Linked Lists as a Class

Linked Lists

Earlier in the term we looked at Single Linked Lists as a set of functions called from main.

All of that material still holds when they are in a class, but the list is simpler to code in a class and easier to use. You just include the appropriate header (.h) file and then call the methods. All of the initialization and clean-up is done for you in the constructor and destructor.

Since head is a class variable, it is not necessary to pass it in to methods or have it returned, so the calling sequences are simpler also.

Basic structure

The underlying design is the same:

A linked list is a set of links joined using pointers between them. A **link** is the basic unit, much in the same way that a chain is composed of separate links joined together. Links can be implemented as structs or objects, in this example we are showing using structs to create the list.

The list consists of a pointer to the start of the list, typically called the **head**. Each **link** then contains some data and a pointer to the next item, typically labeled next. Finally the last link in the list points to **nullptr** to terminate the list. This is shown in the following diagram:

```
head -> link -> link -> link -> … -> link -> nullptr
```

Link Defintion

Since the link is a struct that is necessary for the Linked List class and not required for anyone else to know about, its definition is included in the header file for the linked list.

Here we will be defining a structure that contains the data a pointer to the next Link. Note that we can do this sort of circular references because all pointers are of the same length, so the compiler knows how much space to allocate for the entire structure

```
struct Link {
    int value;
    Link * next;
};
```

For demonstration purposes, value was defined to be an integer. In the real world, you could have it be any data type or combination of data types. If you were making a linked list of objects, you would normally use a pointer to the data value to reduce copying. For example, if using the class MyClass that contains a single integer value, your code might look like:

```
#include "MyClass.h"

struct Link {
    Link * next;
    MyClass * value;
};
```

Class List

Since the LinkedList is being defined in a class, we need to define head as a private variable, have a constructor, destructor, and any access methods that we want to use.

For this example, we are going with a simple list that allows adding a new value at the head, removing and returning the value from the head, searching the list to see if a given value is present, returning the number of links in the list, and returning a string that contains all of the values in the list in a formatted manner.

Given all the above, the class definition would be:

```
#include <string>
#include "MyClass.h"

struct Link {
    Link * next;
    MyClass * value;
};

class StructList {
private:
    Link *head;

    // optional recursive methods
    void recDestruct(Link * ptr);
    bool recFind(Link * ptr, int value);

public:
    StructList();
    ~StructList();

    void addValue(int value);
    bool findValue(int value);
    int removeHead();
    int getLength();
    std::string displayList();
};
```

The following example code is going to be showing both recursive and iterative methods for the destructor and findValue, so it is necessary to define recursive methods that are called from the public non-recursive methods.

If you were not going to use recursion for those two functions, you would not need those private methods.

Constructor

The only thing necessary in the constructor is to initialize the **head** to **nullptr**.

```
// constructor, defines head
StructList::StructList()
{
    head = nullptr;
}
```

Destructor

Since we are using dynamic memory, it is necessary to delete any remaining links when we are done with the list.

It starts at the head and walks down the list, deleting any links that are remaining. *If the links contain a pointer to an object on the heap, it is necessary to also delete that object.*

First a recursive version:

```
  // destructor, deletes items, using recursion
  StructList::~StructList()
  {
      recDestruct(head);
  }

// recursive method to walk down list, deleting items starting at tail
// if the link contains a pointer to an object, delete that object also
void StructList::recDestruct(Link * ptr)
{
    if ( ptr )
    {
        recDestruct(ptr->next);
        // if link->value is a pointer, delete what it points to
        delete ptr->value;
        delete ptr;
    }
}
```

Next a non-recursive version:

```
// destructor, deletes items, using iterative loop
// if the link contains a pointer to an object, delete that object also
StructList::~StructList()
{
        // start at the head
    Link * ptr = head;

        // continue until reach nullptr
    // version one
    // while ( ptr != nullptr )

    // version two
    while ( ptr )
    {
            // save the address of this link
            // update the pointer used to traverse the list
            // then delete the link
        Link * temp = ptr;
        ptr = ptr->next;
            // if the link contains a pointer, delete the object it points to
        delete temp->value;
        delete temp;
    }

}
```

Add at the head

The next method we need is one that adds a new value to the list. We add it at the head since that is the simplest way to add an item.

There are two cases for adding at the head, first the list is empty and you just create a new link and set the head equal to it.

```
head -> nullptr;
head -> newLink(value) -> nullptr;
```

The second is when you already have links in the list. Now you need to create the new link and set it to point to the existing link and then set head to point to the new link.

```
head -> link1 ->  … -> nullptr;
head -> newLink(value) -> link1 -> … -> nullptr;
```

Since they both set next of the newLink to the current value of head, you only need to code a single case.

