List Iterators

Lecture 28
Sections 17.1 - 17.3

Robb T. Koether

Hampden-Sydney College

Wed, Apr 4, 2018

1. Sequential Access
2. List Iterators
3. The Iterator Class
4. Example
5. Reverse Iterators
6. Assignment

Outline

1. Sequential Access
2. List Iterators
3. The Iterator Class
4. Example
5. Reverse Iterators
6. Assignment

Sequential Access of List Members

A for Loop

```
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*.

Outline

1. Sequential Access
2. List Iterators
3. The Iterator Class
4. Example
5. Reverse Iterators
6. Assignment

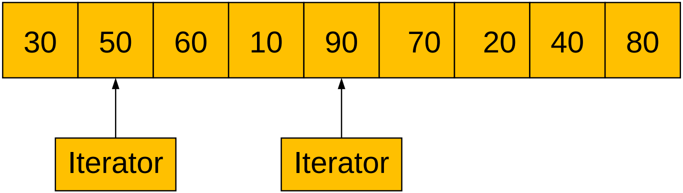
List Iterators

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, which depends on the type of list.
- An array list iterator uses an index.
- A linked list iterator uses a node pointer.

Advantages of Iterators



30	50	60	10	90	70	20	40	80

Iterator

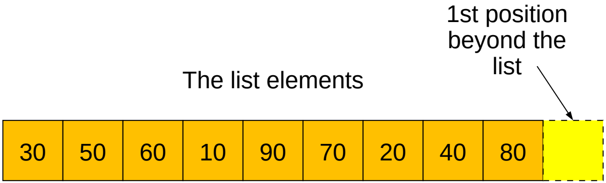
Iterator

- 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



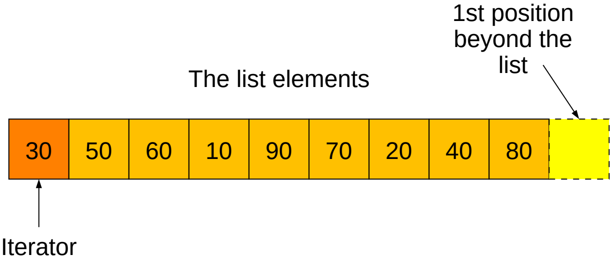
The list elements

1st position beyond the list

30	50	60	10	90	70	20	40	80	

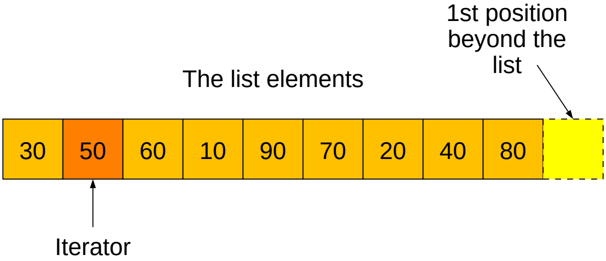
The list of elements

List Iterator Behavior



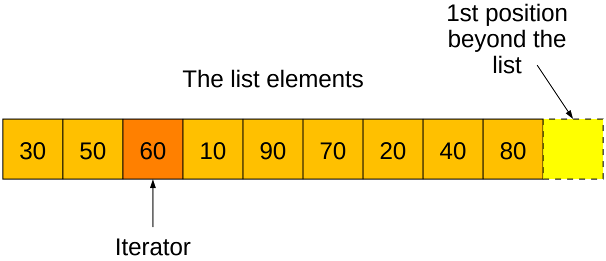
The (forward) iterator begins at the head.

List Iterator Behavior



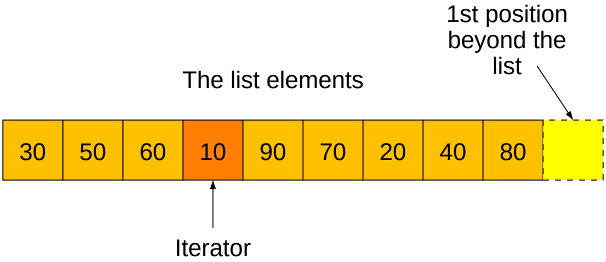
Then it advances to position 1.

List Iterator Behavior



Then to position 2.

