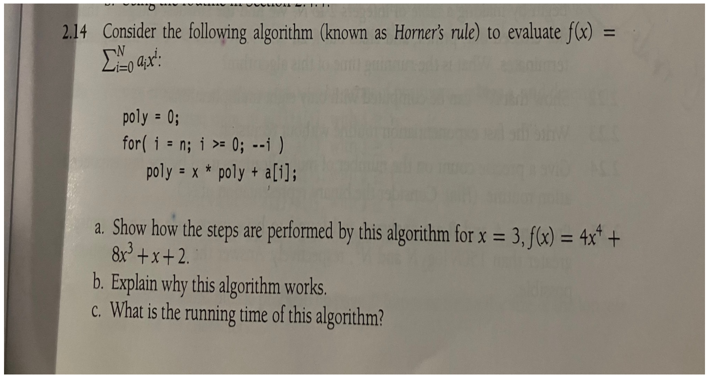
CSCI 335
Software Design and Analysis III
Lecture 7: Lists and the STL

Professor Anita Raja
09-19-22

Agenda

- Homework Exercises
- Iterators and const_iterators
- Vector and List Implementation
  - Vector vs List in STL
  - Iterators
- Trees Intro

Exercise 2.14



2.14 Consider the following algorithm (known as *Horner's rule*) to evaluate $f(x) = \sum_{i=0}^{N} a_i x^i$:

```
poly = 0;
for( i = n; i >= 0; --i )
    poly = x * poly + a[i];
```

a. Show how the steps are performed by this algorithm for $x = 3$, $f(x) = 4x^4 + 8x^3 + x + 2$.

b. Explain why this algorithm works.

c. What is the running time of this algorithm?

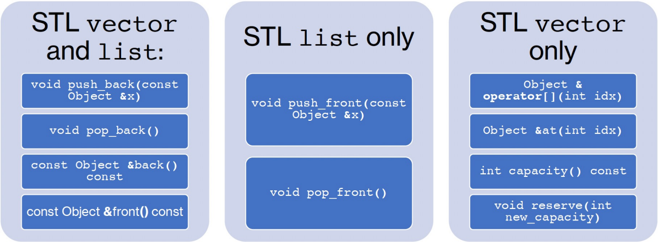
Homework Exercises

- Order the following functions by growth rate:

N, $N^{1/2}$, $N^{1.5}$, N^2, $N \log N$, $N \log \log N$, $N \log^2 N$, $N \log(N^2)$, $2/N$, 2^N , $2^{N/2}$, 37, $N^2 \log N$, N^3

- Find 2 functions f(N) and g(N) such that neither f(N) = O((g(N)) nor g(N)= O(f(N)

STL vector vs list



STL vector and list:

- void push_back(const Object &x)
- void pop_back()
- const Object &back() const
- const Object &front() const

STL list only

- void push_front(const Object &x)
- void pop_front()

STL vector only

- Object & **operator**[](int idx)
- Object &at(int idx)
- int capacity() const
- void reserve(int new_capacity)

Implementation of a vector

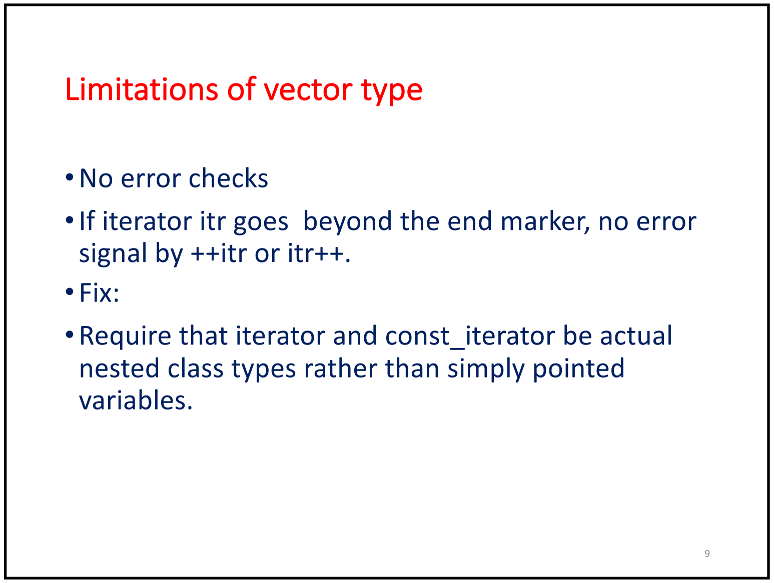
- How?
- Dynamic array
  - Deep copy (copy constructor, operator=, destructor)
  - resize and reserve
  - overload operator[]
  - iterator and const_iterator
- Is this better than a simple array?

