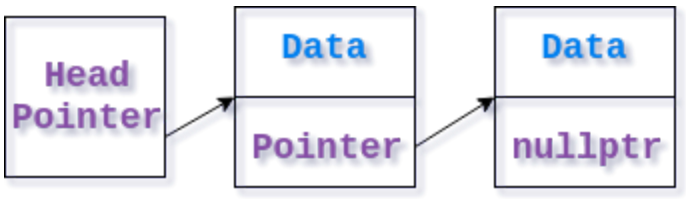
Linked List

A linked list is a data structure in which data is stored in a **node** and each **node** is linked together via a pointer. A **node** is generally a **struct** (in terms of CSII) with variable(s) for data and a pointer to another instance of the **struct**. An example of a **node**:




```
struct Node
{
    int data; //data can be any type
    Node* link; //link to another Node in the form of a pointer
};
```

In this example, there is an **int** variable for data and a pointer of type **Node**. You are able to put as many variables for data as you would like. This gives more leniency than an array or vector in which you are only able to have one base type and have only one piece of data per element.

Since pointers are heavily used in a linked list, link lists are dynamically allocated. This means that they are not stored in memory sequentially and are linked solely by the pointers. This means that if a node is deleted and the two portions of the linked list are not reconnected properly, data will be lost.

Insertion

At the beginning of a linked list is what is known as the **head pointer** (or **head**). This pointer points to the first node of the linked list. Usually **head** is initialized to **nullptr**, such as **Node* head = nullptr;** before pointing to the first node of a linked list. This brings us to an example function for the creation of a node for a linked list such that new nodes are inserted in the front of the linked list:

```
void nodeCreation(Node* &head) //Pass by reference
{
    Node* newNode = new Node; //Dynamically allocate a Node
    newNode->data = 100; //Set data to int of your choice (ex 100)
    newNode->link = head; //Link up to rest of linked list
    head = newNode; //Make head point to newly created node
}
```

The head pointer is passed by reference as it will be altered in this function. A new dynamically allocated node is created.

Remember when dealing with pointers, they must be dereferenced to access the data it points to. Also, since you are dealing with a **struct**, you utilize the dot operator. These two things combined form

the arrow operator. See **Pointers** or **Struct** for more information on the arrow operator.

(May help to draw out this paragraph) The link of the new node will point to wherever the head pointer is currently pointing. For the first created node of the linked list, the link will point to **nullptr** as head was initialized to **nullptr**. The head pointer is then reinitialized to point to the newly created node. The second node created will need to be linked to the rest of the linked list. Therefore, its link will point to wherever head is currently pointing. This time head is pointing to the node that was created the previous time. Head is then reinitialized to point to the newly created node. This pattern continues until you are finished inserting new nodes.

What was created was a verbose depiction of the picture on the previous page. The head pointer points to the newest created node. That node points to the node created during the last function invocation which points to **nullptr** to signify the end of the linked list. Again, each new node will be added to the front making the first created node the last node in the linked list and the newest node the first node of the linked list. It is possible to insert elsewhere in a linked list. Here are two functions to insert either in the front or the back of your linked list:

```
void insertFront(Node* &head, Node* &back, int data)
{
    Node* newNode = new Node;
    newNode->data = data;
    newNode->link = head;
    head = newNode;

    if(back == nullptr) //Only one node present, so it is both the
        back = head;    //front node and the back node
}

void insertBack(Node* &head, Node* &back, int data)
{
    Node* newNode = new Node;
    newNode->data = data;
    newNode->link = nullptr; //Last node always points to nullptr
    if(back) //Check to see if at least one node exists
        back->link = newNode; //Link up node to rest of linked list
    back = newNode; //Make back point to newly created last node

    if(head == nullptr) //Only one node present, so it is both the
        head = back;    //front node and the back node
}
```

It is important to remember that you cannot dereference a **nullptr**. This is why there is a check to see if at least one node exists in the **insertBack** function. If there is no existing node, then there is no linked list to connect. Therefore, this newly created node would be the first node of the linked list.

Traversal

Traversing a linked list requires a pointer akin to head or back. If you do not wish to create a new pointer for traversal in a function, pass the head pointer by **const** value so whatever you do inside the function cannot alter the location of head/the contents. If you want to display your linked list, you may do so with a function as such:

```
void display(const Node* head)
{
    while(head) { //Check to see if head points to a valid node
        cout << head->data << endl;
        head = head->link; //Move head to next node in sequence
    }
} //However creating a pointer for traversal makes easier-to-read code
```

The **while** loop will terminate once **head** is equal to **nullptr**. This will display the data from the front of the list to the back of the list. To display the linked list backward would best require a doubly linked list, which is in the Standard Template Library.

