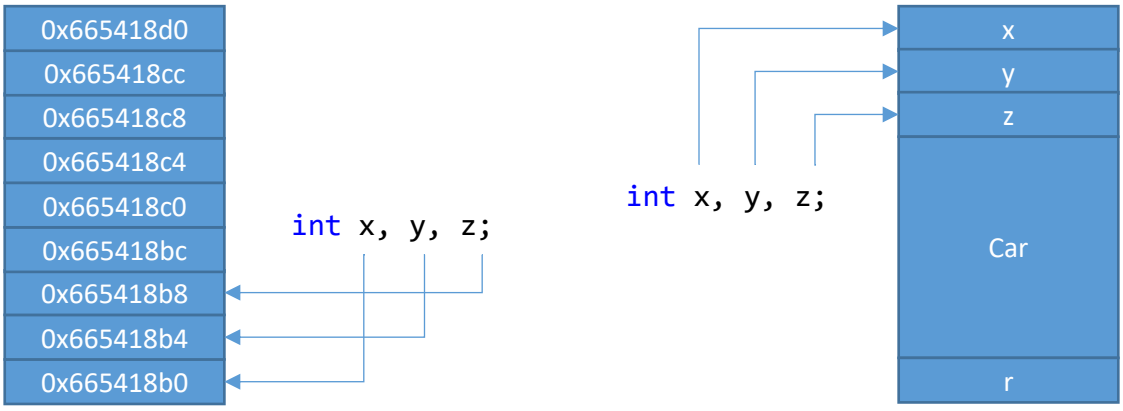
Background

Remember Memory?

Variables, functions, pointers—everything takes up SOME space in memory. Sometimes that memory is occupied for only a short duration (a temporary variable in a function), sometimes that memory is allocated at the start of a program and hangs around for the lifetime of that program. Visualizing memory can be difficult sometimes, but very helpful. You may see diagrams of memory like this:


```
int x, y, z;
Vehicle car;
char r, g, b, a;
Vehicle *ptr = &car;
```

Or, you may see diagrams like these:

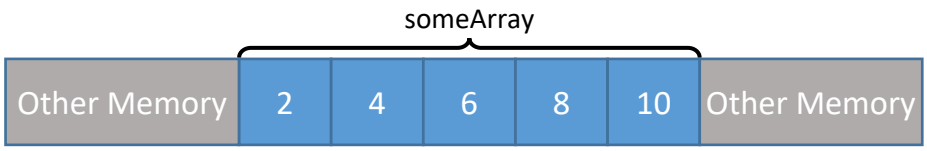


If you are trying to draw out some representation of memory to help you solve a problem, any of these (or some alternative that makes sense to you) will be fine.

Arrays

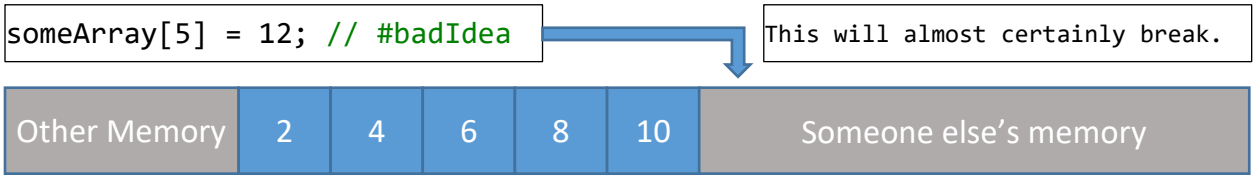
Arrays are stored in what is called **contiguous** memory. A contiguous memory block is one that is not interrupted by anything else (i.e. any other memory block). So if you created an array of 5 integers, each of those integers would be located one after the other in memory, with nothing else occupying memory between them. This is true for all arrays, of any data type.

```
int someArray[5];
someArray[0] = 2;
someArray[1] = 4;
someArray[2] = 6;
someArray[3] = 8;
someArray[4] = 10;
```



