Doubly Linked List in C++

November 2024

Contents

1 What is a doubly linked list?

We have looked at a linked list earlier in this course. That was what is known as a singly linked list. Elements are stored in nodes, each node has a pointer to th next node, and we maintain a head pointer. From the head pointer, we can get to each node and access the element stored there.

A doubly linked list is a data structure in which each node not only stores a pointer to the next node, but also to the previous node. We also maintain a pointer to the tail element, in addition to a head pointer. Figure 1 contrasts the two data structures.

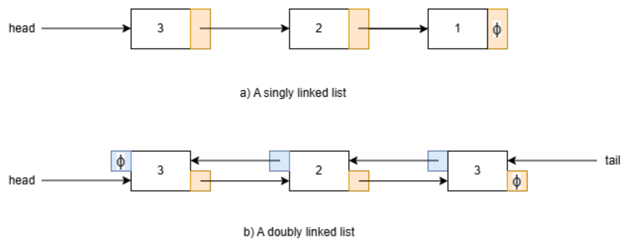

Figure 1: A singly linked list of integers vs a doubly linked list of integers

In a doubly linked list, while iterating, we can move forward as well as backward in the list. We can also start traversing it from the beginning as well as the end. This is similar to the `std::list` data structure that is part of the STL.

2 Introduction

In this handout, we learn how to create a doubly linked list from scratch in C++. Our linked list will be a template class so that we can use it to store any kind of data.

We'll create a linked list with a bidirectional iterator, much like the STL. This will enable us to use our `List` class using the same code that one would use with the `std::list` class.

One design approach that greatly simplifies the code for a linked list is to use *sentinel* nodes for the head and the tail. These are nodes that don't store any data. Even when the linked list is empty, these nodes still exist. It might not be immediately obvious what profound effect this simple choice has on the code complexity. If interested, once you're done implementing the approach given in this handout, repeat it without the sentinel nodes and compare the two implementations.

Let's begin!

3 Implementation

This section describes the implementation of the doubly linked list, step by step.

3.1 Template class

To create a template class, we start with the following:

```
template<typename T>
class List {

};
```

We declare that our class named `List` is a template class, with a type parameter named `T`.

3.2 Node structure

We should declare a node structure (or class). We have two options. We can either declare it outside the `List` class, or inside it. We'll implement it inside the class since outside the class, no one should have to worry about the node structure. The user of the class should just be able to store or retrieve whatever type of data they want. They need not think about the `next` and `prev` pointers. We'll take care of those within the class itself.

Here's the code with the `Node` structure:

```
template<typename T>
class List {
    protected:
        struct Node {
            T value;
            Node* next;
            Node* prev;
        };
};
```

We declare the `Node` structure in the `protected` portion of the class. This way, if you create another class derived from `List`, that class can conveniently access the `Node` structure.

The structure has a member variable named **value** that stores the data item. It has a type that matches the template parameter. There are two pointers named **next**, and **prev** that are meant to point to the next and previous nodes, respectively.

3.3 List class member variables

The **List** class needs to store at least two member variables: **head** and **tail**, so let's add those:

```
template<typename T>
class List {
    protected:
        struct Node {
            T value;
            Node* next;
            Node* prev;
        };
        Node* head;
        Node* tail;
};
```

3.4 Constructor and destructor

The constructor needs to create the sentinel nodes for head and tail. The destructor needs to deallocate all the nodes in the linked list. Let's start with the constructor.

An empty linked list should have just the sentinel nodes, pointing to each other like Figure 2. Let's write code for creating this setup in the constructor as shown below:

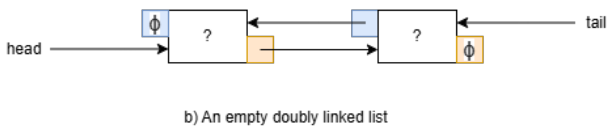

Figure 2: An empty doubly linked list

```
template<typename T>
class List {
    protected:
        struct Node {
            T value;
            Node* next;
            Node* prev;
        };
        Node* head;
        Node* tail;
    public:
        List() {
            head = new Node;
            tail = new Node;
            head -> prev = nullptr;
            head -> next = tail;
            tail -> prev = head;
            tail -> next = nullptr;
        }
};
```

This should be fairly easy to understand. Now, on to the destructor.

In the destructor, we need to deallocate all the nodes one by one. We'll start at the **head** node, and keep deleting the nodes one by one. While we're at it, let's create a helper function named `clear()` which deletes all the data nodes, i.e., all nodes excluding the head and tail sentinel nodes. The STL class has a function with the same name that does exactly this, too.

```
template<typename T>
class List {
    protected:
        struct Node {
            T value;
            Node* next;
            Node* prev;
        };
        Node* head;
        Node* tail;
    public:
        List() {
            head = new Node;
            tail = new Node;
            head -> prev = nullptr;
            head -> next = tail;
            tail -> prev = head;
            tail -> next = nullptr;
        }
        void clear() {
            Node* temp1 = head -> next;
            while (temp1 != tail) {
                Node* temp2 = temp1 -> next;
                delete temp1;
                head -> next = temp2;
                temp2 -> prev = head;
                temp1 = temp2;
            }
        }
};
```

We set a `temp1` pointer to the first data node. The first data node is the node next to the head. Wait a minute! What if the list is empty? In that case, this pointer wouldn't point to the first data node, right? Sure. But that wouldn't create a problem, because the next thing we write is a `while` loop that runs only as long as `temp1` isn't the same as `tail`. So, the code inside this loop will only run for data nodes. If there are no data nodes, the loop wouldn't run a single iteration, we don't have anything to do in `clear()` and we simply return.

