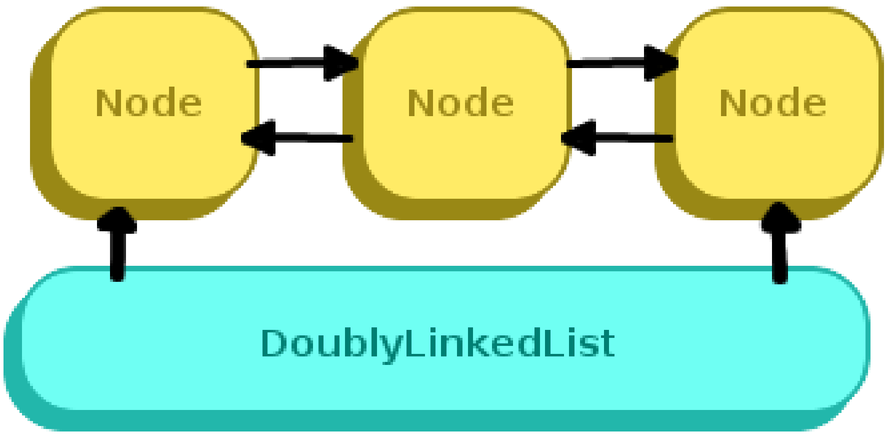
# Project 1: Linked List



## 1.1 Information

**Topics:** Linked lists, pointers, basic program implementation

**Turn in:** All source files (.cpp and .hpp).

**Starter files:** Download on GitHub or D2L.

### 1.1.1 About

In this project, you will be implementing a Doubly Linked List.

There are two parts of the project: The tests, and an actual program. When you start the program, it will ask you which to run.

```
-----------------------
1. Run tests
2. Run program
3. Exit
>>
```

For the first part of the assignment, you will work on implementing the DoublyLinkedList, using the tests to verify your work. For the second part, you will go into the `project1_program` and replace usage of the STL `list` with your DoublyLinkedList instead.

```
Project 1 - Linked List/
 ├─project1_main.cpp ...  Contains main()
 ├─project1_LinkedList.hpp ...  Class declaration
 ├─project1_LinkedList.cpp ...  Class function definitions
 ├─project1_Tester.hpp ...  Unit test functions
 ├─project1_CustomerData.hpp ...  CustomerData, the data
 │                                stored in the LinkedList
 ├─project1_CustomerData.cpp
 ├─project1_program.hpp ...  The program functionality
 ├─project1_program.cpp
 ├─cuTEST/ ...  Unit test framework
 │  ├─Menu.hpp
 │  ├─StringUtil.hpp
 │  ├─TesterBase.hpp
 │  └─TesterBase.cpp
 └─CodeBlocks Project/ ...  Code::Blocks project file
    └─VS2015 Project 1 ...  Visual Studio project files
```

Your project needs to be compiling as a C++11 project. To set this up in Code::Blocks, go to Settings, Compiler, then in the Compiler Flags pane, check the "Have g++ follow the C++11 ISO C++ language standard".

## 1.2 The tests

This project contains unit tests. To begin with, all the functions that you need to implement have exceptions...

```
throw runtime_error( "PopFront() not yet implemented" );
```

As you're implementing each function, you will remove these `throw` statements so that the functions will run during the testing. You can still leave some of the throw statements in the functions you're not currently working with; even if Visual Studio stops at a given function, you can stop the program and refresh the test results page.

# Contents

1.1 Information . . . . . . 1
1.1.1 About . . . . . . 1
1.2 The tests . . . . . . 2

**2 Doubly Linked List basics 4**
2.1 Node . . . . . . 4
2.1.1 Constructor . . . . . . 4
2.2 LinkedList . . . . . . 5
2.2.1 Constructor . . . . . . 5
2.2.2 Destructor . . . . . . 5
2.2.3 IsEmpty . . . . . . 5
2.2.4 Size . . . . . . 5
2.2.5 Clear . . . . . . 5
2.2.6 Adding new items . . . . . . 6
2.2.7 Removing items . . . . . . 8
2.2.8 Accessing items . . . . . . 10

