List Iterators

Lecture 36
Section 9.4

Robb T. Koether

Hampden-Sydney College

Fri, Apr 17, 2009
Outline

1. Sequential Access
2. List Iterators
3. The Iterator Class
4. Reverse Iterators
5. Assignment
Sequential Access of List Members

### A for Loop

```java
for (int i = 0; i < list.size(); i++)
    list[i] = 0;
```

- Consider the `for` loop above.
- How efficient is it if list is an `ArrayList`?
- How efficient is it if list is a `LinkedList`?
- Notice that we are accessing the members of the list *sequentially.*
A Possible Solution

**LinkedListwCurrPos Class**

```cpp
template <class T>
class LinkedListwCurrPos : LinkedList<T>
{
    LinkedListNode<T>* node;
};
```

- To improve sequential access, we need a way to “hold” a position in the list.
- We could introduce a new List data member that would refer to a position in the List.
  - The index of the current position, or
  - A pointer to the current node.
But we would be limited to holding only one position at a time.

What if we wanted to hold two or more positions, such as in a sort function?

This approach could quickly become very complicated.
Definition (List Iterator)

A list iterator is an object that is associated with a list and refers to a position in that list.

- The iterator uses the most efficient means available to do this, depending on the type of list.
- An array list iterator uses an index.
- A linked list iterator uses a node pointer.
Advantages of Iterators

- Since the iterator holds a position within the list, it can readily access that position’s successor, thereby greatly improving sequential access.
- Furthermore, as a separate object, we may create as many iterators for a list as we like.
List Iterator Behavior

- The iterator begins at one end of the list.
- The iterator advances one element at a time.
- The iterator stops when it moves *beyond* the other end of the list.

*Forward iterators* advance from head to tail.

*Reverse iterators* advance from tail to head.
List Iterator Behavior

- The list of elements

The list elements

30 50 60 10 90 70 20 40 80

1st position beyond the list
The (forward) iterator begins at the head.
Then it advances to position 1.
List Iterator Behavior

- Then to position 2.

The list elements

1st position beyond the list

30 50 60 10 90 70 20 40 80

Iterator
List Iterator Behavior

And so on...

The list elements

1st position beyond the list

Iterator
List Iterator Behavior

- And so on...

The list elements

30 50 60 10 90 70 20 40 80

1st position beyond the list

Iterator
List Iterator Behavior

- And so on...

The list elements

- 30
- 50
- 60
- 10
- 90
- 70
- 20
- 40
- 80

1st position beyond the list

Iterator
List Iterator Behavior

- And so on...

The list elements

1st position beyond the list

30 50 60 10 90 70 20 40 80

Iterator
List Iterator Behavior

- And so on...

The list elements

30 50 60 10 90 70 20 40 80

1st position beyond the list

Iterator
List Iterator Behavior

- And so on...

The list elements

| 30 | 50 | 60 | 10 | 90 | 70 | 20 | 40 | 80 |

1st position beyond the list

Iterator
Until it goes *beyond* the last position.

The list elements

```plaintext
30  50  60  10  90  70  20  40  80
```
The Iterator Class

- **We create the LinkedListwIter class as a subclass of the LinkedList class.**
- **We define the Iterator class within the LinkedListwIter class.**
The Iterator Class

class LinkedListwIter : public LinkedList
{
    public:
        class Iterator
        {
            public:
                Iterator();
                :
            
        public:
            // LinkedListwIter member functions
            private:
            // LinkedListwIter data members
        };
The Iterator Class

class LinkedListwIter : public LinkedList {
    public:
        class Iterator {
            public:
                Iterator();
            :
                    :
            ;
        }
    public:
        // LinkedListwIter member functions
    private:
        // LinkedListwIter data members
};
The Iterator Class

- This places the `Iterator` class within the scope of the `LinkedListIter` class.
- Therefore, the full name if the `Iterator` class is `LinkedListIter<T>::Iterator`
List Iterator Data Members

- `const` LinkedList<T>* mList;
  A pointer to the associated list.
- LinkedListNode<T>* node;
  A pointer to a node in the associated list.

- The data members have `protected` access.
- The mList data member is a constant.
- Therefore, it may be set only when the Iterator is constructed.
List Iterator Member Functions

- `Iterator(const LinkedListwIter& lst, LinkedListNode* p);`
  Constructs an iterator associated with a specified list.

- `bool isEqual(const Iterator& it) const;`
  Determines whether two iterators are equal.
List Iterator Member Functions

- `T& operator*();`
  Returns the list value pointed to by the iterator.

- `Iterator& operator++();`
  Advances the iterator to the next list element.
List Iterator Member Functions

- \texttt{bool operator== (const Iterator& it) const;}
  
  Compares two iterators for equality.

- \texttt{bool operator!=(const Iterator& it) const;}
  
  Compares two iterators for inequality.
Iterators and Iterators

List Iterators

Robb T. Koether

Sequential Access

List Iterators

The Iterator Class

Reverse Iterators

Assignment

LinkedListwIter Member Functions

- **Iterator begin()** `const;

  Returns a new iterator set to the beginning of this list.

- **Iterator end()** `const;

  Returns a new iterator set to the end of this list.
Sequential Access with Iterators

A for Loop

```cpp
for (Iterator it = list.begin();
     it != list.end(); ++it)
  *it = 0;
```

- Now consider the `for` loop again.
- How efficient is it if list is an `ArrayListWithIter`?
- How efficient is it if list is a `LinkedListWithIter`?
Additional List Member Functions

- \( T \) \text{getElement}(\text{const} \text{ Iterator}\& \text{curr}) \text{ const};
  Returns the list element that the Iterator is pointing to.

- \( T\& \) \text{getElement}(\text{const} \text{ Iterator}\& \text{curr});
  Returns a reference to the list element that the Iterator is pointing to.

- \text{void} \text{setElement}(\text{const} \text{ Iterator}\& \text{curr}, \text{const} \ T\& \text{value});
  Sets the value of the list element that the Iterator is pointing to.
Additional List Member Functions

- **T** operator[](Iterator& curr) **const**;
  Returns the list element that the Iterator is pointing to.

- **T&** operator[](Iterator& curr);
  Returns a reference to the list element that the Iterator is pointing to.

- **Iterator** searchIter(const T& value);
  Searches for the specified value and returns an Iterator to it if it is found. If it is not found, then the Iterator is equal to end().

- **void** sortIter();
  Sorts the list by using Iterators rather than indexes.
We can use the operator -- to back up to the previous list member.

For an `ArrayList iterator`,
- How would we do this?
- What would happen if we were at the head of the list?

For a `LinkedList iterator`,
- How would we do this?
- What would happen if we were at the head of the list?
Reverse Iterators

Definition (Reverse Iterator)

A reverse iterator is an iterator that works in the opposite direction.

- What does it mean for a reverse iterator to be at the “beginning” of a list?
- What does it mean for a reverse iterator to be at the “end” of a list?
- How would we increment a reverse iterator?
- How would we decrement a reverse iterator?
Homework

- Read Section 9.4, pages 475 - 491.