To print higher-resolution math symbols, click the
Hi-Res Fonts for Printing button on the jsMath control panel.

Laboratory 2: Linked Lists

Week of Monday, September 9, 2013

Objective:

This laboratory introduces you to some advanced class development in C++, creating and using iterators, manipulating pointers, and linked data structures. It also addresses some issues involving testing and software development.

Background:

The linked list is a basic data structure from which one can implement stacks, queues, sets, and many other data structures. Lists may be singly- or doubly-linked. In this lab we will implement a doubly-linked list.

Reading(s):

1. Readings on the Collab wiki Readings page.
2. Tutorial 2: GDB, which is on the Collab wiki

Procedure

Pre-lab

1. Optionally read one of the online alternative readings shown on Collab.
2. Implement the three classes as described below: ListNode, TestListNode, and ListItr. You should have most, if not all, of the code working ***before*** coming to lab. TAs will be available to help in lab if you still have questions. Note that it is okay to not have the code perfectly working prior to lab – for the pre-lab grade, we are going to be looking to see if you have made significant progress, not that it is fully working (that's the post-lab).
3. If there is a particular method that is causing you a lot of trouble (i.e. you can't get it working), don't spend inordinate amounts of time on it – move on, and come back to that one during the in-lab.
4. **Making the implementation phase less frustrating:** develop in small chunks, then **test/debug incrementally. It is much easier to debug this way!** (And also less frustrating and confusing.) Here is a sample process for implementing ListNode.cpp:
    1. Implement the ListNode constructor in ListNode.cpp
    2. Write a short test harness, TestListNode.cpp, which has a main(). The body of main should test the ListNode constructor. In other words, create some ListNodes in main() using the constructor.
    3. Build and run the test program you just wrote to see if it produces the results you expect. Some items you will want to check are the initialization of the next and previous pointers and the initial value of value
    4. Use this same general process for List and for ListItr. For List and ListItr, implement the member functions one at a time and test. To do this, you will still need to provide "dummy" versions of the other member functions in your .cpp file as placeholders so that the code will build. **You will need to create a list of test cases to use to test your classes.** The TAs may ask you to test more cases during lab.
5. Read through the remainder of this document before coming to lab. Also read the tutorial on Unix debugging (called gdb-tutorial), as we will be using that during the in-lab.
6. Make sure you submit all 7 files listed below! Your code will not compile unless all 7 files are submitted. Also, if your code does not compile, and you cannot figure out why, comment out the erroneous code until it does compile. And make sure you have the right filename capitalization!
7. Files to download: List.h, ListNode.h, ListItr.h, ListTest.cpp
8. Files to submit: ListNode.h/cpp, ListItr.h/cpp, List.h/cpp, ListTest.cpp

In-lab

1. Carry out the tutorial on how to use Unix debuggers. The tutorial is in the 'tutorials' section of the Collab website, and is called gdb-tutorial. The debugger is an important tool that you will use extensively throughout the semester to debug your code. You will need to download the lab2.cpp file for the tutorial.
2. In the future, if you have a post-compilation problem with your program (crash, etc.), the TAs will not help you until you have run it through the debugger and learned all that can be learned from this. So make sure you understand the tutorial!
3. Submit your debugged version of lab2.cpp to inlab2. Remember the standard identifying header information. You should keep a copy of your files on Home Directory. ***This will count as your lab attendance credit, so be sure to do this before continuing.***

4. Verify to yourself that your methods are working properly with your linked list code using the debugger that you just learned about. If you have not yet completed your linked list implementation, use the debugger to help you identify the issues/problems with parts of your current implementation. You should do this by using the simple test cases that you used in the pre-lab. Consult with a TA if you have questions.
5. Files to download: lab2.cpp
6. Files to submit: lab2.cpp