Implementation of Vector template

- How?
- Dynamic array
  - Deep copy (copy constructor, operator=, destructor).
  - resize and reserve.
  - overload operator[] with mutator and accessor function.
  - iterator and const_iterator.
- Is this better than a simple array?
  - Vector is a First-class type.
  - Can be copied.
  - Memory it uses can be automatically reclaimed.

Optional Homework Exercise

- Go over vector class code Fig. 3.7 and 3.8 and accompanying notes
  - Store size, capacity, primitive arrya s data members
  - Copy constructor and used by operator=
  - swapping in a copy done using move operator
  - Implementation using a copy constructor and swap while simple certainly not the most efficient method esp when vectors of same size – better to simply copy each element using operator=
  - limited error checking
  - Resize and reserve routines - expanding capacity to twice as large
  - Host of other routines like empty, size, push_back..

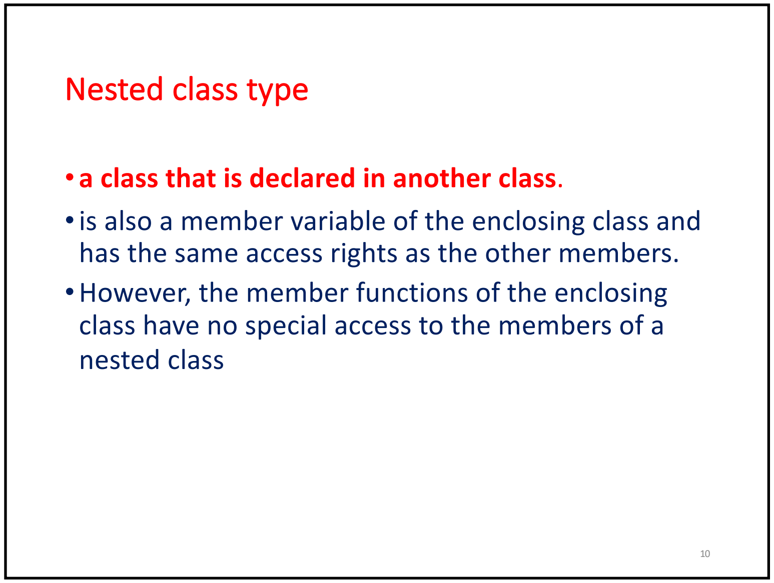


Limitations of vector type

- No error checks
- If iterator itr goes beyond the end marker, no error signal by ++itr or itr++.
- Fix:
- Require that iterator and const_iterator be actual nested class types rather than simply pointed variables.

9

9

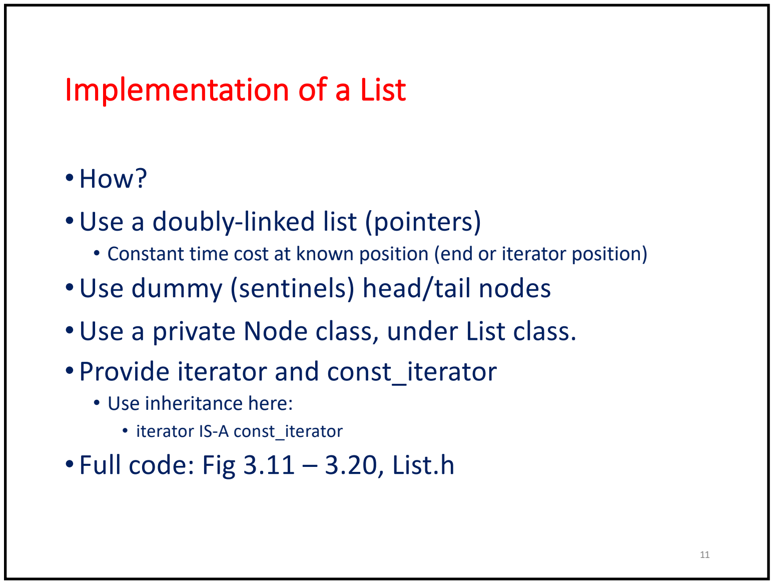


Nested class type

- **a class that is declared in another class.**
- is also a member variable of the enclosing class and has the same access rights as the other members.
- However, the member functions of the enclosing class have no special access to the members of a nested class