First an example where we have an integer value in the struct:

```
//creates new link
//adds link to head of list
void StructList::addValue(int value)
{
        // create a new link
    Link * tempLink  = new Link;

    // update the link to point where head points
    // and give it the address of our temp object
    tempLink->value = value;
    tempLink->next = head;

    // finally, update head to point to the new object
    head = tempLink;

}
```

Next an example where we are adding a link that points to a new object of type MyClass. In this example, the data for that object is passed in and we must create an instance of the object on the heap. Another implementation would have the object created in the calling program and a link to it passed in.

```
//creates new link
//adds link to head of list
void StructList::addValue(int value)
{
        // since we are using a pointer to an object
        // create a new object with the value
    MyClass * temp = new MyClass(value);

        // now create a new link
    Link * tempLink = new Link;

        // update the link to point where head points
        // and give it the address of our temp object
    tempLink->value = temp;
    tempLink->next = head;

        // finally, update head to point to the new object
    head = tempLink;
}
```

Searching

There are two ways that we search an array, depending on whether it is ordered or not. If it is ordered, we can do a binary search. This requires the ability to continually divide the array into smaller segments using indices. Since linked lists do not have indices, we have to use the second method (linear) for searches, whether the list is ordered or not. Thus to do a search, we start at the head, check each link for the desired value and quit when we either find it or we reach the end.

First is the recursive version of search. It uses the private recursive recFind method:

```
// search to see if an object is present
// calls private recursive method
bool StructList::findValue(int value)
{
    return recFind(head, value);
}

// recursive method to search the list
bool StructList::recFind(Link * ptr, int value)
{
    // base case for recursion, quit if done
    if ( ptr == nullptr )
        return false;

    // not at the end -- two ways to code compare

    // version one, creates a temp object
    MyClass * temp = ptr->value;
    if (temp->getValue() == value )
        return true;

    // version two, uses double dereference
    if ( ptr->value->getValue() == value )
        return true;

    // not at the end, not the right now
    // continue searching with next link
    return recFind(ptr->next, value);
}
```

Next is an iterative version of the same method. It does not need a private method:

```
// search to see if an object is present
// uses iterative method to walk down the list
// This version is assuming that you have a simple integer value in your struct
// rather than a pointer to an object that contains the integer
bool StructList::findValue(int value)
{
        // This uses a while loop rather than a for loop

        // set pointer to start at the head
    Link * ptr = head;

        // continue until we get to the end
        // note this is using a compare rather than nullptr being false
    while ( ptr != nullptr )
    {
        if ( ptr->value == value )
            return true;
        ptr = ptr->next;
    }
    return false;

}
```

Remove from the head

The next method we need is one that removes the link at the head and returns its value. Note that if the return type depends on the type of the value in the Link struct. If the Link value is an integer, return an integer. If the Link value is a pointer to an object, return the pointer to the object. *If you return a pointer to an object, it is up to the calling program to delete that object.*

If the list is empty (head == nullptr), then you can simply return nullptr in the second instance, but you have to have some error condition defined (or throw an exception) in the first. After we have covered exception handling, you can simply throw an exception in either case.

The first instance is where Link contains an integer:

```
// remove the item at head and return its value
int StructList::removeHead()
{
        // deal with empty list
    if ( head == nullptr )
        return -1;

        // non-empty, so get first item and process
    Link * tempLink = head;
    int value = head->value;

        // update head
    head = head->next;

        // delete old link
    delete tempLink;

        // return the value
    return value;

}
```

The second is where Link contains a pointer to an object, it is identical except in the type of value and what we return on an empty list.

```
// remove the item at head and return its value
// this is where link contains a pointer to an object
MyClass * StructList::removeHead()
{
    // deal with empty list
    if ( head == nullptr )
        return nullptr;

    // non-empty, so get first item and process
    Link * tempLink = head;
    MyClass * value = tempLink->value;

    // update head
    head = head->next;

    // delete old link
    delete tempLink;

    // return the value
    return value;

}
```

Counting the links

This uses a simple for loop to walk down the list and see how many links. It takes advantage of the fact that the **boolean** value of **nullptr** is false.

```
// walk down the list and count how many links
// non-recursive version
int StructList::getLength()
{
        // initialize local variable for counter
    int counter = 0;

        // as an example, showing how to use a for loop instead of a while loop
        // note that nullptr evaluates as boolean to false
    for( Link * ptr = head; ptr; ptr = ptr->next )
        counter++;

        // and return the number of links found
    return counter;

}
```

Displaying

Like the counting method above, this uses a simple for loop to walk down the list. It uses stringstream to build up the output using the insertion operator in C++. Note that we have to explicitly get the value from MyClass, if we had defined an overloaded operator<< then we could have used it instead:

```
// build a string  by walking down the list
std::string StructList::displayList()
{
    std::stringstream buffer;

         // start at the head
    Link * ptr = head;

        // continue until at the end
    while ( ptr != nullptr )
    {
            // get the value from the object pointed to by this link
        buffer << ptr->value->getValue();

            // version two, if overloaded operator<< was defined
        // buffer << ptr->value;

            // update pointer
        ptr = ptr->next;

            // if not at end, add comma to string
        if ( ptr )
            buffer << ", ";
    }

        // now return the string
    return buffer.str();
}
```