On the other hand, if `temp1` does point to a data node, then we first save the pointer to the next data node in `temp2`. Then, we deallocate `temp1`. If we deallocated `temp1`, first, we'd lose the pointer to the next data node.

Now that the data node pointed to by `temp1` has been deleted, it is no longer the first node in the linked list. As such, `head` should now point to `temp2`. Similarly, `temp2`'s `prev` pointer should point to `head`, instead of `temp1`. So, we update those two values.

The next thing we do in the `while` loop is to set `temp1` equal to `temp2`, so that in the next iteration, the next data node is processed using `temp1`.

This process is shown step by step in the Figure 3.

Now that our `clear()` method is implemented, let's implement the destructor. The only additional thing that the destructor must do is to deallocate the head and tail sentinel nodes.

```
template<typename T>
class List {
    protected:
        struct Node {
            T value;
            Node* next;
            Node* prev;
```

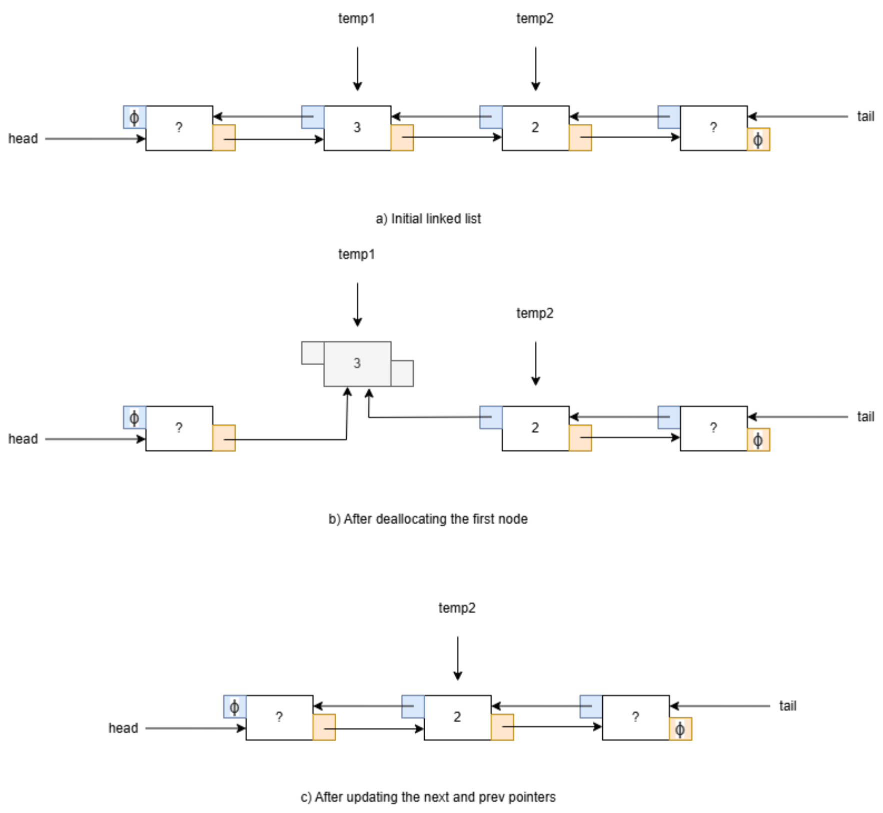

Figure 3: Removing the first node from the linked list

```
        };
        Node* head;
        Node* tail;
    public:
        List() {
            head = new Node;
            tail = new Node;
            head -> prev = nullptr;
            head -> next = tail;
            tail -> prev = head;
            tail -> next = nullptr;
        }
        void clear() {
            Node* temp1 = head -> next;
            while (temp1 != tail) {
                Node* temp2 = temp1 -> next;
                delete temp1;
                head -> next = temp2;
                temp2 -> prev = head;
                temp1 = temp2;
            }
        }
        ~List() {
            clear();
            delete head;
            delete tail;
        }
};
```

3.5 The Iterator

Now, we'll implement the iterator class. Did you notice how we were defining iterator variables with STL with something like `std::vector<int>::iterator it = list.begin();`? The `std::vector` part means that the `vector` class is a *member* of the `std` namespace. Does that mean the `iterator` is a member of the `vector` template class? You bet! But how do we define such a thing? We've already done that with the `Node` structure. Oh, so that's how! But, what if `iterator` is a class? No worries. We can define inner classes, just the same way.

