CMSC 341
Lecture 5

Announcements

List.H

```
Template <class Object>
class List
public:
  List()
  List(const List &rhs)
  ~List()
  const List &operator=(const List &rhs)
  Bool isEmpty() const
  void makeEmpty()
  void remove (const Object &x)
  void insert (const Object &x,
            const listIter<Object> &p)
```

List.H (cont)

```
  ListIter<Object> first()const;
  ListIter<Object> zeroth()const;
  void insert (const Object &x,
            const listIter<Object> &p);
  ListIter<Object> find(const Object &x);
  ListIter<Object> findPrevious(const Object &x);
private:
  ListNode<Object> *header;
```

ListNode.H

```
Template <class Object>
class ListNode {
   ListNode(const Object &theElement=Object(),
        ListNode *n=N):element(theElement), next(n) {}

   Object element;
   ListNode *next;

   friend class List<Object>;
   friend class ListItr<Object>;
   };
```

ListItr.H

```
template <class Object>
class ListItr {
public:
   ListItr():current(NULL) {}
   bool isPastEnd() const {return current == NULL;}
   void advance() {
        if (!isPastEnd()) current = current->next;}
   const Object &retrieve() const {
        if (isPastEnd()) throw BadIterator();
        return current->element;}
private:
   ListNode<Object> *current;
   ListItr(ListNode<Object> *theNode):current(theNode){}
   friendclass List<Object>;
}
```

Linked List Implementation

```
template <class Object>
List<Object>::List() {
  header = new ListNode<Object>;
}

template <class Object>
List<Object>::isEmpty() {
  return !(header->next);
}
```

Linked List Implementation (cont)

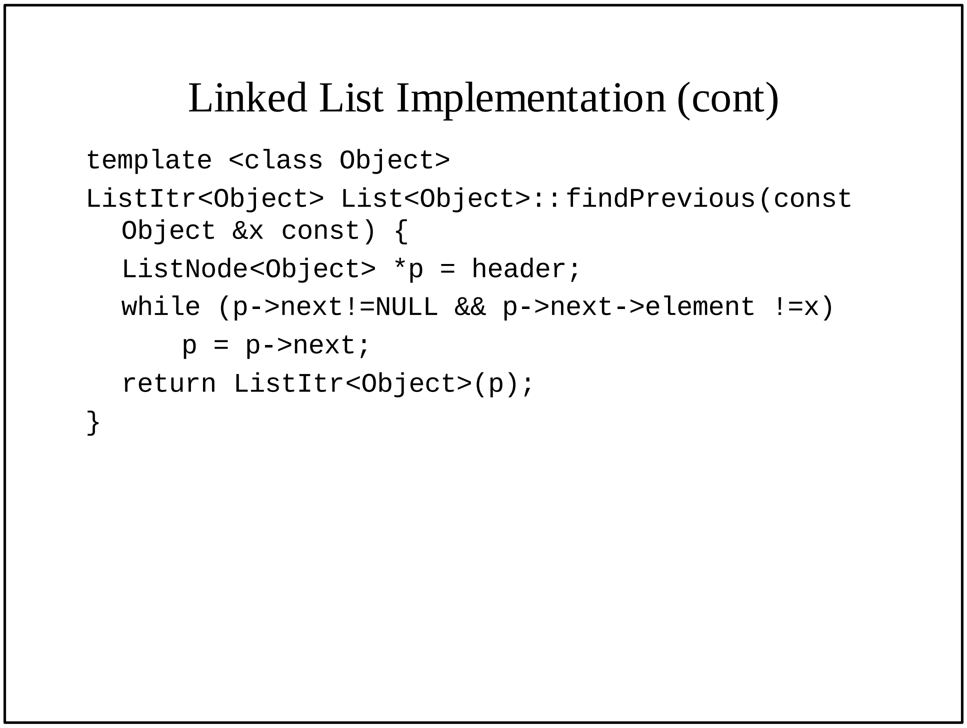
```
template <class Object>
ListItr<Object List<Object>::zeroth() {
  return ListItr<Object>(header);
}

template <class Object>
ListItr<Object> List<Object>::first() {
  return  ListItr<Object>(header->next);
  }
```

Linked List Implementation (cont)

```
template <class Object>
void List<Object>::insert(const Object &x,
      const ListItr<Object> &p) {
  if (!p.isPastEnd()) //text uses p.current!=NULL
}
```