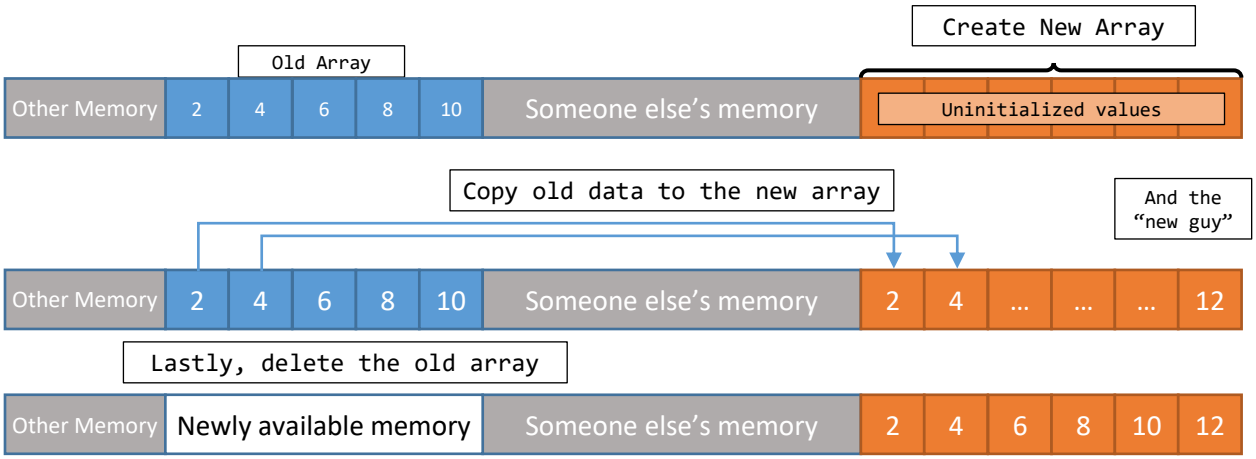
All of the data in an application is not guaranteed to be contiguous, nor does it need to be. Arrays are typically the simplest and fastest way to store data, but they have a grand total of zero features. You allocate one contiguous block, but you can't resize it, removing elements is a pain (and slow), etc.

Consider the previous array. What if you wanted to add another element to that block of memory? If the surrounding memory is occupied, you can't simply overwrite that with your new data element and expect good results.



In this scenario, in order to store one more element you would have to:

1. Create another array that was large enough to store all of the old elements plus the new one
2. Copy over all of the data elements one at a time (including the new element, at the end)
3. Free up the old array—no point in having two copies of the data



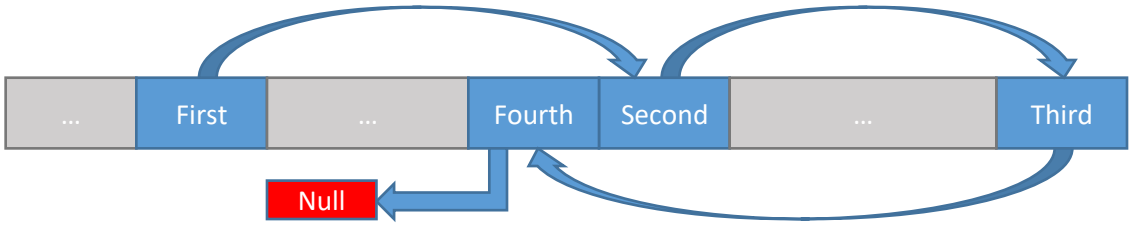
This process has to be repeated each time you want to add to the array (either at the end, or insert in the middle), or remove anything from the array. It can quite costly, in terms of performance, to delete/rebuild an entire array every time you want to make a single change. Cue the Linked List!