What member variables must our iterator class have? Well, an iterator is like a pointer, so we should have a pointer variable defined. What should be the type of the pointer? The pointer points to a node in the linked list, so it should be a pointer to type `Node`. Here's what that gets us:

```
template<typename T>
class List {
    protected:
        struct Node {
            T value;
            Node* next;
            Node* prev;
        };
        Node* head;
        Node* tail;
    public:
        // List constructor, clear(), and destructor omitted

        class Iterator {
            friend class List;
            private:
                Node* ptr;
            public:
                Iterator(Node* p) : ptr(p) {}
        };
};
```

We declare `Iterator` in the `public` part of `List` so that code outside the `List` can create an iterator object. We need to be able to use an iterator in places like `main()`, right?

We declare `List` to be a `friend` of the `Iterator` class, so that in the `List` class, we may access its members directly. We define a pointer variable named `ptr`, as promised earlier. We define a constructor that accepts a `Node` pointer and stores it in the `ptr` pointer member variable.

What else should we be able to do with an iterator?

- Our iterator is to be a bidirectional iterator. We should be able to advance the iterator forward, or move it backward conveniently with the ++ and -- operators like `++it;`, or `it++;` or `--it;` or `it--;`.
- The iterator will never point to the head sentinel node.
- The iterator may occassionally point to the tail sentinel node. This is useful to know when we've reached the end of the linked list, or when we tried to find an element in the list that did not exist.
- The iterator is like a pointer to a stored item. We should be able to get the stored item itself using the * operator like `int value = *it;`.
- If we store a structure variable or class object in the list, we should be able to access its public members using the -> operator like `it -> name`.

These are all operator overloading tasks, so let's get to them one by one. Let's start with the ++ operator.

```
template<typename T>
class List {
    protected:
        struct Node {
            T value;
            Node* next;
            Node* prev;
        };
        Node* head;
        Node* tail;
    public:
        // List constructor, destructor etc.

        class Iterator {
            friend class List;
            private:
                Node* ptr;
            public:
                Iterator(Node* p) : ptr(p) {}
                Iterator& operator ++() {
                    if (ptr && ptr -> next)
                        ptr = ptr -> next;
                    return *this;
                }
        };
};
```

The iterator would not advance past the tail sentinel node, just as it doesn't in the STL containers. An iterator object is like a pointer pointing to some node in the linked list. The class `Iterator` has a `ptr` pointer variable to hold the pointer to that node. So, we first check if `ptr -> next` is not null. Recall that the tail sentinel node's `next` pointer is null. That indirection requires following the `ptr` pointer, so we must first check if `ptr` itself is not null, hence the logical AND in the `if` condition. If the `if` condition evaluates to `true`, our iterator is pointing to a node before the tail sentinel. Therefore, it is safe to advance it to point to the next node. We update the `ptr` pointer to `ptr -> next`. If we are at the sentinel node, we don't change `ptr`. Then, we return the current iterator object.

Can you guess whether the above operator is the pre-increment operator or the post-increment operator? If you guessed, pre-increment, you're right. How do we implement the post-increment operator?

```
template<typename T>
class List {
    protected:
        struct Node {
            T value;
            Node* next;
            Node* prev;
        };
        Node* head;
        Node* tail;
    public:
        // List constructor, destructor etc.

        class Iterator {
            friend class List;
            private:
                Node* ptr;
            public:
                Iterator(Node* p) : ptr(p) {}
                Iterator& operator ++() {
                    if (ptr && ptr -> next)
                        ptr = ptr -> next;
                    return *this;
                }
                Iterator operator ++(int) {
                    Iterator temp = *this;
                    ++(*this);
                    return temp;
                }
        };
};
```

We place the dummy parameter **`int`** in the definition of the post-increment operator. To handle the post-increment operator:

1. Preserve the Current State: We first make a copy of the iterator, called **`temp`**. This allows us to retain the current state before any modifications.

2. Advance the Iterator: We then advance the iterator itself using the pre-increment (++) operator, which we have already defined.

3. Return the Original State: Finally, we return the copied **`temp`** iterator, representing the original state of the iterator before it advanced.

This approach ensures that the post-increment operator behaves correctly, returning the iterator's original value while still advancing its position.

Note that this version of the ++ operator returns an **`Iterator`** object by value, rather than reference. Returning a local variable **`temp`** by reference wouldn't make sense, because it would be destroyed as soon as the function returns.

Could you implement the `--` operators on your own? Unlike the ++ operator, these should not reach the head sentinel node. In other words, if you keep decrementing an iterator, it should stop moving once it reaches the first data node. We'll provide the code for decrement operator, later. Right now, let's implement the bare-essential functionality to be able to test the code we've written.

We can start writing code that initializes an iterator at the beginning of a list, and then advances it until it reaches the end. The decrement operator can wait.