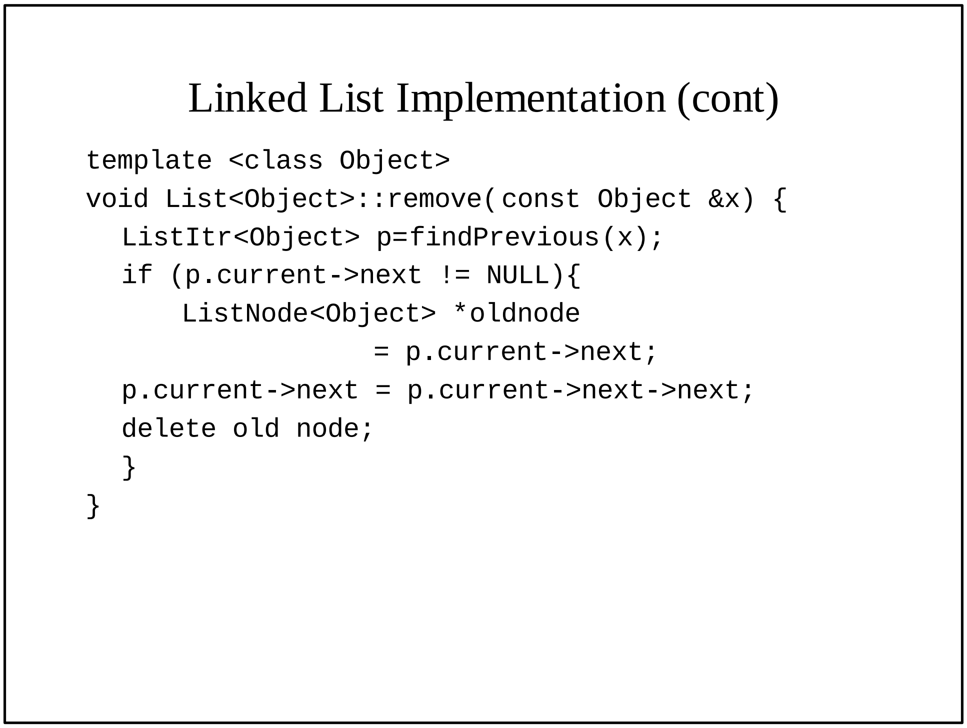
Linked List Implementation (cont)

```
template <class Object>
ListItr<Object> List<Object>::find(const Object
  &x const) {
  ListNode<Object> *p = header->next;
  while (p!=NULL && p->element !=x)
      p = p->next;
  return ListItr<Object>(p);
}
```



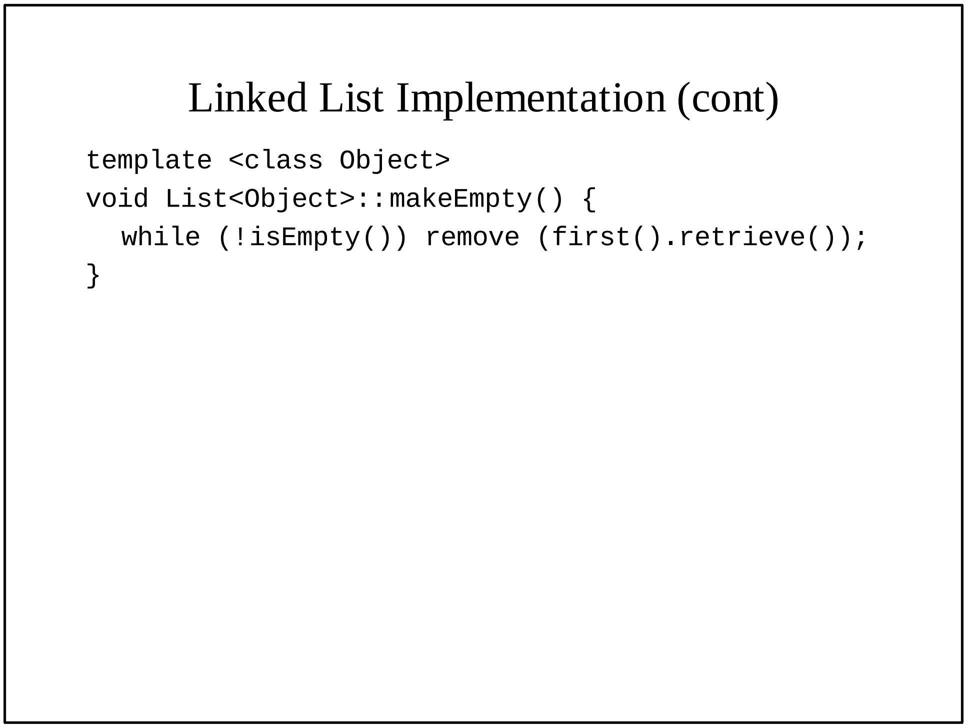
Linked List Implementation (cont)

```
template <class Object>
ListItr<Object> List<Object>::findPrevious(const
  Object &x const) {
  ListNode<Object> *p = header;
  while (p->next!=NULL && p->next->element !=x)
      p = p->next;
  return ListItr<Object>(p);
}
```