**3 The rest of the program 13**
3.1 The CustomerData class . . . . . . 13
3.2 Updating the Program . . . . . . 13
3.2.1 Class declaration . . . . . . 13
3.2.2 Program::LoadCustomers() . . . . . . 14
3.2.3 Program::SaveCustomers() . . . . . . 14

**4 Grading Breakdown 15**

**5 Extra credit: Additional features 16**
5.1 Insert . . . . . . 16
5.2 Remove . . . . . . 16
5.3 operator== and operator!= . . . . . . 16

# Chapter 2

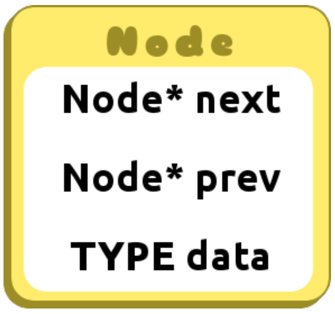
# Doubly Linked List basics

A linked list is a type of structure that only allocates memory for new objects as-needed. Unlike the Dynamic Array, it doesn't pre-allocate large chunks of memory and resize as needed. Instead, each time a new item is pushed into the list, memory for a new "Node" is allocated.

A linked list requires two different classes: A Node and a LinkedList.

## 2.1 Node

A Node contains the data itself, as well as pointers. In a doubly-linked-list, the Node contains pointers to the next node and the previous node. In a singly-linked-list, the node only contains a pointer to the next node.



Using these previous and next pointers to nodes, we can traverse the list by updating a "traversal" pointer... starting at the first node, and step-by-step going to each node's next pointer.

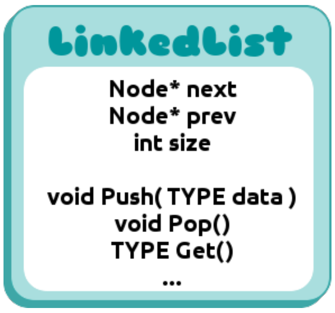
When a Node is first created, its next and previous pointers should be pointing to `nullptr`.

### 2.1.1 Constructor

Make sure to initialize the Node's previous and next pointers to `nullptr` in its constructor.

## 2.2 LinkedList

The LinkedList class contains the main functionality of the list, such as Push to add new items, Pop to remove items, and Get to access items. The LinkedList also contains pointers to the first item in the list and, often, the last item in the list (though not required). The LinkedList is also responsible for keeping track of how many items have been added.



When a LinkedList is empty, whether when it is first created or if everything has been removed, the first and last pointers should be ponting to `nullptr`.

When the LinkedList is destroyed, its destructor should be responsible for freeing up all the memory allocated for the Node items.

### 2.2.1 Constructor

Initialize the first and last pointers to `nullptr`, and set the item count to 0.

### 2.2.2 Destructor

Call the `Clear()` function. We will have `Clear()` handle freeing all the memory.

### 2.2.3 IsEmpty

If the item count is 0 return true, otherwise return false.

### 2.2.4 Size

Return the item count.

### 2.2.5 Clear

Create a while loop - it will continue looping *while the list is not empty.* Within this while loop, simply call `PopFront()` or `PopBack()`.

### 2.2.6 Adding new items

When a Push or Insert function is called, the LinkedList needs to...

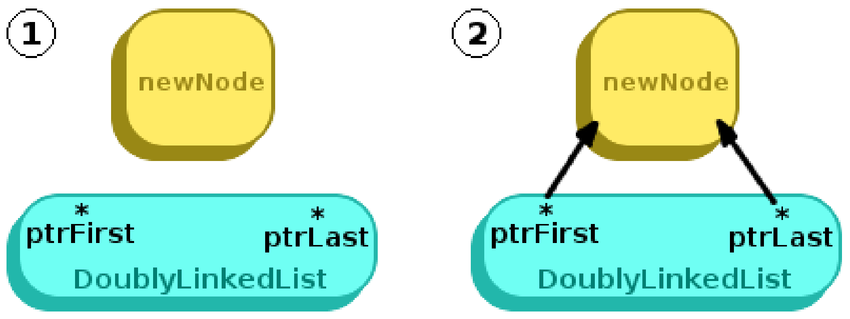
- **allocate memory for the new Node**
- **update the pointers of any existing Nodes that will be the new Node's neighbor**
- **update the pointer for the first or last item in the LinkedList.**