But, we'll need methods to compare two **`Iterator`** objects for equality or inequality. Let's implement those. When are two iterators equal? When they are pointing at the same node. Meaning, they have the same address stored in **`ptr`**. Here's the code:

```
template<typename T>
class List {
    protected:
        struct Node {
            T value;
            Node* next;
            Node* prev;
        };
        Node* head;
        Node* tail;
    public:
        // List constructor, destructor etc.

        class Iterator {
            friend class List;
            private:
                Node* ptr;
            public:
                Iterator(Node* p) : ptr(p) {}
                // Increment operators code omitted
                bool operator ==(const Iterator& other) const {
                    return (ptr == other.ptr);
                }

                bool operator != (const Iterator& other) const {
                    return !(*this == other);
                }
        };
};
```

Our equality comparison function is declared `const` since we don't intend to modify the iterator state in it. It has a return type of `bool` because it returns either `true` or `false`. It accepts another `Iterator` object as parameter by reference. The parameter is also declared as `const` as a best practice. We return `true` if the `ptr` pointer of the callee object and the `other` object are the same, otherwise we return `false`. The != operator makes use of the == operator.

Let's implement the indirection related operators, now.

```
template<typename T>
class List {
    protected:
        struct Node {
            T value;
            Node* next;
            Node* prev;
        };
        Node* head;
        Node* tail;
    public:
        // List constructor, destructor etc.

        class Iterator {
            friend class List;
            private:
                Node* ptr;
            public:
                // All constructor and operators code omitted
                T& operator *() const {
                    return ptr->value;
                }

                T* operator ->() const {
                    return &(ptr -> value);
                }
        };
};
```

The * operator returns the object stored in the linked list node that the iterator is currently pointing to. That object is stored in the `value` member of the `Node` structure.

The -> operator is responsible for accessing the element stored in the node that the iterator is currently pointing to. It achieves this by resolving the iterator object to a pointer that directly refers to the stored element. So, we return the address of `ptr -> value`. The compiler can then use this pointer to access any member variables or functions of the element stored in the linked list.

3.6 Accessing list iterators

We need to be able to get an iterator initialized to point to some valid node in the linked list from outside the `List` class. There are two possibilities, here. One is to get the iterator pointing to the first data node, and the other is to get the iterator pointing to the tail sentinel node. We'll do the first operation in a `begin()` function, and the latter in an `end()` function, just like STL containers.

```
template<typename T>
class List {
    // Protected members omitted
    public:
        // List constructor, destructor etc. and the Iterator class omitted
        Iterator begin() const {
            return Iterator(head->next);
        }
        Iterator end() const {
            return Iterator(tail);
        }
};
```

In the `begin()` function, we construct an `Iterator` object using the pointer to the first data node, and return it. In the `end()` function, we return an `Iterator` object mapped to the tail sentinel node.

3.7 The `insert()` function

We need a function to insert some data into the linked list. Unless we do that, we can't iterate over the linked list to do anything useful. Let's define an `insert()` function to do that. The `insert()` function takes two arguments: an iterator and a value. It inserts a new node containing the specified value immediately before the node that the provided iterator points to.

Here's how we can do this. It is a four step process as shown in Figure 4. First, a new node is created and the data value in it is initialized. This is shown in part (a) of Figure 4. Then, the address of the node pointed to by the iterator is copied into the new node's `next` pointer since it needs to be inserted before the iterator. Also, the address of the node before the one pointed to by the iterator is copied into the `prev` pointer of the new node. These steps are shown in part (b) of Figure 4. Dotted lines connect the variables involved in the copy operations. Next, we copy the new node's address into the `prev` pointer of the node pointed to by the iterator, and the `next` pointer of the one before it. That completes the process. This is shown in part (c) of Figure 4. We want to return an iterator pointing to the new node, so we need to create a new iterator object initialized to the new node's pointer and return it. The code is shown below. All four steps are clearly set apart by empty lines.

```
template<typename T>
class List {
    // Protected members omitted
    public:
        // Other methods omitted
        Iterator insert(const Iterator& pos, const T& val) {
            Node* newone = new Node;
            newone -> value = val;

            Node* current = pos.ptr;

            newone -> next = current;
            newone -> prev = current -> prev;
            current -> prev -> next = newone;
            current -> prev = newone;
```

