CS 11 Data Structures and Algorithms

Lesson 20: Linked Lists

Skip to Main Content

Section 20.1: The intListType class

I don't have a detailed lesson for you on linked lists. Fortunately, the text does a very good job of explaining linked lists. My suggestion is that you start this week by setting aside a few hours to read carefully through the chapter in the text, making sure that you understand every line of code listed. (There is a lot of code in this chapter!) Then study the examples I've provided below. You'll see a lot in common with what you read in the text. Two things that the text doesn't provide are a copy constructor and an assignment operator. You'll see those demonstrated in the example below.

It's important that you study the examples carefully so that you understand everything that is going on. Otherwise you will almost certainly have difficulties with this week's assignment.

If you can take the time to watch the videos this week, you'll be in even better shape.

Here's our client program to test the class we are going to write. Make sure you study it carefully so you understand what it is that we need our class to do. I suggest you make a list of every public member of the class that we are going to need based just on looking at this client, before moving on to the header file.

```
// This is the file intlisttypeclient.cpp


#include <iostream>
#include "intlisttype.h"
using namespace std;
using namespace harden_intlist;


int main() {
    intListType list;

    int num;
    bool found;

    cout << "enter number to insert (negative to quit): ";
    cin >> num;
    while (num >= 0){
        list.insert(num);
        cout << "enter number to insert (negative to quit): ";
        cin >> num;
    }

    list.print();
    cout << endl;

    intListType::size_type numItems;
    numItems = list.size();
    cout << "There are " << numItems << " items." << endl;
    cout << "enter a number to find (negative to quit): ";
    cin >> num;
    while (num >= 0) {
        int result = list.find(num, found);
        if (!found) {
            cout << "not found" << endl;
        } else {
            cout << "found.  The data is " << result << endl;
        }
        cout << "enter a number to find (negative to quit): ";
        cin >> num;
    }


    cout << "enter a number to delete (negative to quit): ";
    cin >> num;
    while (num >= 0) {
```

```
        list.del(num, found);
        if (found) {
            cout << "the list is now ";
            list.print();
            cout << endl;
        } else {
            cout << num << " is not in the list." << endl;
        }
        cout << "enter a number to delete (negative to quit): ";
        cin >> num;
    }

    intListType list2(list);

    cout << "Now list 2 should be a copy of list.  Here it is: ";
    list2.print();
    cout << endl;


    list2.del(3, found);


    cout << "After deleting a 3 from list2, list2 is now: ";
    list2.print();
    cout << endl << "list should be unchanged.  Here it is: ";
    list.print();
    cout << endl;

    list = list2;

    cout << "Now list has been assigned list2 so it should match list2.  Here it is: ";
    list.print();
    cout << endl;

    list.del(4, found);

    cout << "After deleting a 4 from list, list is now: ";
    list.print();
    cout << endl << "list2 should be unchanged.  Here it is: ";
    list2.print();
    cout << endl;

}
```