As we add new items to the linked list, each new element needs memory allocated. To do this, create a local pointer variable within the function, then allocate the memory for the Node, and set up the Node's data.

```
void Push( T data )
{
    Node* newNode = new Node;
    newNode->data = data;
    m_itemCount++;
    // ...
}
```

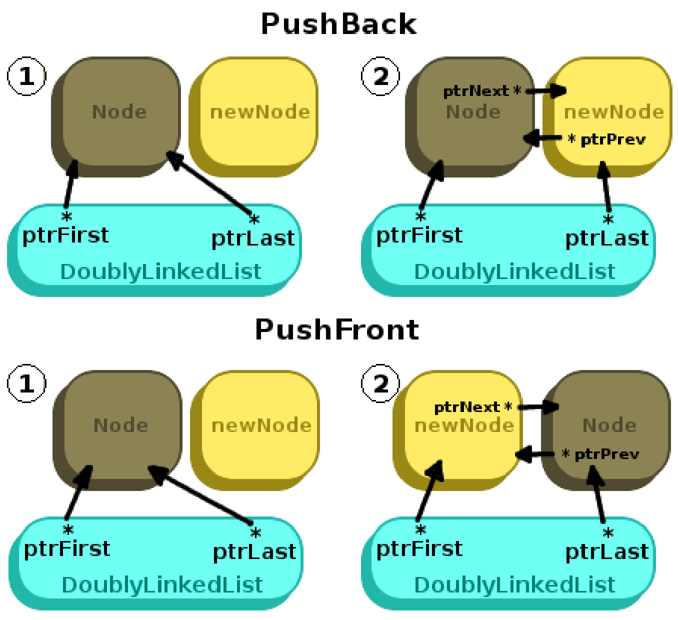
What we do with this newNode will then depend based on whether or not the LinkedList is empty, or if it contains at least one item.

**Empty LinkedList:**



Once a new Node is created, if our LinkedList is empty, the only step we have to do is update its *first* and *last* Node* pointers to the newNode that we've created.

**Non-empty LinkedList:**



If the LinkedList is **not** empty, there are more steps you will have to take care of. These steps also depend on whether you're writing a **Push Back** (insert as new end) or **Push Front** (insert as new beginning) function.

- PushBack:
    1. Set the current *last Node*'s next pointer to the new node.
    2. Set the new node's previous pointer to the current *last Node*.
    3. Update the *last Node* pointer of the LinkedList to point at the new Node.
- PushFront:
    1. Set the current *first Node*'s previous pointer to the new node.
    2. Set the new node's next pointer to the current *first Node*.
    3. Update the *first Node* pointer of the LinkedList to point at the new Node.

**Don't delete!**
You won't need to call delete in the same function; the Pop function will be responsible for freeing any allocated memory.

### 2.2.7 Removing items

When you remove an item from the LinkedList, you will have to update the pointers of the LinkedList, as well as any Nodes that were neighbors to the Node that was removed. Additionally, the two scenarios that have different behaviors here are if the LinkedList contains *only one item* (so you're removing the last item), or if it contains more than one item.

**Pop when list is empty?** If Pop is called and the list is already empty, the usual behavior is to ignore it.

For example, if we were using the STL `list`, we can see on its documentation page (http://www.cplusplus.com/reference/list/list/pop_back/) its exception safety level:

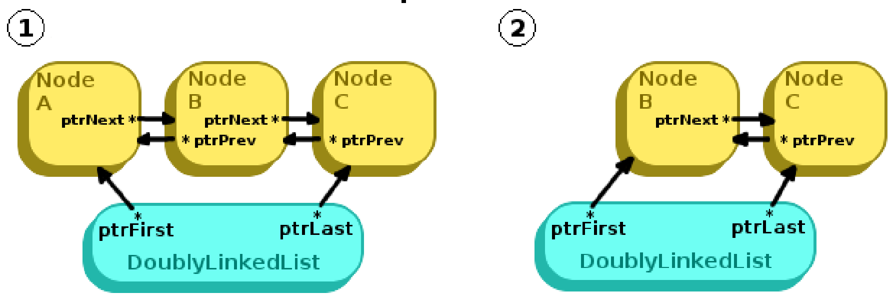
> **Exception safety** If the container is not empty, the function never throws exceptions (no-throw guarantee). Otherwise, it causes *undefined behavior*.