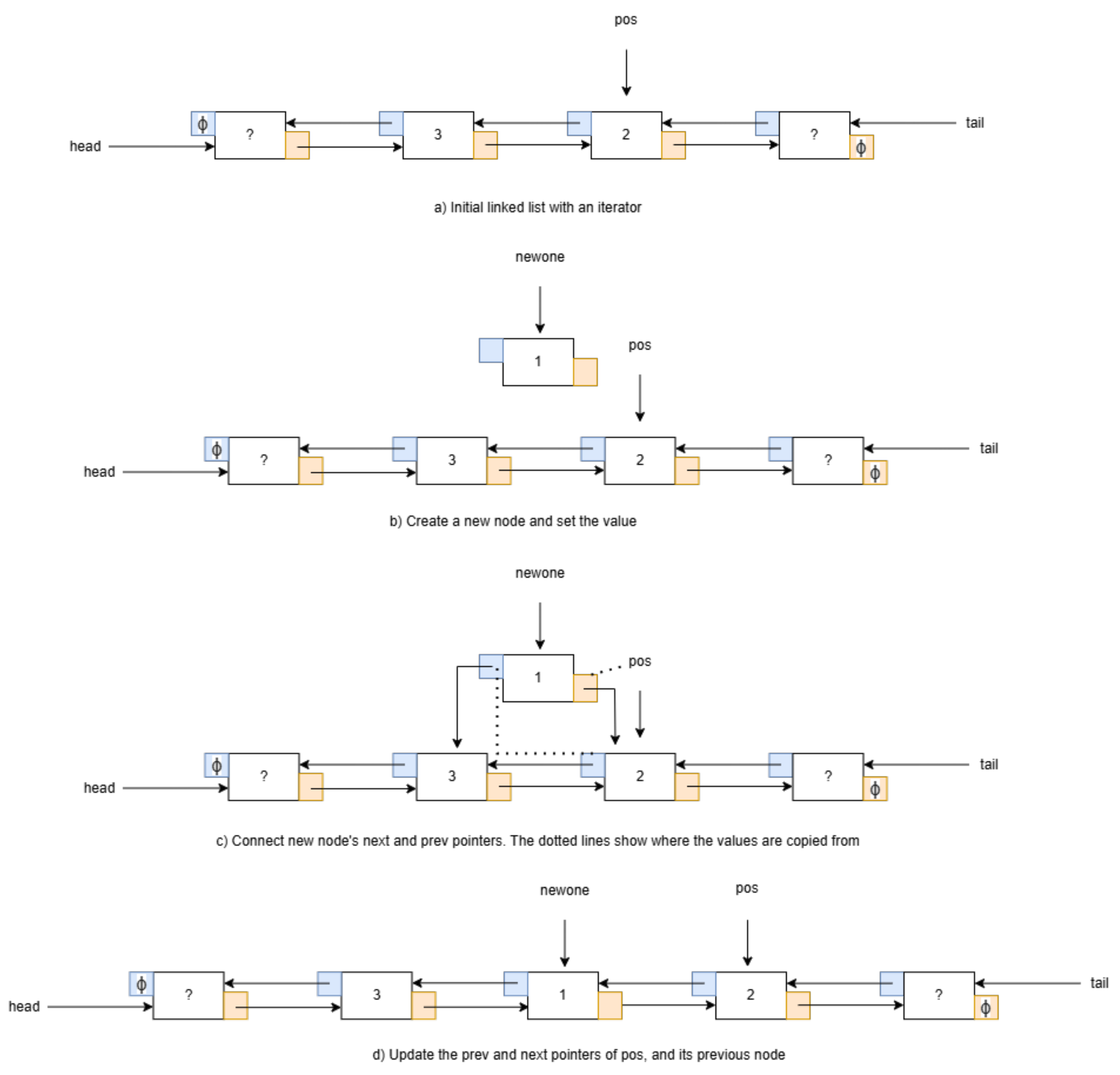
pos
head
?
3
2
?
tail
ϕ
ϕ
a) Initial linked list with an iterator
newone
1
pos
b) Create a new node and set the value
c) Connect new node's next and prev pointers. The dotted lines show where the values are copied from
d) Update the prev and next pointers of pos, and its previous node

Figure 4: Inserting a new node before the node corresponding to an iterator

```
            return Iterator(newone);
        }
};
```

3.8 Testing the implementation

With this code addition, we are ready to test our linked list class. We put all of the above code in a file named `list.h`. We create a file named `main.cpp` and write the following code in it:

```
#include <iostream>
#include "newlist.h"
int main() {
    List<int> list;
    List<int>::Iterator it = list.end();
    it = list.insert(it, 3);
    it = list.insert(it, 2);
    it = list.insert(it, 1);
    it = list.begin();
    while (it != list.end()) {
        std::cout << *it << " -> ";
        ++it;
    }
    std::cout << std::endl;

    for (auto i = list.begin(); i != list.end(); ++i) {
        std::cout << *i << " -> ";
    }
    std::cout << std::endl;

    list.clear();
    std::cout << "After clearing" << std::endl;
    for (auto i = list.begin(); i != list.end(); ++i) {
        std::cout << *i << " -> ";
    }
    std::cout << std::endl;
}
```

We instantiate a `List` object of template type `int`. We obtain an iterator object pointing to the end of this list. We use this iterator to insert the values, 3, 2, and 1, respectively, while updating the iterator to point to the newly inserted `int`. This should result in the values 1, followed by 2, and 3 in the list.

We use a `while` loop to iterate over the linked list, while using the * operator to display the stored value, and the ++ operator to advance the iterator.

We, then, use a `for` loop to iterate over the list from beginning to the end. You should see `1 -> 2 -> 3 ->` displayed twice on the screen when you run this program.