10

10

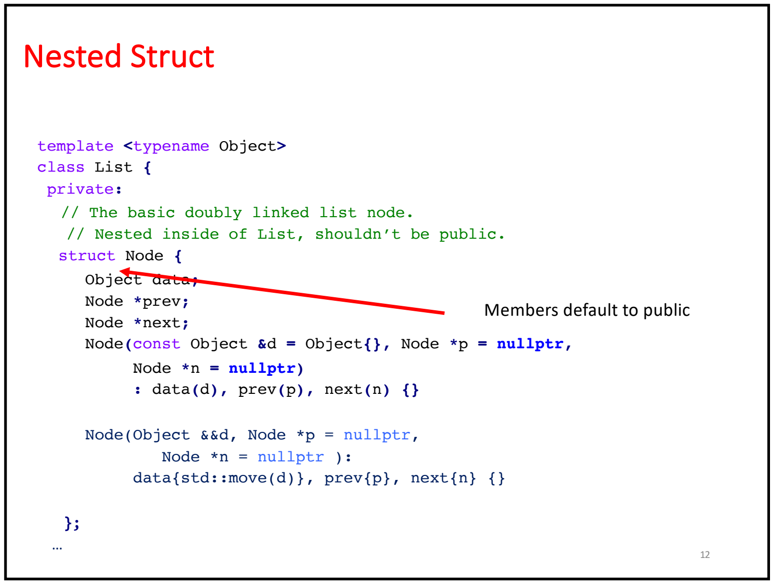


Implementation of a List

- How?
- Use a doubly-linked list (pointers)
  - Constant time cost at known position (end or iterator position)
- Use dummy (sentinels) head/tail nodes
- Use a private Node class, under List class.
- Provide iterator and const_iterator
  - Use inheritance here:
    - iterator IS-A const_iterator
- Full code: Fig 3.11 – 3.20, List.h

11

11



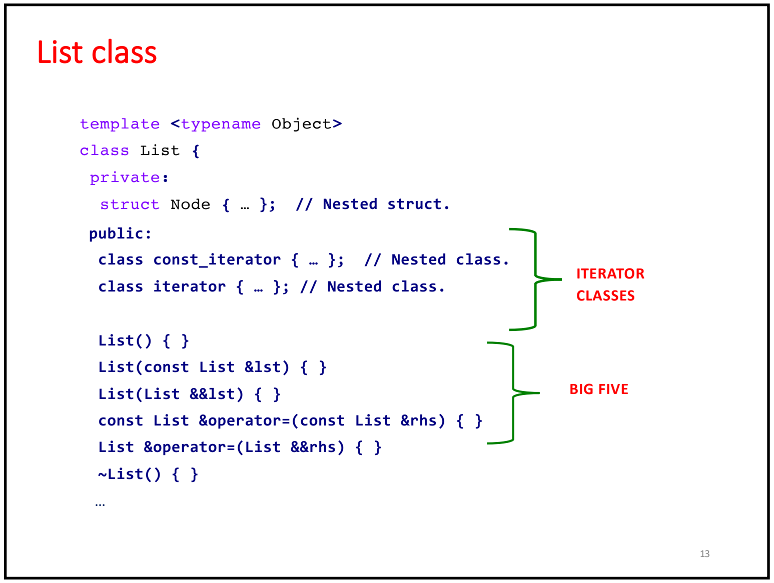
Nested Struct

```
template <typename Object>
class List {
 private:
   // The basic doubly linked list node.
    // Nested inside of List, shouldn’t be public.
  struct Node {
     Object data;
     Node *prev;
     Node *next;
     Node(const Object &d = Object{}, Node *p = nullptr,
          Node *n = nullptr)
          : data(d), prev(p), next(n) {}

     Node(Object &&d, Node *p = nullptr,
             Node *n = nullptr ):
          data{std::move(d)}, prev{p}, next{n} {}

   };
 …
```