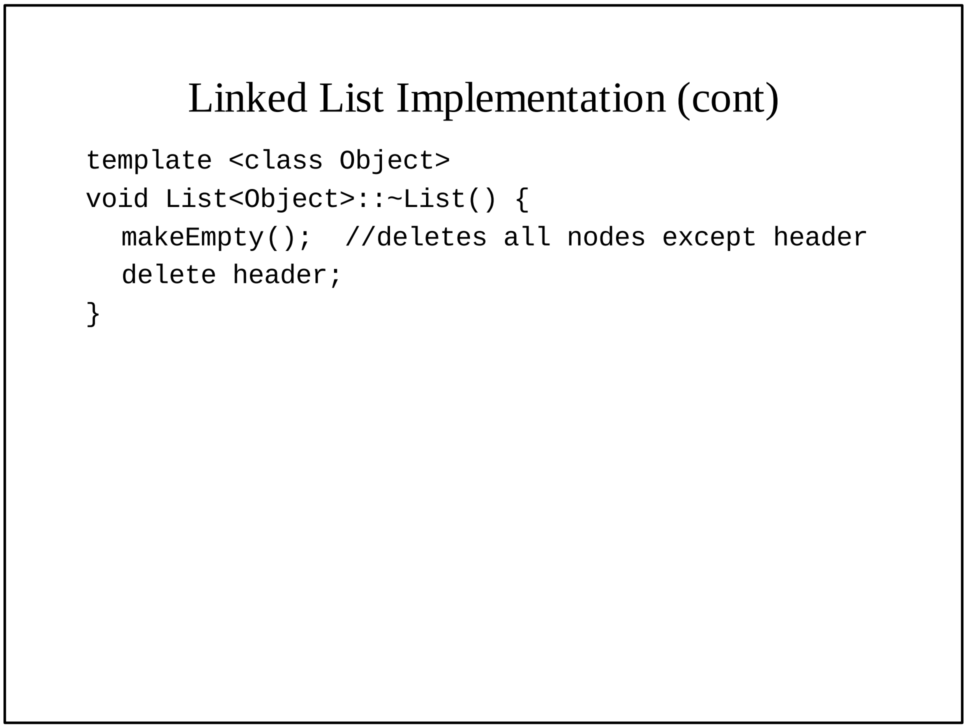
Linked List Implementation (cont)

```
template <class Object>
void List<Object>::remove(const Object &x) {
  ListItr<Object> p=findPrevious(x);
  if (p.current->next != NULL){
      ListNode<Object> *oldnode
                  = p.current->next;
  p.current->next = p.current->next->next;
  delete old node;
  }
}
```



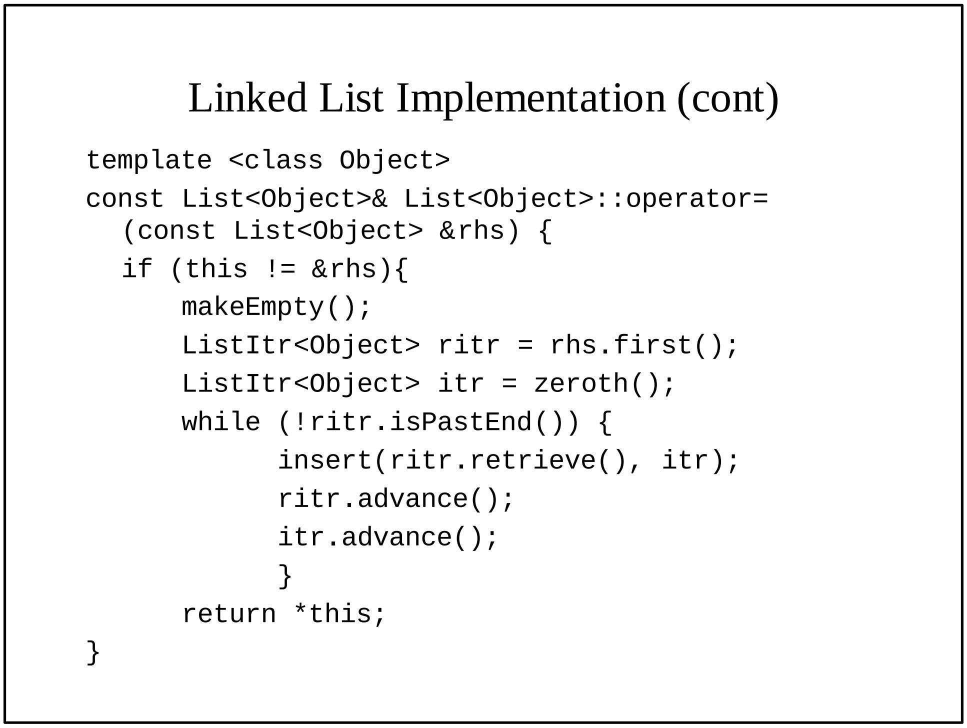
Linked List Implementation (cont)

```
template <class Object>
void List<Object>::makeEmpty() {
  while (!isEmpty()) remove (first().retrieve());
}
```



Linked List Implementation (cont)

```
template <class Object>
void List<Object>::~List() {
  makeEmpty();  //deletes all nodes except header
  delete header;
}
```



Linked List Implementation (cont)

```
template <class Object>
const List<Object>& List<Object>::operator=
  (const List<Object> &rhs) {
  if (this != &rhs){
      makeEmpty();
      ListItr<Object> ritr = rhs.first();
      ListItr<Object> itr = zeroth();
      while (!ritr.isPastEnd()) {
            insert(ritr.retrieve(), itr);
            ritr.advance();
            itr.advance();
            }
      return *this;
}
```