We are also testing out the `clear()` method and after a call to it, you don't see any values displayed by the **for** loop because after the initialization, the loop index variable `i` fails the test condition.

Now, let's test the `List` class with instances of a structure.

```
#include <iostream>
#include "newlist.h"
#include <string>

struct Person {
    std::string name;
    int age;
};

int main() {
    List<Person> people;
    Person p1;
    p1.name = "John Wick";
    p1.age = 48;

    Person p2;
    p2.name = "Jane Wick";
    p2.age = 32;

    Person p3;
    p3.name = "Allah Ditta";
    p3.age = 87;

    auto it2 = people.end();
    it2 = people.insert(it2, p1);
    it2 = people.insert(it2, p2);
    it2 = people.insert(it2, p3);
    for (it2 = people.begin() ; it2 != people.end(); it2++) {
        Person& p = *it2;
        std::cout << it2 -> name << " aged: " << it2 -> age << std::endl;
    }
}
```

This should work just as well. Icing on the cake would be if `Person` were a class with an overloaded « operator so that we could simply do `std::cout « *it2;` in the `for` loop.

3.9 The iterator -- operators

You should be able to implement the pre and post decrement (`--`) operators in the `Iterator` class. Here is the implementation for reference:

```cpp
template<typename T>
class List {
    // Protected members omitted
    public:
        // Other methods omitted
        Iterator& operator --() {
            if (ptr && ptr -> prev)
                ptr = ptr -> prev;
            return *this;
        }
        Iterator operator --(int) {
            Iterator temp = *this;
            --(*this);
            return temp;
        }
        // Other code omitted
};
```

Now, let's test the reverse iteration with the following code in the `main()` function:

```cpp
#include <iostream>
#include "newlist.h"
#include <string>

struct Person {
    std::string name;
    int age;
};

int main() {
    List<int> list;
    List<int>::Iterator it = list.end();
    it = list.insert(it, 3);
    it = list.insert(it, 2);
    it = list.insert(it, 1);

    for (auto i = list.begin(); i != list.end(); ++i) {
        std::cout << *i << " -> ";
    }
    std::cout << std::endl;

    std::cout << std::endl << "Backwards..." << std::endl;
    it = list.end() ;
    while(it != list.begin())  {
        std::cout << *it << " <- ";
        --it;
    }
    std::cout << *it << std::endl;
}
```

Now, you should see the list contents displayed in forward as well as reverse. The reverse iteration shows an additional 0 because the `end()` method returns an iterator pointing to the tail sentinel node. That is expected behavior. If that is not desirable, we can insert a `it--;` statement before the reverse iteration.

3.10 Copy constructor and assignment operator

It would be great if we could do something like `List<int> l2(l1);` or `l3 = l1;` to copy one linked list to another. Do we need to overload the copy constructor and assignment operator for that? The answer is yes, because we are managing dynamically allocated memory and the compiler provided versions of these operators do a shallow copy.

The first thing you should notice as you start writing the copy constructor is that you'll be repeating the code you wrote in the default constructor. That is not a good practice. So, we'll refactor that code out of the constructor and into a helper function. We'll put it into the `protected` part since we shouldn't have to call it from outside the class code.

```
template<typename T>
class List {
    protected:
        // Other declarations omitted
        void init() {
            head = new Node;
            tail = new Node;
            head -> prev = nullptr;
            head -> next = tail;
            tail -> prev = head;
            tail -> next = nullptr;
        }
    public:
        List() {
            init();
        }
        List(const List& other) {
            init();
            Iterator it1 = end();
            Iterator it2 = other.begin();
            while(it2 != other.end()) {
                insert(it1, *it2);
                it2++;
            }
        }
        // Other code omitted
};
```

We put the code previously in the constructor into a protected member function named `init()`. We insert a call to it in the default constructor. In the copy constructor, we insert a call to `init()` as well. That sets up the sentinel nodes in the new object. Now, onto copying the data from the `other` list into the new one. We get an iterator to the end of the new list, and one to the beginning of the `other` list. We iterate over the `other` list until we reach its end. In every iteration, we take the current node from `other` list (`*it2`) and insert it before the tail of the new list. We'll need to do the same thing in the = operator as well, so perhaps we should put this code in another helper function.

```
template<typename T>
class List {
    protected:
        // Other code omitted
        void copy(const List& other) {
            Iterator it1 = end();
            Iterator it2 = other.begin();
            while(it2 != other.end()) {
                insert(it1, *it2);
                it2++;
            }
        }
    public:
        List() {
            init();
        }
        List(const List& other) {
            init();
            copy(other);
        }
        void operator =(const List& other) {
            clear();
            copy(other);
        }
        // Other code omitted
};
```

We put the data copying code into a protected member function named `copy()`. We add calls to it from the copy constructor as well as the operator =. Now, let's test the copy constructor.

```
#include <iostream>
#include "newlist.h"
int main() {
    List<int> list;
    List<int>::Iterator it = list.end();
    it = list.insert(it, 3);
    it = list.insert(it, 2);
    it = list.insert(it, 1);
    it = list.begin();

    List<int> l2(list);
    auto itl2 = l2.end();
    l2.insert(itl2, 4);
    std::cout << "Copied one..." << std::endl;
    for (auto i = l2.begin(); i != l2.end(); ++i) {
        std::cout << *i << " -> ";
    }
    std::cout << std::endl;

    std::cout << "Original one..." << std::endl;

    for (auto i = list.begin(); i != list.end(); ++i) {
        std::cout << *i << " -> ";
    }
    std::cout << std::endl;
}
```

We copy the list that we'd created earlier into a new object named `l2`. To make sure that we're not working with a shallow copy, we insert a new value into the new list and display both the original and the new list, which turn out to be different.