Now the header file, followed by the implementation file

```
// Here is the file intlistype.h

#ifndef INT_LIST_TYPE_H
#define INT_LIST_TYPE_H
#include <cstdlib>

namespace harden_intlist {
    class intListType {
        public:
            typedef std::size_t size_type;
            typedef int value_type;
            intListType();
            void insert(const value_type& insertMe);
            void print() const;
            size_type size() const;
            value_type find(const value_type& findMe, bool& found) const;
            void del(const value_type& deleteMe, bool& found);
            void clear();
            void clone(const intListType& cloneMe);
            intListType(const intListType& copyMe);
            intListType operator=(const intListType& right);
            ~intListType();
        private:
            struct node {
                value_type data;
                node* next;
            };
            node* list;
    };
}
#endif
```

```
// Here is the file intlisttype.cpp
```

```
#include <iostream>
#include "intlisttype.h"
using namespace std;

namespace harden_intlist {
    intListType::intListType() {
        list = nullptr;
    }



    void intListType::insert(const value_type& insertMe) {
        node* temp = new node;
        temp -> data = insertMe;
        temp -> next = list;
        list = temp;
    }



    void intListType::print() const {
        node* tempPtr = list;
        while (tempPtr != nullptr) {
            cout << tempPtr -> data << " ";
            tempPtr = tempPtr -> next;
        }
    }


    intListType::size_type intListType::size() const {
        size_type count = 0;
        node* tempPtr = list;

        while (tempPtr != nullptr) {
            count++;
            tempPtr = tempPtr -> next;
        }

        return count;
    }


    intListType::value_type intListType::find(
                const value_type& findMe,
                bool& found) const {

        node* curPtr = list;
        while (curPtr != nullptr && curPtr -> data != findMe) {
            curPtr = curPtr -> next;
        }

        // assert: curPtr -> data == findMe || curPtr == nullptr

        if (curPtr != nullptr) {
            found = true;
            return curPtr -> data;
        } else {
            found = false;
            return value_type();
        }
    }


/* ALTERNATE VERSION OF FIND()

    intListType::value_type intListType::find(
                const value_type& findMe,
                bool& found) const {

        node* curPtr = list;
        while (curPtr != nullptr) {
            if (curPtr -> data == findMe) {
                found = true;
                return curPtr -> data;
            }
            curPtr = curPtr -> next;
        }
        found = false;
        return value_type();
    }
*/
```

```
void intListType::del(const value_type& deleteMe, bool& found) {
    node* prevPtr = list;

    if (list == nullptr) {
        found = false;
    } else if (list -> data == deleteMe) {
        node* tempPtr = list;
        list = list -> next;
        delete tempPtr;
        found = true;
    } else {
        while (prevPtr -> next != nullptr && prevPtr -> next -> data != deleteMe) {
            prevPtr = prevPtr -> next;
        }

        if (prevPtr -> next == nullptr) {
            found = false;
        } else {
            node* tempPtr = prevPtr -> next;
            prevPtr -> next = prevPtr -> next -> next;
            delete tempPtr;
            found = true;
        }
    }
}


intListType::intListType(const intListType& copyMe) {
    clone(copyMe);
}


intListType intListType::operator=(const intListType& right) {
    if (this != &right) {
        clear();
        clone(right);
    }
    return *this;
}


intListType::~intListType() {
    clear();
}



void intListType::clear() {
    node* tempptr = list;
    while (tempptr != nullptr) {
        list = list -> next;
        delete tempptr;
        tempptr = list;
    }
}



// Note: This function looks really long, but most of it is the explanations that
// I have added.  There are 13 lines of actual code.

void intListType::clone(const intListType& copyMe) {
    if (copyMe.list == nullptr) {
        list = nullptr;
    } else {

            // 1. Copy the first node in the original list

        list = new node;
        list -> data = copyMe.list -> data;

                    // 2. Get sourcePtr and newListPtr set up to copy the remaining nodes:
                    // 2a. newListPtr should always point to the last node we created in the new list
                    // 2b. sourcePtr should always point to the next node to be copied in the original list

        node* newListPtr = list;
        node* sourcePtr = copyMe.list -> next;

        // 3. Now copy the rest of the list

        while (sourcePtr != nullptr) {

            // 3a. Add a new node to the end of the new list
            // 3b. Move newListPtr down to the newly added node (to meet rule 2a above)
            // 3c. Populate the newly added node
```

```
                // 3d. Advance sourcePtr to the next node to be copied in the original list
                //     (to meet rule 2b above)

                newListPtr -> next = new node;
                newListPtr = newListPtr -> next;
                newListPtr -> data = sourcePtr -> data;
                sourcePtr = sourcePtr -> next;
            }

            // 4. Make sure the list ends with a nullptr

            newListPtr -> next = nullptr;
        }
    }
}
```

Section 20.2: The LL class

The STL has a container class named "list" that is implemented using linked lists. Next we will do an example that will be our own version of this class. (Technical note: actually what we will be implementing will be a version of the STL ***forward_list*** class. The STL ***list*** class is actually more complicated, since it allows traversing the list both forward and backward.)