List Iterator Behavior



1st position beyond the list

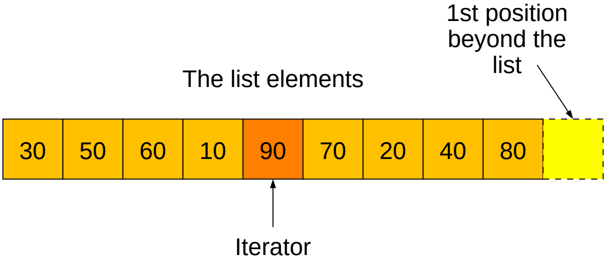
The list elements

30	50	60	10	90	70	20	40	80	

Iterator

And so on...

List Iterator Behavior



1st position beyond the list

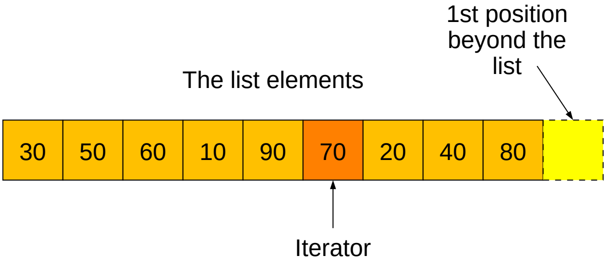
The list elements

30 50 60 10 90 70 20 40 80

Iterator

And so on...

List Iterator Behavior



1st position beyond the list

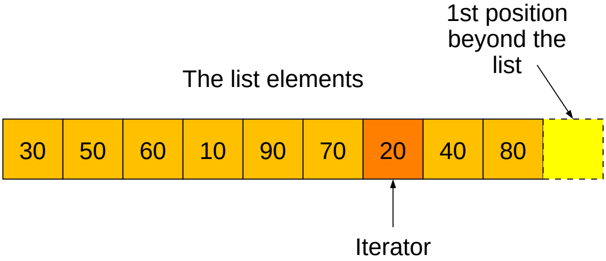
The list elements

30	50	60	10	90	70	20	40	80	

Iterator

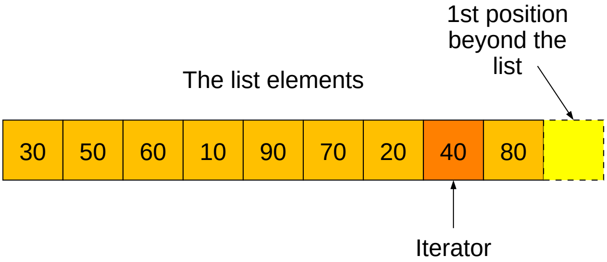
And so on...

List Iterator Behavior



And so on...

List Iterator Behavior



1st position beyond the list

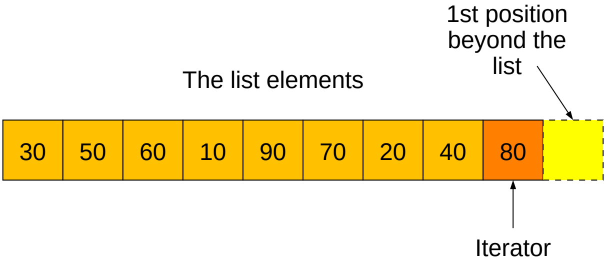
The list elements

30 50 60 10 90 70 20 40 80

Iterator

And so on...

List Iterator Behavior



1st position beyond the list

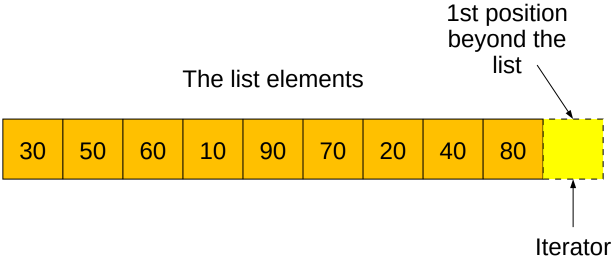
The list elements

30	50	60	10	90	70	20	40	80	

Iterator

And so on...

List Iterator Behavior



Until it goes *beyond* the last position.

Outline

1. Sequential Access
2. List Iterators
3. The Iterator Class
4. Example
5. Reverse Iterators
6. Assignment

The Iterator Class

- We create the `LinkedListwIter` class as a subclass of the `LinkedList` class.
- We define the `Iterator` class *within* the `LinkedListwIter` class.
- Thus, the `Iterator` class is a dependent class, depending on the `LinkedListwIter` class.

The Iterator Class

The Iterator Class