Linked List

The basic concept behind a Linked List is simple:

1. It's a container that stores its elements in a non-contiguous fashion
2. Each element knows about the location of the element which comes after it (and possibly before, more on that later)

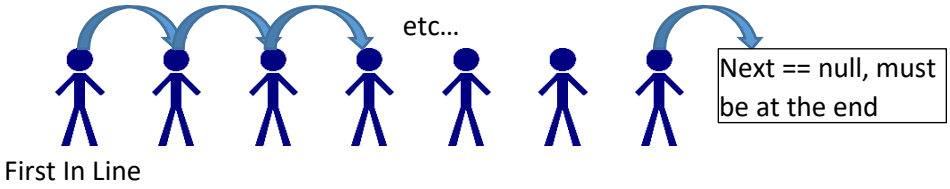
So instead of a contiguous array, where element 4 comes after element 3, which comes after element 2, etc… you might have something like this:



Each element in the Linked List (typically referred to as a "node") stores some data, plus some sort of reference (a pointer, in C++) to whatever node should come next. The First node knows only about itself, and the Second node. The Second node knows only about itself, and the Third, etc. In this example the Fourth node has a null pointer as its "next" node, indicating that we've reached the end of the data.

A real-world example can be helpful as well:

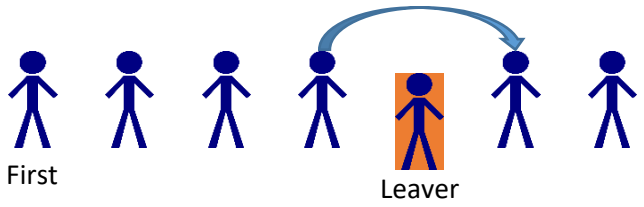
Think about a line of people, with one person at the front of the line. That person might know about the person who is next in line, but no further than that (beyond him or herself, the person at the front doesn't need to know or care). The second person in line might know about the third person in line, but no further. Continuing on this way, the last person in line knows that there is no one else that follows, so that must be the end. Since no one is behind that person, they must be at the end of the line.



So… What are the advantages of storing data like this? When inserting or removing elements into an array, the entire array has to be reallocated. With a Linked List, only a small number of elements are affected. Only elements surrounding the changed element need to be updated, and all other elements can remain unaffected. This makes the Linked List much more efficient when it comes to adding or removing elements.

Now, imagine one person wants to step out of line. If this were an array, all of the data would have to be reconstructed elsewhere. In a Linked List, only three nodes are affected: 1) The person leaving, 2) the person in front of that person, and 3) the person behind that person.

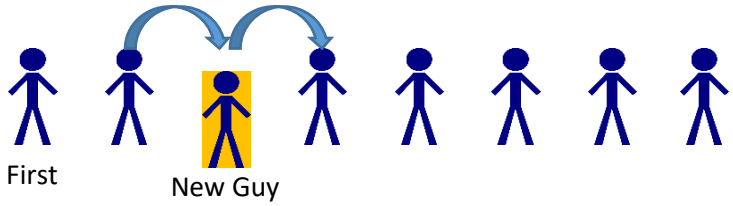
Imagine you are the person at the front of the line. You don't really need to know or care what happens 10 people behind you, as that has no impact on you whatsoever.



If the 5th person in line leaves, the only parts of the line that should be impacted are the 4th, 5th, and 6th spaces.

1. Person 4 has a new "next" Person: whomever was behind the person behind them (Person 6).
2. Person 5 has to be removed from the list.
3. Person 6… actually does nothing. In this example, a Person only cares about whomever comes after them. Since Person 5 was before Person 6, Person 6 is unaffected. (A Linked List could be implemented with two-way information between nodes—more on that later).

The same thought-process can be applied if someone stepped into line (maybe a friend was holding their place):



In this case, Person 2 would change their "next" person from Person 3, to the new Person being added. New Guy would have his "next" pointer set to whomever Person 2 was previously keeping track of, Person 3. Because of the ordering process, Person 3 would remain unchanged, as would anyone else in the list (aside from being a bit irritated at the New Guy for cutting in line).

So that's the concept behind a Linked List. A series of "nodes" which are connected via pointer to one another, and inserting/deleting nodes is a faster process than deleting and reconstructing the entire collection of data. Now, how to go about creating that?