If you would like to search for a particular piece of data, you may traverse through your linked list similarly to the above example. Here is an example of finding all instances of a particular **int** and displaying which nodes contain that value. If it is not found, a message saying as such is displayed.

```
void search(Node* head, int target)
{
    int count = 1; //Count for proper node placement
    bool found = false;
    while(head) {
        if(head->data == target) {
            cout << target << " found in node: " << count << endl;
            found = true;
        }
        head = head->link; //Move to next node in sequence
        ++count; //Increase count with each head pointer movement
    }
    if(found == false)
        cout << target << " not found in linked list\n";
}
```

Deletion

As briefly noted on the first page, deletion needs to be carried through correctly or you will lose some or all of your data as the data is not sequential in memory. For simplicity, the following example will be of a linked list with only a head pointer.

```
void deleteNode(Node* &head, int target)
{
    if(head == nullptr) //Check for empty linked list
        return;

    Node* connectPtr = head; //Pointer for connection
    Node* delPtr = head->link; //Pointer for deletion
    if(head->data == target) { //Deletion of first node
        head = head->link; //Move head to next node in list or nullptr
        delete connectPtr; //Delete first node
        return;
    }
    while(delPtr) { //Cycle through rest of linked list
        if(delPtr->data == target) {
            connectPtr->link = delPtr->link; //Relink linked list
            delete delPtr; //Delete node
            return;
        } else { //Move pointers to subsequent nodes if no match found
            connectPtr = delPtr;
            delPtr = delPtr->link;
        }
    }
}
```

This function first checks to see if there is in fact nodes in the linked list. If the linked list is empty, there is nothing to be checked nor deleted and the function terminates.

The next portion is declaring two **Node***, one for deletion and one for reconnecting the link list. They are initialized one node apart with **connectPtr** pointing to the node before **delPtr**.

If the first node of the linked list needs to be deleted, simply move **head** to **head->link** and delete **connectPtr**, which points to the first node.

If the target is elsewhere in the linked list, traverse the linked list by moving both pointers to the next node in line until **delPtr** finds the target. When it finds the first instance the the target, you will need to reconnect the linked list before **delPtr** with the rest of the linked list after **delPtr**. This is why **connectPtr** points to the node before **delPtr**. To reconnect, change **connectPtr->link** to point to

the node after **delPtr**, which would be **delPtr->link**. Once the linked list is reconnected, delete **delPtr**. If **delPtr** reaches **nullptr**, target was not found and the function terminates without deleting anything.

STL List

In the Standard Template Library, there exists two lists: **forward_list** and **list**. The **forward_list** is implemented as a singly linked list and can only be iterated forward. This is similar to the previous examples of this section as there is only one link per node pointing to the subsequent node. The preprocessor directive **#include <forward_list>** is required. The **list** is implemented as a doubly linked list and has bidirectional iteration. Each node points to the node before it and after it. This is the one you will use. The preprocessor directive **#include <list>** is required. These are sequential containers as each node can be accessed one after another. Remember, the nodes are not sequential in memory. Additionally, if you are not **using namespace std;** you need to **using std::list;** at the top of your program or prepend **std::** when declaring a list or iterator. Here are some basic member functions for list:

To declare a list you use the template **list<TYPE> myList** where **TYPE** is your base type of the list, such as **char**, **int**, **class**, etc. You are able to initialize a list on the same line as such:

```
std::list<int> myList; //Empty list
std::list<char> myList(3, 'a'); //Three nodes with the value of a
std::list<char> anotherList(myList); //Copies myList to anotherList
```

Notice, if you do not create your own linked list, then each node in the Standard Template Library will contain one piece of data per node.

Since list has bidirectional iterators, you are able to traverse your list forward, backward (reverse), and with constant iterators (can only access data but not change it). As with a vector, the **end** iterator points to **one past the last node**. The declaration and initialization is the same as vectors in terms of iterators:

```
std::list<int>::iterator it = myList.begin();
std::list<char>::const_iterator cit = myList.cbegin();
std::list<double>::reverse_iterator rit = myList.rbegin();
```

Remember iterators are like pointers such that they need dereferenced to access the data they point to.

Using the iterators, you are able to utilize a **for** loop to display the linked list. Here is an example using the iterator **it** from above:

```
for( ; it != myList.end(); it++) //Iterator already initialized above
    cout << *it << endl;         //No need for first for loop argument
```

You are able to display your linked list in order without an iterator using a **for** loop as such:

```
for(auto e : myList)
    cout << e << endl;
```

You are able to see if a list is empty by the member function **empty()**. It will return a **bool** value true if empty or false if there are nodes in the linked list. For example: **if(myList.empty())**

To obtain the size of your linked list, use the member function **size()**. This will display how many nodes are in the linked list.

To display the first node's data, use **front()**. To display the last node's data, use **back()**. For example: **cout << myList.back();**.

To insert a node in the front of a linked list, use **push_front(DATA)**. To insert a node in the back of a linked list, use **push_back(DATA)**. **DATA** is what you want to insert into the node. For example in a **char** list **myList.push_back('b');** will create a node at the end of the linked list that contains the data **b**.

To delete the first node of a linked list, use **pop_front()**. To delete the last node of a linked list, use **pop_back()**.

To clear a linked list, use **clear()**. This will make the linked list empty.

To remove all duplicates in a linked list, use **unique()**.

To sort all nodes, use **sort()**. For example sorting a list of integers would place the lowest number first and the highest number last.

To append one list to another, use **merge()**. For example, to append list2 to list1, you would **list1.merge(list2);**. Remember when you do this, the appended list (list2 in this case) will be empty as all its contents transferred to list1.

To reverse a linked list, use **reverse()**.

For a complete list of member functions, including inserting and erasing nodes other than the front or the back of the linked list, visit **http://www.cplusplus.com/reference/list/list/**