Now, let's test the = operator as well:

```
#include <iostream>
#include "newlist.h"
int main() {
    List<int> list;
    List<int>::Iterator it = list.end();
    it = list.insert(it, 3);
    it = list.insert(it, 2);
    it = list.insert(it, 1);
    it = list.begin();

    List<int> l2;
    l2 = list;
    auto itl2 = l2.end();
    l2.insert(itl2, 4);
    std::cout << "Copied one..." << std::endl;
    for (auto i = l2.begin(); i != l2.end(); ++i) {
        std::cout << *i << " -> ";
    }
    std::cout << std::endl;

    std::cout << "Original one..." << std::endl;

    for (auto i = list.begin(); i != list.end(); ++i) {
        std::cout << *i << " -> ";
    }
    std::cout << std::endl;
}
```

Now, we first create an empty list and then copy the original list into it. Then, we add a new value into the new list and make sure that the two lists are independent.

3.11 Working with STL algorithms

Wouldn't it be great if we could use this collection class with the STL algorithms library. We have implemented a bidirectional iterator, so it should be able to work with functions like `std::find()`, `std::reverse()`, `std::copy()`, `std::equal()` etc.

To be able to work with a templatized container class like ours that has a bidirectional iterator implemented, the STL algorithm library requires certain tags to be applied to our class. All that requires is adding the following lines to the `Iterator` class definition in the `public` portion:

```
using iterator_category = std::bidirectional_iterator_tag;
using value_type        = T;
using difference_type   = std::ptrdiff_t;
using pointer           = T*;
using reference         = T&;
```

The first tag tells the compiler that this iterator is a bidirectional iterator, i.e., it supports both the `++` as well as the `--` operators. The second tag tells the compiler that the type of data (or value) over which this iterator iterates is the template type `T`. The third tag tells the compiler how to calculate the difference between the values of two iterator objects. Over here, we are saying that it should use the same arithmetic as it does for ordinary pointers. The next tag tells the compiler that the iterator will use `T*` as the pointer type, internally. Similarly, the last tag specifies the reference type for the values that the iterator iterates over.

Having added this code, we can now call several of the algorithms library functions on our `List` class. Here's an example:

```
#include <iostream>
#include "newlist.h"
int main() {
    List<int> list;
    List<int>::Iterator it = list.end();
    it = list.insert(it, 3);
    it = list.insert(it, 2);
    it = list.insert(it, 1);
    it = list.begin();
    int target = 30;

    it = std::find(list.begin(), list.end(), target);
    if (it != list.end()) {
        std::cout << target << " found in the list" << std::endl;
    }
    else {
        std::cout << target << " not found in the list" << std::endl;
    }

    target = 3;

    it = std::find(list.begin(), list.end(), target);
    if (it != list.end()) {
        std::cout << target << " found in the list" << std::endl;
    }
    else {
        std::cout << target << " not found in the list" << std::endl;
    }
}
```

We insert a few values into a `List` container object. We then call `std::find()` to look for a value that doesn't exist in the list, as well as a value that does. Running the program verifies that the `std::find()` function worked correctly on our container class with our custom iterator class.

Isn't this magical! We wrote our own code of which C++ had no prior knowledge whatsoever. Yet, its algorithms library works seamlessly with our container class.

Could you rewrite the `clear()` function using iterators? Earlier, we wrote it using direct pointer manipulation because we hadn't implemented the iterator class, yet.

3.12 The `erase()` function

An essential functionality for a container is to be able to remove an element from it. Let's implement an `erase()` function that accepts an iterator, deletes the node it points to, and returns an iterator pointing to the node after the deleted node.

```
template<typename T>
class List {
    // Other code omitted
    public:
        // Other code omitted
        Iterator erase(const Iterator& pos) {
            if (pos.ptr && pos.ptr -> next && pos.ptr -> prev) {
                pos.ptr -> prev -> next = pos.ptr -> next;
                pos.ptr -> next -> prev = pos.ptr -> prev;
                Iterator temp(pos.ptr -> next);
                delete pos.ptr;
                return temp;
            }
            return Iterator(nullptr);
        }
};
```

We first make sure that we are working with an iterator that points to a valid data node. If so, we connect its previous node to its next node. We create an `Iterator` object using its next node's pointer. We delete the node that the input iterator points to. Finally, we return the iterator to the next node. If the iterator doesn't point to a valid data node, we return an `Iterator` object pointing to null.