Terminology

Node	The fundamental building block of a Linked List. A node contains the data actually being stored in the list, as well as 1 or more pointers to other nodes. This is typically implemented as a **nested class** (see below).
Singly-linked	A Linked List would be singly-linked if each node only has a single pointer to another node, typically a "next" pointer. This only allows for uni-directional traversal of the data—from beginning to end.
Doubly-linked	Each node contains 2 pointers: a "next" pointer and a "previous" pointer. This allows for bi-directional traversal of the data—either from front-to-back or back-to-front.
Head	A pointer to the first node in the list, akin to index 0 of an array.
Tail	A pointer to the last node in the list. May or may not be used, depending on the implementation of the list.

Nested Classes

The purpose of writing a class is to group data and functionality. The purpose of a **nested** class is the same—the only difference is where we declare a nested class. We declare a nested class like this:

```
class MyClass
{
public:
   // Nested class
   struct NestedClass
   {
      int x, y, z;
      int SomeFunction();
   };
private:
   // Data for "MyClass"
   NestedClass myData[5];
   NestedClass *somePtr;
   float values[10];
   // Etc…
};
```

```
// To create nested classes…
// Use the Scope Resolution Operator
MyClass::NestedClass someVariable;

// With a class template…
TemplateClass<float> foo;
TemplateClass<float>::Nested bar;

/* NOTE 1: You can make nested classes
private if you wish, to prevent access to
them outside of the encapsulating class. */

/* Why is NestedClass a struct in this
case? No reason in particular. Remember a
struct is just a class with public access
by default. */

/* NOTE 2: Nested classes and templates can
make for some ugly code. Be sure to read
the section at the end about typename */
```

Additional reading: http://en.cppreference.com/w/cpp/language/nested_types

The nature of the Linked List is that each piece of information knows about the information which follows (or precedes) it. It would make sense, then, to create some nested class to group all of that information together.

```
LinkedListClass
{
    NestedNodeClass
    {
        // The data you are storing
        // A pointer to the next node
        // A pointer to the previous node (if doubly-linked)
    };
    // A node pointer to the head
    // A node pointer to the tail
    // How many nodes are there?
};
```

Benefits and Drawbacks

All data structures in programming (C++ or otherwise) have advantages and disadvantages. There is no "one size fits all" data structure. Some are faster (in some cases), some have smaller memory footprints, and some are more flexible in their functionality, which can make life easier for the programmer.

Linked List versus Array – Who Wins?

Array	Linked List
Fast access of individual elements as well as iteration over the entire array	Changing the Linked List is fast – nodes can be inserted/removed very quickly
Random access – You can quickly "jump" to the appropriate memory location of an element	Less affected by memory fragmentation, nodes can fit anywhere in memory
Changing the array is slow – Have to rebuild the entire array when adding/removing elements	No random access, slow iteration and access to individual elements
Memory fragmentation can be an issue for arrays—need a single, contiguous block large enough for all of the data	Extra memory overhead for nodes/pointers

Code Structure

As shown above, the Linked List class itself stores very little data: Pointers to the first and last nodes, and a count. In some implementations, you might only have a pointer to the first node, and that's it. In addition to those data members, your Linked List class must conform to the following interface:

```
/*==== Behaviors ====*/
void PrintForward() const;
void PrintReverse() const;
void PrintForwardRecursive(const Node *node) const;
void PrintReverseRecursive(const Node *node) const;

/*==== Accessors ====*/
unsigned int NodeCount() const;
void FindAll(vector<Node *> &outData, const T&value) const;
const Node *Find(const T &data) const;
Node *Find(const T &data);
const Node * GetNode(unsigned int index) const;
Node * GetNode(unsigned int index);
Node *Head();
const Node *Head() const;
Node *Tail();
const Node *Tail() const;
```

```
/*==== Insertion ====*/
void AddHead(const T &data);
void AddTail(const T &data);
void AddNodesHead(const T *data, unsigned int count);
void AddNodesTail(const T *data, unsigned int count);
void InsertAfter(Node *node, const T &data);
void InsertBefore(Node *node, const T &data);
void InsertAt(const T &data, unsigned int index);

/*==== Removal ====*/
bool RemoveHead();
bool RemoveTail();
unsigned int Remove(const T &data);
bool RemoveAt(int index);
void Clear();

/*==== Operators ====*/
const T & operator[](unsigned int index) const;
T & operator[](unsigned int index);
bool operator==(const LinkedList<T> &rhs) const;
LinkedList<T> & operator=(const LinkedList<T> &rhs);

/*==== Construction / Destruction ====*/
LinkedList();
LinkedList(const LinkedList<T> &list);
~LinkedList();
```