**Removing last item** If you're removing the last item from the LinkedList, you will have to make sure to free the memory for the Node, but also update the LinkedList's Node pointers to the first and last item; these should both point at `nullptr`.

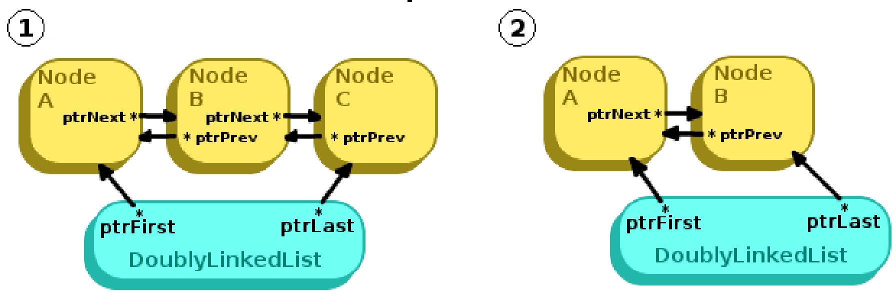
```
void Pop( T data )
{
    // It is the last item
    if ( m_ptrFirst == m_ptrLast )
    {
        delete m_ptrFirst;
        m_ptrFirst = nullptr;
        m_ptrLast = nullptr;
        m_itemCount--;
    }
    // ...
}
```

**Pop when we have more than one item in the list** In our DoublyLinkedList we can Pop items off from the front of the list or the back of the list, so the way to implement this changes a bit for each.

**PopFront**



**PopBack**



- PopBack:
  1. Locate the second-to-last item in the LinkedList (With a DoublyLinkedList, you can simply get the last Node's *previous* item.)
  2. Update the second-to-last Node's next pointer to be `nullptr`.
  3. Free the memory pointed to by the last Node pointer
  4. Update the last Node pointer to point to the (formerly) second-to-last Node
  5. Decrease the item count
- PopFront:
  1. Locate the second item in the LinkedList (You can get the first Node's *next* item.)
  2. Update the second Node's previous pointer to be `nullptr`.
  3. Free the memory pointed to by the first Node pointer
  4. Update the first Node pointer to point to the (formerly) second Node
  5. Decrease the item count

**Keeping track of the second Node**
Your code might look ugly if you're trying to do these steps like

```
m_ptrLast->ptrPrev->ptrNext = nullptr;
```

Instead of working like this, simply make a local Pointer to keep track of your second-to-last or second item...

```
// PopBack
Node* ptrSecondToLast = m_ptrLast->m_ptrPrev;
ptrSecondToLast->ptrNext = nullptr;
// ...

// PopFront
Node* ptrSecond = m_ptrFirst->m_ptrNext;
ptrSecond->ptrPrev = nullptr;
// ...
```

### 2.2.8 Accessing items

For our DoublyLinkedList, we will be writing a `GetFront`, `GetBack`, and `operator[]` functions. For each of these, the functions will return the data-type related to the data *within* the Node class - the user doesn't care that there are Node objects or not, they only care about the data.

**Accessing the first and last item** To access the data for the first Node, you simply return the data item.

```
// Get data at front
return m_ptrFirst->m_data;

// Get data at back
return m_ptrLast->m_data;
```

The main thing you have to watch out for here is if the list is empty (or otherwise, if the first pointer or last pointer is set to `nullptr`).

**Getting an item at some index** The tricky part is when the user wants to access an item at some position... Because the LinkedList's elements are not contiguous in memory, we cannot access item $n$ as easily as we would with an array. Instead, we have to begin at the first Node and work our way forward (via the next pointers), counting what place we're at. This is known as *traversing the list*.

When traversing the list, we need to make a traversal pointer that begins at the first item, as well as a counter to keep track of our current position in the list.

```
int counter = 0;
Node* ptrCurrent = m_ptrFirst;
```

**Common error: Allocating memory**
Note here that the `ptrCurrent` pointer is simply a pointer to some existing memory; we are **NOT** allocating new memory here!