3.13 The `size()` method

Next, we want to implement a method that returns the current size of a `List` object. We can either maintain the size of the list as we insert and remove elements to it, or we can determine the size by traversing the list elements each time `size()` is called. The former approach is more efficient.

```
template<typename T>
class List {
    protected:
        struct Node {
            T value;
            Node* next;
            Node* prev;
        };
        Node* head;
        Node* tail;
        size_t sz;
        void init() {
            head = new Node;
            tail = new Node;
            head -> prev = nullptr;
            head -> next = tail;
            tail -> prev = head;
            tail -> next = nullptr;
            sz = 0;
        }
        void copy(const List& other) {
            Iterator it1 = end();
            Iterator it2 = other.begin();
            while(it2 != other.end()) {
                insert(it1, *it2);
                it2++;
            }
            sz = other.sz;
        }
    public:
        // Other code omitted
        void clear() {
            Node* temp1 = head -> next;
            while (temp1 != tail) {
                Node* temp2 = temp1 -> next;
                delete temp1;
                head -> next = temp2;
                temp2 -> prev = head;
                temp1 = temp2;
            }
            sz = 0;
        }
        Iterator insert(const Iterator& pos, const T& val) {
            Node* newone = new Node;
            newone -> value = val;

            Node* current = pos.ptr;

            newone -> next = current;
            newone -> prev = current -> prev;
            current -> prev -> next = newone;
            current -> prev = newone;

            sz++;
            return Iterator(newone);
        }
        Iterator erase(const Iterator& pos) {
            if (pos.ptr && pos.ptr -> next && pos.ptr -> prev) {
                pos.ptr -> prev -> next = pos.ptr -> next;
                pos.ptr -> next -> prev = pos.ptr -> prev;
                Iterator temp(pos.ptr -> next);
                delete pos.ptr;
                sz--;
                return temp;
            }
            return Iterator(nullptr);
        }
        int size() {
            return sz;
        }
        bool empty() {
            return sz == 0;
        }
};
```

We add a `sz` member variable. Each time we add a node, we increment it. Each time we remove a node, we decrement it. When we copy from another list, we copy the value of the other list's `sz` variable. While we're at it, we also implemented an `empty()` method that returns `true` if the list if empty, or `false` otherwise.

3.14 front() and back()

Many applications find it useful to peek at the first or last element in a container. Let's implement methods to perform these operations.

```
template<typename T>
class List {
    // Other code omitted
    public:
        // Other code omitted
        T& front() const {
            return *begin();
        }
        T& back() const {
            Iterator e = end();
            --e;
            return *e;
        }
};
```

In the `front()` method, we obtain an iterator to the first element, and then perform indirection on it to get the actual value stored there. In case of `back()`, the `end()` method returns an iterator that points to the tail sentinel. So, to get to the last data node, we decrement the iterator, then perform indirection on it.

4 Other tasks

Can you implement the following methods?

- `void push_front(const T& val)`: Inserts the value `val` to the front of the list.
- `void push_back(const T& val`: Inserts the value `val` to the end of the list.
- `void pop_front()`: Removes the first element from the list.
- `void pop_back()`: Removes the last element from the list.

5 Creating a stack using a doubly linked list

A stack is a last-in first-out (LIFO) data structure. It is a collection of items of the same type. But it does not provide random access to any item at all. Items can only be inserted at one end, and removed from that end. It is like a stack of trays in PDC. You pick the tray at the top, or you place a tray at the top.

The typical operations on a stack are:

- `bool empty()`: Returns `true` if the stack is empty, or `false` otherwise.
- `void push(T& val)`: Pushes the value of `val` on top of the stack.
- `T pop()`: Removes the value at the top of the stack and returns it.
- `T peek()`: Returns the value at the top of the stack without removing it.
- `size_t size()`: Returns the number of items currently in the stack.

How may we use our doubly linked list class to implement a stack? We should define a `Stack` class and use composition to create a `List` object in it. The methods `empty()`, and `size()` are simple pass-throughs. What about `push()` and `pop()`? We need to maintain the LIFO order, here. From `push()`, we could call `push_back()` on the `List` object. From `pop()`, we could call `pop_back()`. From `peek()`, we could call `back()` on the `List` object.

6 Creating a queue using a doubly linked list

Everybody knows what a queue is. Well, almost everybody does. You add an item only at one end (the rear) of the queue, and remove an item only from the other end (the front). Here are the common queue operations:

- `bool empty()`: Returns `true` if the queue is empty, or `false` otherwise.
- `void enqueue(T& val)`: Adds the value of `val` to the end of the queue.
- `T dequeue()`: Removes the value at the front of the queue and returns it.
- `T peek()`: Returns the value at the front of the queue without removing it.
- `size_t size()`: Returns the number of items currently in the queue.

Can you implement a queue as a class using composition just like we did for the stack?