Here's our client program to test the class we are going to write. Make sure you study it carefully so you understand what it is that we need our class to do.

```
#include <iostream>
#include "LL.h"
using namespace std;

void print(const LL<int>& printMe);


int main() {
    LL<int> l1, l2;

    l1.push_front(1000);
    LL<int>::iterator it = l1.begin();
    for (int i = 0; i < 10; i++) {
        l1.insert_after(it, i);
        l2.push_front(i);
    }

    cout << "l1: ";
        print(l1);

    cout << "l2: ";
        print(l2);

    cout << "Size of l1: " << l1.size() << endl;
    it = l1.begin();
    l1.delete_after(it);
    cout << "Setting iterator to first item in l1 and calling delete_after(): "
         << endl;
    cout << "l1: ";
        print(l1);
        cout << "New size of l1: " << l1.size() << endl;

        l1.front() = 1001;
        cout << "Using front to set first item in l1 to 1001: ";
        cout << "l1: ";
        print(l1);

        cout << "Using front() to print first item in l1: " << l1.front() << endl;

        cout << "Using empty(), pop_front(), and front() to print l1 and "
                 << "also clear it: " << endl;

        while (!l1.empty()) {
                cout << l1.front() << " ";
                l1.pop_front();
        }
        cout << endl;

        cout << "l1 should be empty, so testing Empty_List_Error on it: " << endl;
```

```
        try {
                l1.pop_front();
        } catch (LL<int>::Empty_List_Error e) {
                cout << "oops, tried to pop_front() from l1 but it was empty! " << endl;
        }
}


void print(const LL<int>& printMe) {
    for (LL<int>::const_iterator i = printMe.begin(); i != printMe.end(); i++) {
        cout << *i << " ";
    }
    cout << endl;
}
```

Now the header file and implementation file. Notice that there are two versions of the front() function. We'll see why in lesson 20.3

```
//  This is the file LL.h

#ifndef LL_h
#define LL_h

#include <cstdio>

template <class T>
class LL {
  public:
    typedef size_t size_type;
    typedef T value_type;

  private:
    struct node {
        value_type data;
        node* next;
    };
    node* list;


  public:
    class iterator {
      public:
        iterator(node* initial = nullptr) {
            current = initial;
        }

        value_type& operator*() const {
            return current->data;
        }

        iterator& operator++() {
            current = current->next;
            return *this;
        }

        iterator operator++(int) {
            iterator original(current);
            current = current->next;
            return original;
        }

        bool operator==(iterator other) const {
            return current == other.current;
        }

        bool operator !=(iterator other) const {
            return current != other.current;
        }

        const node* link() const {
            return current;
        }

        node*& link() {
            return current;
        }

      private:
        node* current;
    };
```

```
    class const_iterator  {
      public:
        const_iterator(const node* initial = nullptr) {
            current = initial;
        }

        const value_type& operator*() const {
            return current->data;
        }

        const_iterator& operator++() {
            current = current->next;
            return *this;
        }

        const_iterator operator++(int) {
            const_iterator original(current);
            current = current->next;
            return original;
        }

        bool operator==(const const_iterator other) const {
            return current == other.current;
        }

        bool operator !=(const const_iterator other) const {
            return current != other.current;
        }

        const node* link() const {
            return current;
        }

        node*& link() {
            return current;
        }
      private:
        const node* current;
    };  // end of iterator class declarations.  LL class continues below.

    iterator begin() {
        return iterator(list);
    }

    iterator end() {
        return iterator();
    }

    const_iterator begin() const {
        return const_iterator(list);
    }

    const_iterator end() const {
        return const_iterator();
    }

    class Empty_List_Error{};
    LL();
    LL(const LL& copyMe);
    LL operator=(const LL& right);
    ~LL();
    bool empty() const;
    size_type size() const;
    void clone(const LL& copyMe);
    void clear();
    void pop_front();
    void push_front(const value_type& item);
    value_type& front();
    const value_type& front() const;
    void insert_after(iterator& position, const value_type& insertMe);
    void delete_after(iterator& position);
};


#include "LL.cpp"
#endif



//  This is the file LL.cpp

#include <cassert>
#include <iostream>
```

```
using namespace std;

template <class T>
LL<T>::LL () {
    list = nullptr;
}

template <class T>
void LL<T>::delete_after(iterator& position) {
    assert (position.link() -> next != nullptr);
    node* tempptr = position.link() -> next;
    position.link() -> next = position.link() -> next -> next;
    delete tempptr;
}



template <class T>
void LL<T>::insert_after(iterator& position, const value_type& insertMe) {
    node* newnode = new node;
    newnode -> data = insertMe;
    newnode -> next = position.link() -> next;
    position.link() -> next = newnode;
    position.link() = newnode;
}



template <class T>
void LL<T>::pop_front() {
    if (empty()) {
        throw Empty_List_Error();
    } else {
        node* deleteMe = list;
        list = list->next;
        delete deleteMe;
    }

}




template <class T>
typename LL<T>::value_type& LL<T>::front() {
    if (empty()) {
        throw Empty_List_Error();
    } else {
        return list->data;
    }
}




template <class T>
const typename LL<T>::value_type& LL<T>::front()  const {
    if (empty()) {
        throw Empty_List_Error();
    } else {
        return list->data;
    }
}




template <class T>
void LL<T>::clear() {
    node* deleteMe;
    while (list != nullptr) {
        deleteMe = list;
        list = list->next;
        delete deleteMe;
    }
}
```

```
// Note this is an exact copy of the clone() function from intListType

template <class T>
void LL<T>::clone(const LL<T>& copyMe) {
        if (copyMe.list == nullptr) {
                list = nullptr;
        } else {
                list = new node;
                list -> data = copyMe.list -> data;

                node* newListPtr = list;
                node* sourcePtr = copyMe.list -> next;
                while (sourcePtr != nullptr) {
                        newListPtr -> next = new node;
                        newListPtr = newListPtr -> next;
                        newListPtr -> data = sourcePtr -> data;
                        sourcePtr = sourcePtr -> next;
                }

                newListPtr -> next = nullptr;
        }
}




// returns the number of items in the LL object.
template <class T>
typename LL<T>::size_type LL<T>::size() const {
    LL<T>::size_type numberOfItems = 0;
    node* curPtr = list;
    while (curPtr != nullptr) {
        ++numberOfItems;
        curPtr = curPtr->next;
    }
    return numberOfItems;
}




// returns true if the LL object is empty, false otherwise.
template <class T>
bool LL<T>::empty() const {
    return list == nullptr;
}




// insert x at the front of the LL object.
template <class T>
void LL<T>::push_front(const value_type& item) {
    node* oldList = list;
    list = new node;
    list->next = oldList;
    list->data = item;
}









// THE BIG-3

template <class T>
LL<T>  LL<T>::operator=(const LL<T> & right) {
    if (this != &right) {
        clear();
        clone(right);
    }
    return *this;
}
```

```
template <class T>
LL<T>::LL (const LL<T> & copyMe) {
    clone(copyMe);
}




template <class T>
LL<T>::~LL() {
    clear();
}
```