Members default to public

12

12



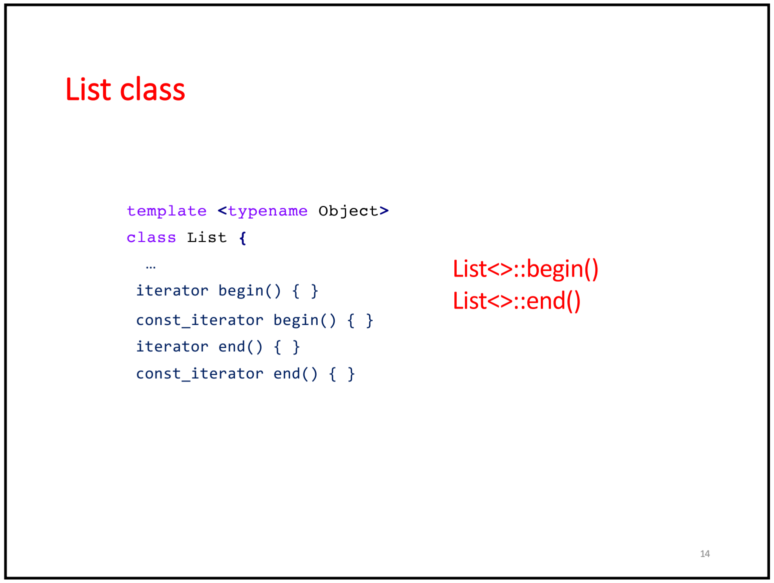
List class

```
template <typename Object>
class List {
 private:
  struct Node { … };  // Nested struct.
 public:
  class const_iterator { … };  // Nested class.
  class iterator { … }; // Nested class.

  List() { }
  List(const List &lst) { }
  List(List &&lst) { }
  const List &operator=(const List &rhs) { }
  List &operator=(List &&rhs) { }
  ~List() { }
  …
```

ITERATOR CLASSES

BIG FIVE

13

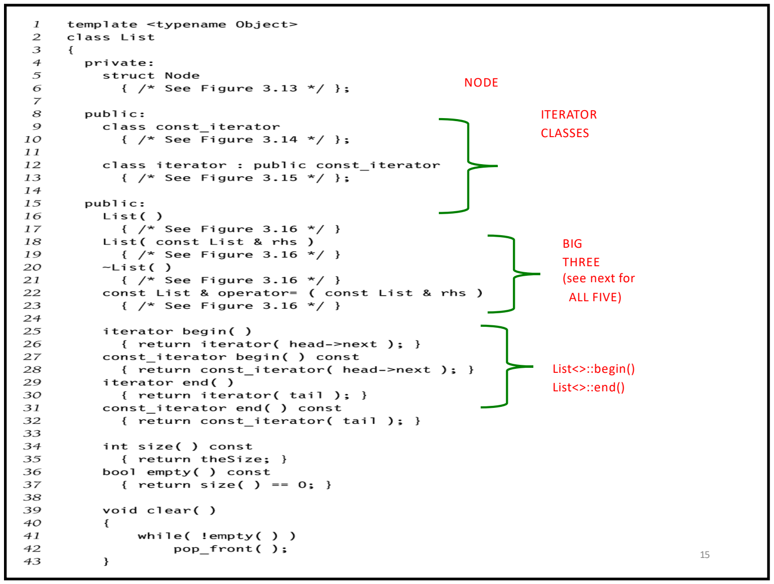


List class

```
template <typename Object>
class List {
  …
 iterator begin() { }
 const_iterator begin() { }
 iterator end() { }
 const_iterator end() { }
```

List<>::begin()
List<>::end()

14



```
template <typename Object>
class List
{
  private:
    struct Node
      { /* See Figure 3.13 */ };

  public:
    class const_iterator
      { /* See Figure 3.14 */ };

    class iterator : public const_iterator
      { /* See Figure 3.15 */ };

  public:
    List( )
      { /* See Figure 3.16 */ }
    List( const List & rhs )
      { /* See Figure 3.16 */ }
    ~List( )
      { /* See Figure 3.16 */ }
    const List & operator= ( const List & rhs )
      { /* See Figure 3.16 */ }

    iterator begin( )
      { return iterator( head->next ); }
    const_iterator begin( ) const
      { return const_iterator( head->next ); }
    iterator end( )
      { return iterator( tail ); }
    const_iterator end( ) const
      { return const_iterator( tail ); }

    int size( ) const
      { return theSize; }
    bool empty( ) const
      { return size( ) == 0; }

    void clear( )
    {
        while( !empty( ) )
            pop_front( );
    }
```