```
class LinkedListwIter : public LinkedList
{
// Iterator class definition
    public:
        class Iterator
        {
            public:
                Iterator();
                  ⋮
        };
//  LinkedListwIter class definition
    public:
    //  LinkedListwIter member functions
    private:
    //  LinkedListwIter data members
};
```

The Iterator Class

The Iterator Class

```
class LinkedListwIter : public LinkedList
{
// Iterator class definition
    public:
        class Iterator
        {
            public:
                Iterator();
                  ⋮
        };
//  LinkedListwIter class definition
    public:
    //  LinkedListwIter member functions
    private:
    //  LinkedListwIter data members
};
```

The Iterator Class

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

List Iterator Data Members

List Iterator Data Members

```
const LinkedList<T>* m_list;
LinkedListNode<T>* m_node;
```

- `m_list` – A pointer to the associated list.
- `m_node` – A pointer to a node in the associated list.
- The data members have **`protected`** access.
- Note that `m_list` is constant.
- Therefore, once it is set by the `Iterator` constructor, it cannot be changed.

List Iterator Data Members

List Iterator Data Members

```
const LinkedList<T>* m_list;
LinkedListNode<T>* m_node;
```

- `m_list` – A pointer to the associated list.
- `m_node` – A pointer to a node in the associated list.
- The data members have **`protected`** access.
- Note that `m_list` is constant.
- Therefore, once it is set by the `Iterator` constructor, it cannot be changed.
- Why is that a good idea?

List Iterator Constructors

List Constructors

```
Iterator(const LinkedListwIter& lst,
    LinkedListNode* p);
```

- `Iterator(LinkedListwIter, LinkedListNode*)` – Constructs an iterator associated with a specified list.

List Iterator Facilitators

List Iterator Facilitators

```
bool isEqual(const Iterator& it) const;
```

- `isEqual()` – Determines whether two iterators are equal.

List Iterator Member Functions

List Iterator Member Functions

```
T& operator*();
Iterator& operator++();
```

- **operator***() – Returns the list value pointed to by the iterator.
- **operator**++() – Advances the iterator to the next list element.

List Iterator Member Functions

List Iterator Member Functions

```
bool operator==(const Iterator& it) const;
bool operator!=(const Iterator& it) const;
```

- `operator==()` – Compares two iterators for equality.
- `operator!=()` – Compares two iterators for inequality.

LinkedListwIter Member Functions

LinkedListwIter Member Functions

```
Iterator begin() const;
Iterator end() const;
```

- `begin()` – Returns a new iterator set to the beginning of this list.
- `end()` – Returns a new iterator set to the end of this list.

Sequential Access with Iterators

A for Loop

```
typedef LinkedListwIter<int>::Iterator Iterator;
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 `ArrayListwIter`?
- How efficient is it if list is a `LinkedListwIter`?

Additional List Member Functions

Additional List Member Functions

```
T element(const Iterator& curr);
T& element(const Iterator& curr);
```

- `T element() const` – Returns a copy of the list element that the Iterator is pointing to.
- `T& element()` – Returns a reference to the list element that the Iterator is pointing to.

Additional List Member Functions

Additional List Member Functions

```
T operator[](Iterator& curr) const;
T& operator[](Iterator& curr);
```

- `T operator[]() const` – Returns a copy of the list element that the Iterator is pointing to.
- `T& operator[]()` – Returns a reference to the list element that the Iterator is pointing to.

Outline

1. Sequential Access
2. List Iterators
3. The Iterator Class
4. Example
5. Reverse Iterators
6. Assignment

Iterator Example

Selection Sort with Indexes

```
void sort()
{
    int base = 0;
    while (base < size)
    {
        int min = base;
        int curr = min;
        while (++curr < size)
            if (element[curr] < element[min])
                min = curr;
        T temp = element[base];
        element[base] = element[min];
        element[min] = temp;
        ++base;
    }
}
```

Iterator Example

Selection Sort with Pointers

```
void sort()
{
    LinkedListNode<T>* base = m_head;
    while (base != NULL)
    {
        LinkedListNode<T>* min = base;
        LinkedListNode<T>* curr = min;
        while ((curr = curr->m_next) != NULL)
        if (curr->m_value < min->m_value)
            min = curr;
        T temp = base->m_value;
        base->m_value = min->m_value;
        min = temp;
        base = base->m_next;
    }
}
```

Iterator Example

Selection Sort with Iterators

```
void sort()
{
    Iterator base.begin();
    while (base != end())
    {
        Iterator min = base;
        Iterator curr = min;
        while (++curr != end())
            if (*curr < *min)
        min = curr;
        T temp = *base;
        *base = *min;
        *min = temp;
        ++base;
    }
}
```

Additional List Member Functions

Additional List Member Functions

```
Iterator searchIter(const T& value);
void sortIter();
```

- `searchIter()` – 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()`.
- `sortIter()` – Sorts the list by using Iterators rather than indexes.

Decrementing an Iterator

- 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?

Outline

1. Sequential Access
2. List Iterators
3. The Iterator Class
4. Example
5. Reverse Iterators
6. Assignment

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?

Outline

1. Sequential Access
2. List Iterators
3. The Iterator Class
4. Example
5. Reverse Iterators
6. Assignment

Assignment

Homework

- Read Sections 17.1 - 17.3.