Section 20.3: Two versions of "front()"

Let's suppose that we want to add a member function to the intListType class named "front()" that returns the first item in an intListType object. We would like to be able to use this function in the usual way -- for example, "cout << list1.front();". But we'd also like this function call to be allowed on the left side of an assignment operator, so that we can use it to modify the first item in an intListType object -- for example, "list1.front() = 47;". (The STL "list" class has a member function named "front()" that behaves this way.) This will require us to have two versions of the front() function. In order to understand why we need two versions, consider the following client, which uses a hypothetical class named "LL".

```
#include <iostream>
#include <cassert>
#include "ll.h"
using namespace std;
using namespace cs_linkedlist;

int main() {

// part 1

     LL l1;
     l1.push_front(1);
     cout << "cout << l1.front(); should be 1: ";
     cout << l1.front() << endl;



 // part 2

     l1.front() = 2;
     cout << "cout << l1.front(); should be 2: ";
     cout << l1.front() << endl;



// part 3

     const LL l2(l1);
     cout << "cout << l2.front(), should be 2: ";
     cout << l2.front() << endl;


 // part 4

     l2.front() = 2;
     cout << "cout << l2.front(), should cause error: ";
     cout << l2.front() << endl;
}
```

Here is an explanation of why we cannot solve this problem with just one version of the front() function. We must have two version.

First note that the difference will be only in the function header. The bodies of these two functions will be identical.

Parts 1, 2, and 3 in the LL example above should work. Part 4 should give a syntax error. Make sure this makes sense before reading on.

1. If we only have the basic form, int front(), then we can't have front() on the left of an assignment operator (i.e., we want part 2 to work, but it will give a syntax error).
2. If we change it to int& front(), then front() cannot be called with a const LL calling object (i.e., we want part 3 to work, but it will give a syntax error).
3. If we change it to int& front() const, then a const LL calling object is allowed on the left side of an assignment operator (i.e., part 4 works, but should not).

The solution is to have 2 versions of front():

```
int front() const;
int& front();
```

One additional complication: Because we expect that we will be making this into a templated class, the return type may be an object. So, just like we pass parameters that are objects by const-reference instead of by value, we change the return type of our front() function from "return-by-value" to "return by const-reference". So, the final form will be:

```
const int& front() const;
int& front();
```