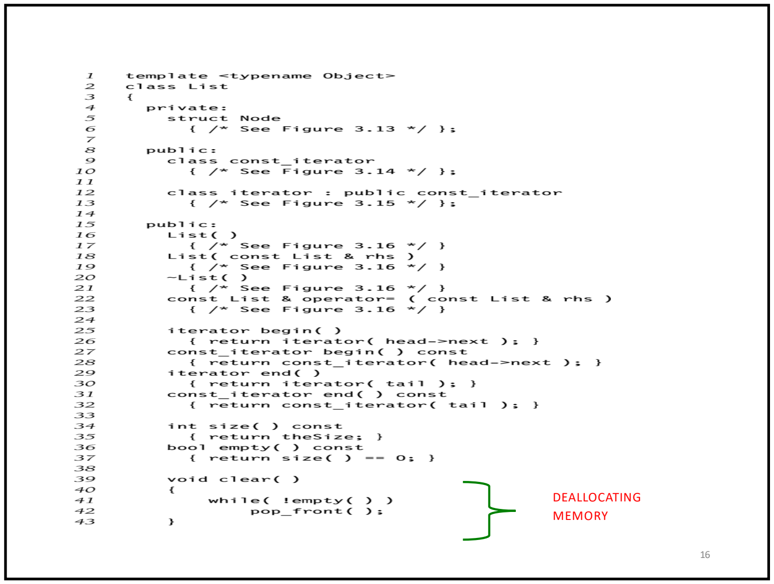
NODE

ITERATOR
CLASSES

BIG
THREE
(see next for
ALL FIVE)

List<>::begin()
List<>::end()

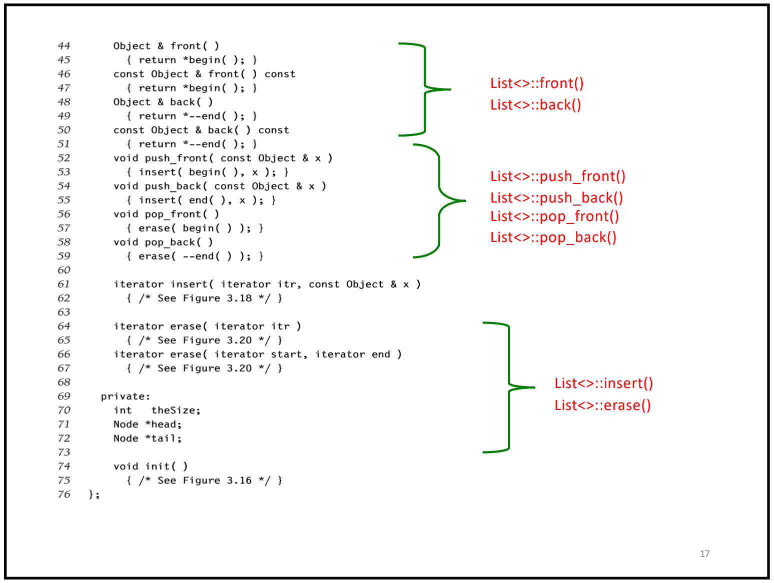
15



```
template <typename Object>
class List
{
  private:
    struct Node
      { /* See Figure 3.13 */ };

  public:
    class const_iterator
      { /* See Figure 3.14 */ };

    class iterator : public const_iterator
      { /* See Figure 3.15 */ };

  public:
    List( )
      { /* See Figure 3.16 */ }
    List( const List & rhs )
      { /* See Figure 3.16 */ }
    ~List( )
      { /* See Figure 3.16 */ }
    const List & operator= ( const List & rhs )
      { /* See Figure 3.16 */ }

    iterator begin( )
      { return iterator( head->next ); }
    const_iterator begin( ) const
      { return const_iterator( head->next ); }
    iterator end( )
      { return iterator( tail ); }
    const_iterator end( ) const
      { return const_iterator( tail ); }

    int size( ) const
      { return theSize; }
    bool empty( ) const
      { return size( ) == 0; }

    void clear( )
    {
        while( !empty( ) )
            pop_front( );
    }
```