Then, we simply need to "walk forward" the amount of times specified in the parameters. The header for the `operator[]` function in our program look like this:

`CustomerData& LinkedList::operator[]( const int index )`

so in this case, we want to go to position *index*.

which begins at the first item, and steps through each node via each node's ptrNext pointer.

We cannot randomly access data in a linked list because it is not implemented with an array. Because we only keep track of the first and last item with pointers, we have to traverse the list to find some item at position n. When can we stop "walking" the list? Once our `counter` is equal to the `index`!

```
while ( counter != index )
{
    // ...
}
```

And within the loop, all we have to do is keep moving our traversal pointer forward by one, and incrementing our counter.

```
while ( counter != index )
{
    // Go to the next item
    ptrCurrent = ptrCurrent->ptrNext;
    counter++;
}
```

However, this code on its own could cause a crash - if the size of the list is smaller than the index parameter, then the ptrCurrent will walk off the list and be set to `nullptr`. Then, if it tries to access `ptrCurrent->ptrNext`, the program will crash!

So, before you implement any of the "walking", you should do an error check to make sure the index is valid - and throw an exception if it is invalid! (less than 0, or greater-than-or-equal-to the LinkedList size!)

# Chapter 3

# The rest of the program

## 3.1 The CustomerData class

The data that our Node classes wrap is a CustomerData type. You don't need to necessarily know the specifics about this type to work with the LinkedList, but it is pre-declared in the `project1_CustomerData` hpp and cpp files.

```
struct CustomerData
{
    int id;
    string username;
    string ssn;
    string cardNumber;
    string cardType;

    // ...
};
```

## 3.2 Updating the Program

Once you're finished writing the LinkedList, you will go into the Program class and update it so that it is no longer using the STL `list` anymore. This requries updating a few areas.

### 3.2.1 Class declaration

In `project1_program.hpp`, you will first update the member variables so that the `m_customers` variable is a `LinkedList` instead of a `list<CustomerData>`.

### 3.2.2 Program::LoadCustomers()

In this function, instead of using `m_customers.push_back( data )`, you will update it to use your LinkedList's `PushBack` function.

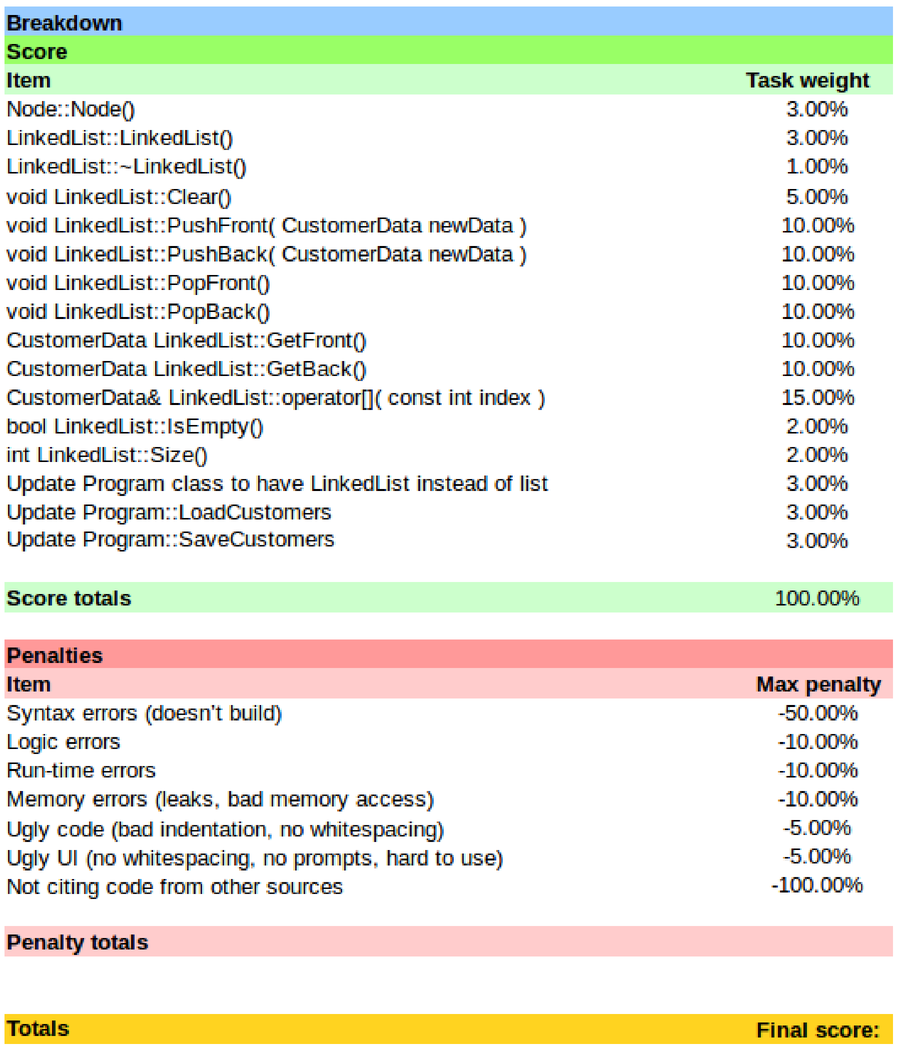
### 3.2.3 Program::SaveCustomers()