Post-lab

1. For this lab you will be submitting your code electronically via online grading system to postlab2. Your fully functional code must contain the following 7 files:
    1. List.h and List.cpp
    2. ListNode.h and ListNode.cpp
    3. ListItr.h and ListItr.cpp
    4. and the test harness, ListTest.cpp
2. *Be sure you submit all 7 files!* If you don't, then your code will not compile properly, and you will lose points!
3. It is due on the Friday of the week of the lab, at the time listed on the Lab due dates page. Be sure to include: your name, the date, and the name of the file in a banner comment at the beginning of each file you submit.
4. Files to download: no additional files beyond the pre-lab and in-lab
5. Files to submit: ListNode.h/cpp, ListItr.h/cpp, List.h/cpp, ListTest.cpp

---

Pre-lab

Code Description

Linked lists are described on the online Readings.

The code in the book implements singly linked lists using a dummy header node. **You should implement your doubly linked list with dummy nodes as well** (not doing so will result in point deduction). You will want two dummy nodes – **one for the head** and **one for the tail**. A benefit of doing your implementation using dummy nodes is that there are fewer special cases to check for – for example you never have to update the head pointer on an insertBefore() or a remove() – the head pointer always points to the dummy header node. A dummy tail pointer would help out in the same respect. The downside is that you use two extra "empty" nodes. The book describes these issues on page 84.

For this lab you will need to implement three classes:

1. ListNode
2. List
3. ListItr

For simplicity we will just create a list that holds integers (your code could easily later be templated (i.e. made generic) to allow it to contain objects of other types). ***You must use the method names listed below in your code.***

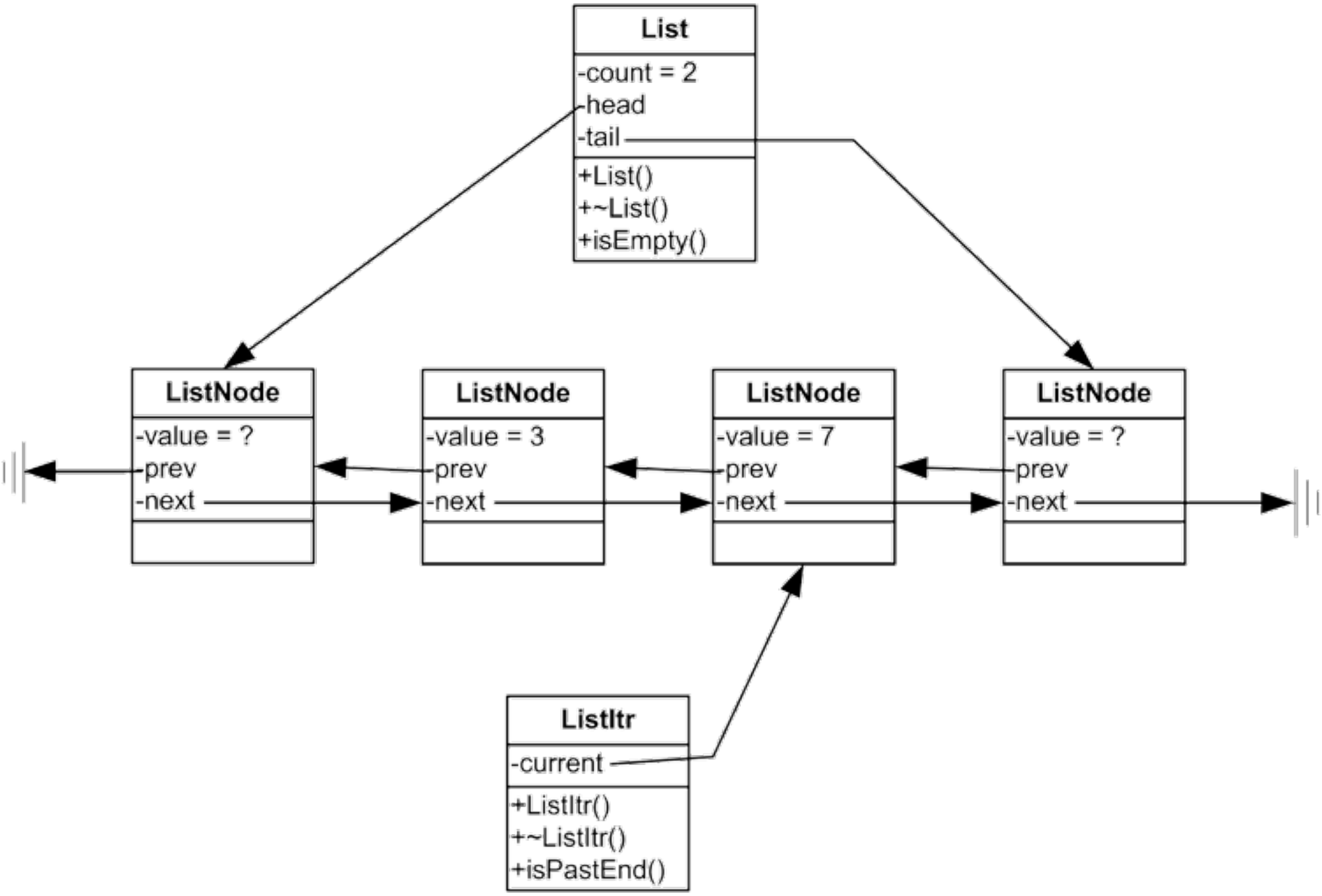
Below are the class definitions for each, which should be kept in header files with the respective filenames.