DEALLOCATING
MEMORY

16



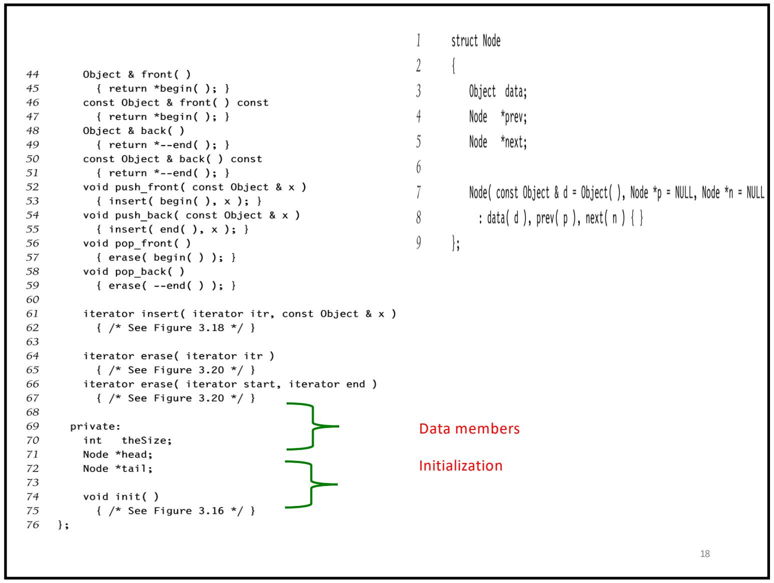
```
    Object & front( )
      { return *begin( ); }
    const Object & front( ) const
      { return *begin( ); }
    Object & back( )
      { return *--end( ); }
    const Object & back( ) const
      { return *--end( ); }
    void push_front( const Object & x )
      { insert( begin( ), x ); }
    void push_back( const Object & x )
      { insert( end( ), x ); }
    void pop_front( )
      { erase( begin( ) ); }
    void pop_back( )
      { erase( --end( ) ); }

    iterator insert( iterator itr, const Object & x )
      { /* See Figure 3.18 */ }

    iterator erase( iterator itr )
      { /* See Figure 3.20 */ }
    iterator erase( iterator start, iterator end )
      { /* See Figure 3.20 */ }

  private:
    int   theSize;
    Node *head;
    Node *tail;

    void init( )
      { /* See Figure 3.16 */ }
};
```

List<>::front()
List<>::back()

List<>::push_front()
List<>::push_back()
List<>::pop_front()
List<>::pop_back()

List<>::insert()
List<>::erase()

17



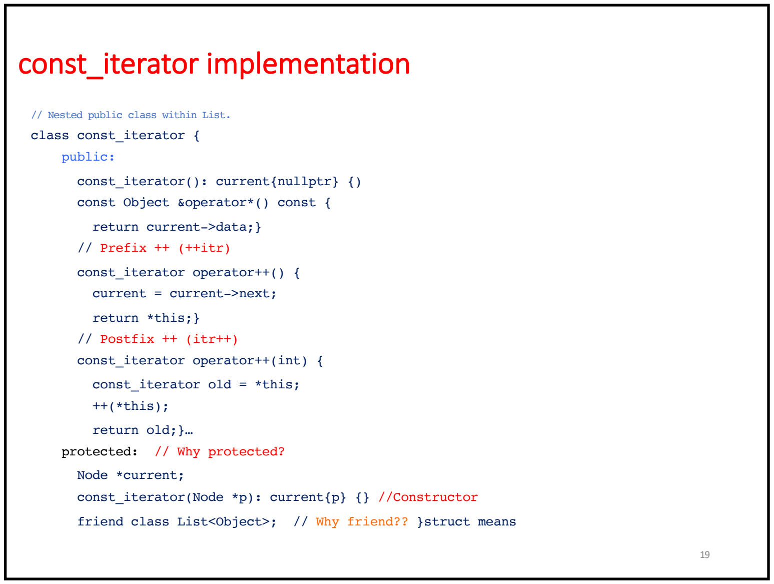
```
struct Node
{
    Object  data;
    Node   *prev;
    Node   *next;

    Node( const Object & d = Object( ), Node *p = NULL, Node *n = NULL
      : data( d ), prev( p ), next( n ) { }
};
```

```
    Object & front( )
      { return *begin( ); }
    const Object & front( ) const
      { return *begin( ); }
    Object & back( )
      { return *--end( ); }
    const Object & back( ) const
      { return *--end( ); }
    void push_front( const Object & x )
      { insert( begin( ), x ); }
    void push_back( const Object & x )
      { insert( end( ), x ); }
    void pop_front( )
      { erase( begin( ) ); }
    void pop_back( )
      { erase( --end( ) ); }

    iterator insert( iterator itr, const Object & x )
      { /* See Figure 3.18 */ }

    iterator erase( iterator itr )
      { /* See Figure 3.20 */ }
    iterator erase( iterator start, iterator end )
      { /* See Figure 3.20 */ }

  private:
    int   theSize;
    Node *head;
    Node *tail;

    void init( )
      { /* See Figure 3.16 */ }
};
```