For this function, all you have to do is erase the original for loop that iterates through the `list<Customerdata>` item, and un-comment-out the other for-loop that works with the LinkedList...

```
for ( int i = 0; i < m_customers.Size(); i++ )
{
    CustomerData cd = m_customers[i];
    output
        << cd.ssn << "\t"
        << cd.cardNumber << "\t"
        << cd.cardType << "\t"
        << cd.username << "\t"
        << endl;
}
```

# Chapter 4

# Grading Breakdown



| | |
|---|---|
| **Breakdown** | |
| **Score** | |
| **Item** | **Task weight** |
| Node::Node() | 3.00% |
| LinkedList::LinkedList() | 3.00% |
| LinkedList::~LinkedList() | 1.00% |
| void LinkedList::Clear() | 5.00% |
| void LinkedList::PushFront( CustomerData newData ) | 10.00% |
| void LinkedList::PushBack( CustomerData newData ) | 10.00% |
| void LinkedList::PopFront() | 10.00% |
| void LinkedList::PopBack() | 10.00% |
| CustomerData LinkedList::GetFront() | 10.00% |
| CustomerData LinkedList::GetBack() | 10.00% |
| CustomerData& LinkedList::operator[]( const int index ) | 15.00% |
| bool LinkedList::IsEmpty() | 2.00% |
| int LinkedList::Size() | 2.00% |
| Update Program class to have LinkedList instead of list | 3.00% |
| Update Program::LoadCustomers | 3.00% |
| Update Program::SaveCustomers | 3.00% |
| | |
| **Score totals** | 100.00% |
| | |
| **Penalties** | |
| **Item** | **Max penalty** |
| Syntax errors (doesn't build) | -50.00% |
| Logic errors | -10.00% |
| Run-time errors | -10.00% |
| Memory errors (leaks, bad memory access) | -10.00% |
| Ugly code (bad indentation, no whitespacing) | -5.00% |
| Ugly UI (no whitespacing, no prompts, hard to use) | -5.00% |
| Not citing code from other sources | -100.00% |
| | |
| **Penalty totals** | |
| | |
| **Totals** | **Final score:** |

# Chapter 5

# Extra credit: Additional features

If you want to implement some additional features for your LinkedList, you can gain some extra credit on this assignment.

## 5.1 Insert

Implement an Insert function that will allow some new data to be added at some *index* (not just the front or back). This needs to also update both its previous and next neighbor Nodes to point to the new item.

## 5.2 Remove

Implement a Remove function that will remove some data at some *index* (not just the front or back). This also needs to update both its previous and next neighbor Nodes to point to each other instead of the Node to be removed.

## 5.3 operator== and operator!=

Iterate through two DoublyLinkedList (you'll need two "walker" Node pointers) and compare the data in each Node at each position. If all the data matches up (and the sizes of both lists are the same), then the lists can be considered equal. Operator== would return `true` and operator!= would return `false`.