Test Harness

We have posted a test harness for testing your whole implementation, ListTest.cpp, on the labs page. ***The classes you implement must work with this test harness.***

UML Diagram

Below is a UML diagram showing how these classes interact with each other.



This diagram shows a list containing two elements, the integers 3 and 7. Note that there are more methods in the List and ListItr classes than what is shown above.

The head and tail pointers in the List class point to dummy nodes – they are put there to make inserting elements into the list easier. It doesn't matter what the value of the dummy notes is set to, as it won't be used. Each ListNode points to the nodes before and after it (although the dummy nodes each have one pointer pointing to NULL).

Thus, our doubly linked list will have only one List object and many ListNode objects (2 more than the number of elements in the list). A ListItr is a separate object, which points to one element in the list (possibly a dummy node). As you call the various methods in ListItr to move the iterator forward and backward, the node that it points to will change.

ListNode

A ListNode contains an integer value, as well as next and previous pointers to other ListNodes.

```
/*
 * Filename: ListNode.h
 * Description: ListNode class definition
 */
#ifndef LISTNODE_H
#define LISTNODE_H

// needed because NULL is part of std namespace
#include <iostream>
using namespace std;

class ListNode {
public:
    ListNode(); //Constructor
```

```
private:
    int value;
    ListNode *next, *previous;  //for doubly linked lists
    friend class List;  // List needs to be able to access/change
    // ListNode's next and previous pointers
    friend class ListItr;       // ListItr needs to access/change

    // Not writing a destructor is fine in this case since there is no
    // dynamically allocated memory in this class

};
#endif
/* end of ListNode.h listing */
```