Data members

Initialization

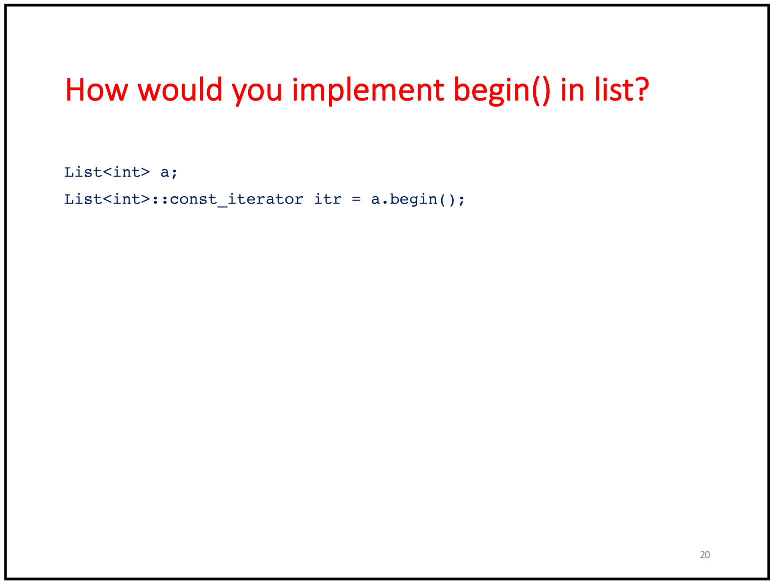
18



const_iterator implementation

```
// Nested public class within List.
class const_iterator {
    public:
      const_iterator(): current{nullptr} ()
      const Object &operator*() const {
        return current->data;}
      // Prefix ++ (++itr)
      const_iterator operator++() {
        current = current->next;
        return *this;}
      // Postfix ++ (itr++)
      const_iterator operator++(int) {
        const_iterator old = *this;
        ++(*this);
        return old;}…
    protected:  // Why protected?
      Node *current;
      const_iterator(Node *p): current{p} {} //Constructor
      friend class List<Object>;  // Why friend?? }struct means
```

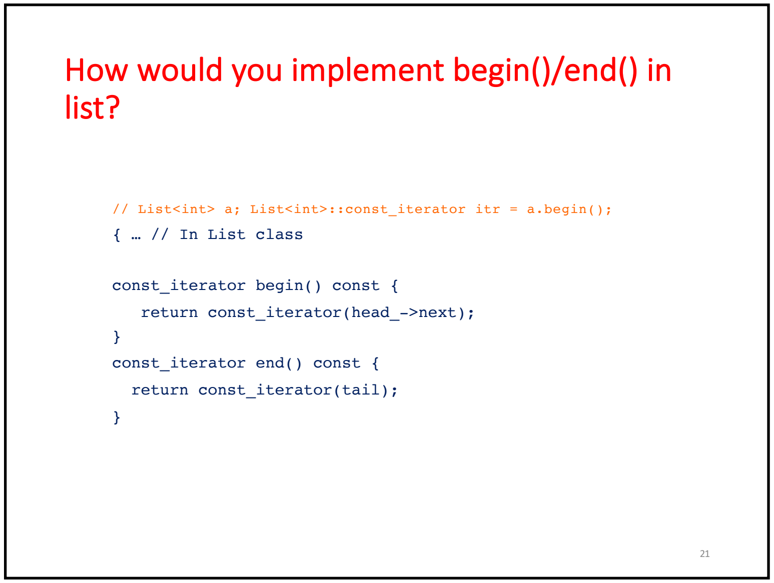
19



How would you implement begin() in list?

```
List<int> a;
List<int>::const_iterator itr = a.begin();
```