Function Reference

Behaviors	
PrintForward	Iterator through all of the nodes and print out their values, one a time.
PrintReverse	Exactly the same as PrintForward, except completely the opposite.
PrintForwardRecursive	This function takes in a pointer to a Node—a starting node. From that node, recursively visit each node that follows, in forward order, and print their values. This function **MUST** be implemented using recursion, or tests using it will be worth no points. Check your textbook for a reference on recursion.
PrintReverseRecursive	Same deal as PrintForwardRecursive, but in reverse.
Accessors	
NodeCount	How many things are stored in this list?

FindAll	Find all nodes which match the passed in parameter value, and store a pointer to that node in the passed in vector. Use of a parameter like this (passing a something in by reference, and storing data for later use) is called an **output parameter**.
Find	Find the **first** node with a data value matching the passed in parameter, returning a pointer to that node. Returns null if no matching node found.
GetNode	Given an index, return a pointer to the node at that index. Throws an exception if the index is out of range. Const and non-const versions.
Head	Returns the head pointer. Const and non-const versions.
Tail	Returns the tail pointer. Const and non-const versions.
Insertion Operations	
AddHead	Create a new Node at the front of the list to store the passed in parameter.
AddTail	Create a new Node at the end of the list to store the passed in parameter.
AddNodesHead	Given an array of values, insert a node for each of those at the beginning of the list, maintaining the original order.
AddNodesTail	Ditto, except adding to the end of the list.
InsertAfter	Given a pointer to a node, create a new node to store the passed in value, after the indicated node.
InsertBefore	Ditto, except insert the new node before the indicated node.
InsertAt	Inserts a new Node to store the first parameter, at the index-th location. So if you specified 4 as the index, the new Node should have 3 Nodes before it. Throws an exception if given an invalid index.
Removal Operations	
RemoveHead	Deletes the first Node in the list. Returns whether or not the operation was successful.
RemoveTail	Deletes the last Node, returning whether or not the operation was successful.
Remove	Remove ALL Nodes containing values matching that of the passed-in parameter. Returns how many instances were removed.
RemoveAt	Deletes the index-th Node from the list, returning whether or not the operation was successful.
Clear	Deletes all Nodes. Don't forget the node count—how many nodes do you have after you deleted all of them?
Operators	
operator[]	Overloaded brackets operator. Takes an index, and returns the **index**-th node. Throws an exception if given an invalid index. Const and non-const versions.
operator=	Assignment operator. After listA = listB, listA == listB is true. Can you utilize any of your existing functions to make write this one? (Hint: Yes you can.)
operator==	Overloaded equality operator. Given listA and listB, is listA equal to listB? What would make one Linked List equal to another? If each of its nodes were equal to the corresponding node of the other. (Similar to comparing two arrays, just with non-contiguous data).
Construction / Destruction	
LinkedList()	Default constructor. How many nodes in an empty list? What is head pointing to? What is tail pointing to? Initialize your variables!
Copy Constructor	Sets "this" to a copy of the passed in LinkedList. The other list has 10 nodes, with values of 1-10? "this" should too.
~LinkedList()	The usual. Clean up your mess. (Delete all the nodes created by the list.)