List

This class represents the list data structure containing ListNodes. It has a pointer to the first (head) and last (tail) ListNodes of the list, as well as a count of the number of ListNodes in the List.

```
/*
 * Filename: List.h
 * Description: List class definition
 *      also includes the prototype for non-member function print()
 */
#ifndef LIST_H
#define LIST_H

#include <iostream>
#include "ListNode.h"
#include "ListItr.h"
using namespace std;

// When reading in ListItr.h first, it starts reading in this file
// before declaring that ListItr is a class.  This file then include
// ListItr.h, but beacuse of the #ifndef LISTITR_H statement, the code
// in that file is not read.  Thus, in this case, this List.h file
// will be read in, and will not know that ListItr is a class, which
// will cause compilation problems later on in this file.  Got it?
// Isn't C++ fun???
class ListItr;

class List {
public:
    List();                             //Constructor
    List(const List& source);   //Copy Constructor
    ~List();                    //Destructor
    List& operator=(const List& source);        //Equals Operator
    bool isEmpty() const;               //Returns true if empty; else false
    void makeEmpty();   //Removes all items except blank head and tail
    ListItr first();            //Returns an iterator that points to
    //the ListNode in the first position
    ListItr last();                     //Returns an iterator that points to
    //the ListNode in the last position
    void insertAfter(int x, ListItr position);
    //Inserts x after current iterator position p
    void insertBefore(int x, ListItr position);
    //Inserts x before current iterator position p
    void insertAtTail(int x);   //Insert x at tail of list
    void remove(int x);         //Removes the first occurrence of x
    ListItr find(int x);                //Returns an iterator that points to
    // the first occurrence of x, else
    // return a blank iterator

    int size() const; //Returns the number of elements in the list
```

```
private:
    ListNode *head, *tail;      //indicates beginning and end of the list
    int count;                  //#of elements in list
    friend class ListItr;
};

// printList: non-member function prototype
void printList(List& source, bool direction);
//prints list forwards when direction is true
//or backwards when direction is false
#endif
/* end of List.h */
```

Explanations:

1. List() is the default constructor. It should initialize all private data members. Pointers are often initialized to NULL.
2. ~List() is the destructor. It should make the list empty and reclaim the memory allocated in the constructor for head and tail
3. List& operator=(const List& source) is the **copy assignment operator**. It is called when code such as the following is encountered: lhs = rhs. The copy assignment operator that you implement will copy the contents of every ListNode in source into this (the reference to the calling List object itself)
4. List(const List& source) is the **copy constructor**. This will create a new list of ListNodes whose contents are the same values as the ListNodes in source.
5. bool isEmpty() This member function returns true if the list is empty, else false
6. void makeEmpty() removes/reclaims all items from a list, except the dummy head and tail nodes.
7. ListItr first() returns an iterator that points to the ListNode in the first position. This is the element **after** the head ListNode (even on an empty list!)
8. ListItr last() return an iterator that points to the ListNode in the last position. This is the element **before** the tail node (even on an empty list!)
9. void insertAfter(int x, ListItr position) inserts x **after** the current iterator position position.
10. void insertBefore(int x, ListItr position) inserts x **before** the current iterator position position.
11. void insertAtTail(int x) inserts x at tail of list.
12. void remove(int x) removes the first occurrence of x.
13. ListItr find(int x) returns an iterator that points to the first occurrence of x. When the parameter is not in the list, return a ListItr object, where the current pointer points to the dummy tail node. This makes sense because you can test the return from find() to see if isPastEnd() is true.
14. int size() returns the number of elements in the list.

In addition, you must implement this non-List member function: void printList(List theList, bool forward) is a **non-member function** that prints a list either forwards (by default – from head to tail) when forward is true, or backwards (from tail to head) when forward is false. *You must use your ListItr class to implement this function.*

Some of the harder methods…

The code for the copy constructor and the operator=() method in the List class are shown below. Although we are providing you with this code, you must understand how it works by the end of the lab, as you will have to implement these types of methods on future labs and exams.

```
List::List(const List& source) {                                    //Copy Constructor
    head=new ListNode;
    tail=new ListNode;
    head->next=tail;
    tail->previous=head;
    count=0;
    ListItr iter(source.head->next);
    while (!iter.isPastEnd()) {                             //deep copy of the list
        insertAtTail(iter.retrieve());
        iter.moveForward();
    }
}

List& List::operator=(const List& source)       { //Equals operator
    if (this == &source)
        return *this;
    else {
```

```
        makeEmpty();
        ListItr iter(source.head->next);
        while (!iter.isPastEnd()) {
            insertAtTail(iter.retrieve());
            iter.moveForward();
        }
    }
    return *this;
}
```