Template types and the typename keyword

The compilation process for templates requires specialization—that is, your compiler essentially copies-and-pastes a version of your template, replacing all the instances of <T> with the type you specified when creating an instance of the class. When dealing with templates within templates, nested template classes, etc… the compiler sometimes needs a bit of assistance when figuring out a type.

In order to know how to specialize, it has to know everything about the template class. If that template has some other template, it needs to know everything… before it knows everything… and therein lies the problem.

The typename keyword is a way of telling your compiler "Hey, what immediately follows typename is a data type. Treat it as such when you compiler everything else." For example:

```
template <typename T>
class Foo
{
public:
    struct Nested
    {
        T someThing;
    };
    Nested SomeFunction();
};
```

When defining the function "SomeFunction" you might write this:

```
template <typename T>
Nested Foo<T>::SomeFunction() // Error, what is a "Nested"?
```

You could clean that up by specifying that a Nested object is part of the Foo class:

```
template <typename T>
Foo::Nested Foo<T>::SomeFunction() // Error, Foo is a template class, label it as such
```

Okay… how about this?

```
template <typename T>
Foo<T>::Nested Foo<T>::SomeFunction() // This SHOULD work, but… still doesn't. Why?
```

When the Foo class is being defined, it is referencing a type, Nested. This type is part of the Foo class, but the compiler doesn't know it's part of the class until it's done defining the class… But, since Foo is in the process of being defined… how can something simultaneously be defined not-yet-defined? Answer: It can't.

Solution: using the typename keyword, tell the compiler that Nested IS IN FACT A TYPE, and so the compiler doesn't have to wait to fully define Foo before finding out what it is.

```
// typename: Yes, compiler, Foo<T>::Nested is a type. Honest. You'll realize it later.
template <typename T>
typename Foo<T>::Nested Foo<T>::SomeFunction()
```

Ugly? You bet! Part of the joy of working with templates? It sure is! Every programming language has quirks like this that you just have to learn over time.

Tips

A few tips for this assignment:

- Start small! Work on one bit of functionality at a time. Work on things like Add() and PrintForward() first, as well as accessors (brackets operator, Head()/Tail(), etc). You can't really test anything else unless those are working.
- Your output is simple: print the data, print newline
- Remember the "Big Three" or the "Rule of Three"
    - If you define one of the three special functions (copy constructor, assignment operator, or destructor), you should define the other two
- Refer back to the recommended chapters in your textbook as well as lecture videos for an explanation of the details of dynamic memory allocation
    - There are a lot of things to remember when memory allocation
- Make charts, diagrams, sketches of the problem. Memory is inherently difficult to visualize, find a way that works for you.
- Don't forget your node count!

Sample Outputs

```
=====Testing AddHead() functionality====
Node count: 6
Print list forward:
10
8
6
4
2
0
Print list in reverse:
0
2
4
6
8
10
```

```
=====Testing AddTail() functionality====
Node count: 8
Print list forward:
0
3
6
9
12
15
18
21
Print list in reverse:
21
18
15
12
9
6
3
0
```

```
=====Testing operator[] to access value of nodes=====
Adding 10 to end of list
Adding 999 to end of list
Adding 22 to end of list
Value of node[0]: 10 Expected value: 10
Value of node[1]: 999 Expected value: 999
Value of node[2]: 22 Expected value: 22
Using brackets operator in a loop to change node values...
0
1
2
```

```
=====Testing RemoveAt() and clearing with RemoveHead()/RemoveTail()
functionality=====
Initial list:
Batman
RemoveMe
Superman
RemoveMe
Wonder Woman
RemoveMe
The Flash

Removing using RemoveAt()...
Batman
Superman
Wonder Woman
The Flash

Attempting to remove out of range using RemoveAt()...
Attempt to RemoveAt(100) failed.

Clearing list using RemoveHead()...
List is empty!
Adding additional nodes...
Robin
Batgirl
Nightwing
Red Hood
Bluebird
Clearing list using RemoveTail()...
List is empty!
```