ListItr

Your ListItr should maintain a pointer to a current position in a List. Your iterator class should look like the class definition below.

```
/*
 * Filename: ListItr.h
 * Description: ListItr class definition
 */
#ifndef LISTITR_H
#define LISTITR_H

#include "ListNode.h"
#include "List.h"

class ListItr {
public:
    ListItr();                  //Constructor
    ListItr(ListNode* theNode); // One parameter constructor
    bool isPastEnd() const;     //Returns true if past end position
    // in list, else false
    bool isPastBeginning() const;//Returns true if past first position
    // in list, else false
    void moveForward(); //Advances current to next position in list
    //(unless already past end of list)
    void moveBackward();        //Moves current back to previous position
    // in list (unless already past beginning of
    // list)
    int retrieve() const;       //Returns item in current position

private:
    ListNode* current;          //holds the position in the list
    friend class List;  // List class needs access to “current”
    // ListNode's private data members
};

#endif
/* end of ListItr.h */
```

Your ListItr class should implement at least the following public methods:

- bool isPastEnd() Returns true if the iterator is currently pointing past the end position in the list (i.e., it's pointing to the dummy tail)
- bool isPastBeginning() Returns true if the iterator is currently pointing past(before) the first position in list (i.e., it's pointing to the dummy head)
- void moveForward() Advances the current pointer to the next position in the list (unless already past the end of the list)
- void moveBackward() Move current back to the previous position in the list (unless already past the beginning of the list)
- int retrieve() Returns the value of the item in the current position of the list

Hints

There are a few things that always cause students some headache. We’ve tried to explain some of them here, in an effort to lessen the frustration it causes.

When compiling your code, you must remember to compile all of your .cpp files in one line:

```
clang++ List.cpp ListItr.cpp ListNode.cpp ListTest.cpp
```

There are ways to compile these programs in pieces, but we will see this later in the semester.

Some students have had problems with the copy constructor and operator=() methods defined above – they would cause a crash (segmentation fault). The methods above work fine in our solution, but require that all the called functionality work properly as well. If it is causing a crash in your code, run it through the debugger to see where it crashes – it's probably a problem with one of your other methods.

In-lab

These are the same steps from the lab procedure section, above.

1. Carry out the tutorial on how to use Unix debuggers. The tutorial is in the 'tutorials' section of the Collab website, and is called gdb-tutorial. The debugger is an important tool that you will use extensively throughout the semester to debug your code. You will need to download the lab2.cpp file for the tutorial.
2. In the future, if you have a post-compilation problem with your program (crash, etc.), the TAs will not help you until you have run it through the debugger and learned all that can be learned from this. So make sure you understand the tutorial!
3. Submit your debugged version of lab2.cpp to inlab2. Remember the standard identifying header information. You should keep a copy of your files on Home Directory. ***This will count as your lab attendance credit, so be sure to do this before continuing.***
4. Verify to yourself that your methods are working properly with your linked list code using the debugger that you just learned about. If you have not yet completed your linked list implementation, use the debugger to help you identify the issues/problems with parts of your current implementation. You should do this by using the simple test cases that you used in the pre-lab. Consult with a TA if you have questions.
5. Files to download: lab2.cpp
6. Files to submit: lab2.cpp

Post-lab

These are the same steps from the lab procedure section, above.

1. For this lab you will be submitting your code electronically via online grading system to postlab2. Your fully functional code must contain the following 7 files:
    1. List.h and List.cpp
    2. ListNode.h and ListNode.cpp
    3. ListItr.h and ListItr.cpp
    4. and the test harness, ListTest.cpp
2. *Be sure you submit all 7 files!* If you don't, then your code will not compile properly, and you will lose points!
3. It is due on the Friday of the week of the lab, at the time listed on the Lab due dates page. Be sure to include: your name, the date, and the name of the file in a banner comment at the beginning of each file you submit.
4. Files to download: no additional files beyond the pre-lab and in-lab
5. Files to submit: ListNode.h/cpp, ListItr.h/cpp, List.h/cpp, ListTest.cpp

---

Untitled note

Find a notebook

Add tag

Add comments

Version 6.0.6: c128369/1.0.1.250

Web Clipper tutorial

Options

Saving clip...

To:

Share

Save

PDF