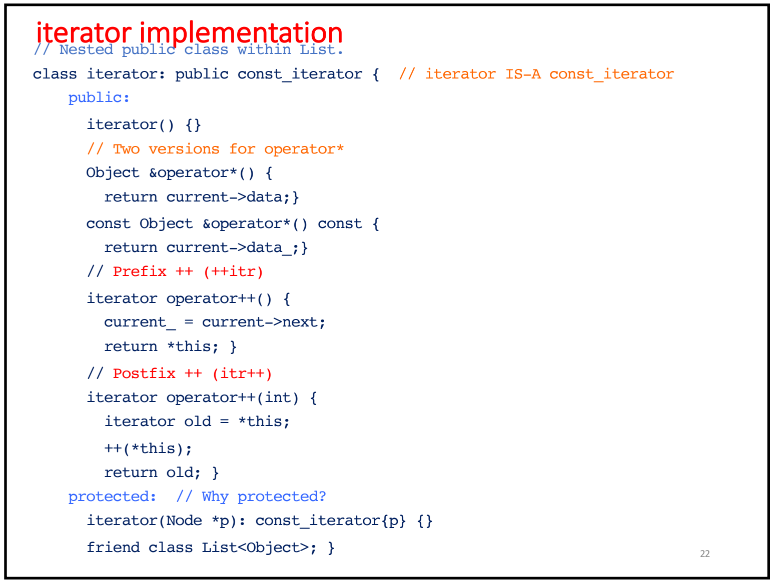
20



How would you implement begin()/end() in list?

```
// List<int> a; List<int>::const_iterator itr = a.begin();
{ … // In List class

const_iterator begin() const {
   return const_iterator(head_->next);
}
const_iterator end() const {
  return const_iterator(tail);
}
```

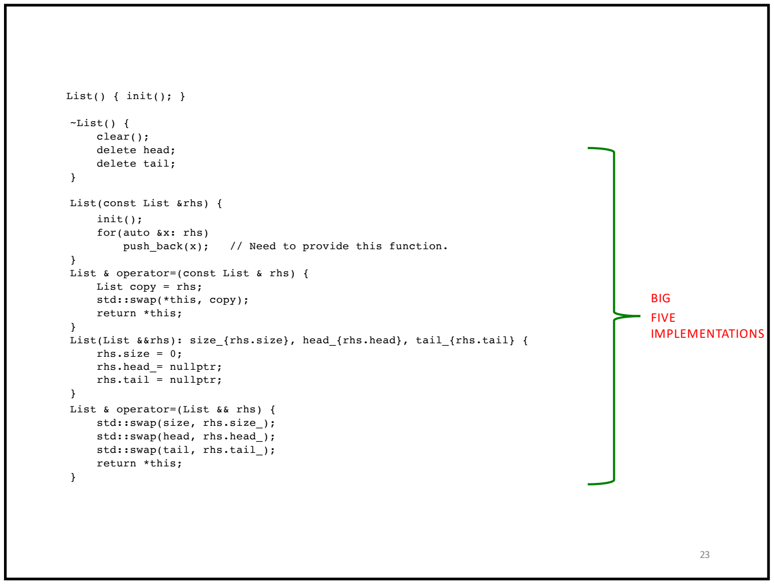
21



iterator implementation

```
// Nested public class within List.
class iterator: public const_iterator {  // iterator IS-A const_iterator
    public:
      iterator() {}
      // Two versions for operator*
      Object &operator*() {
        return current->data;}
      const Object &operator*() const {
        return current->data_;}
      // Prefix ++ (++itr)
      iterator operator++() {
        current_ = current->next;
        return *this; }
      // Postfix ++ (itr++)
      iterator operator++(int) {
        iterator old = *this;
        ++(*this);
        return old; }
    protected:  // Why protected?
      iterator(Node *p): const_iterator{p} {}
      friend class List<Object>; }
```

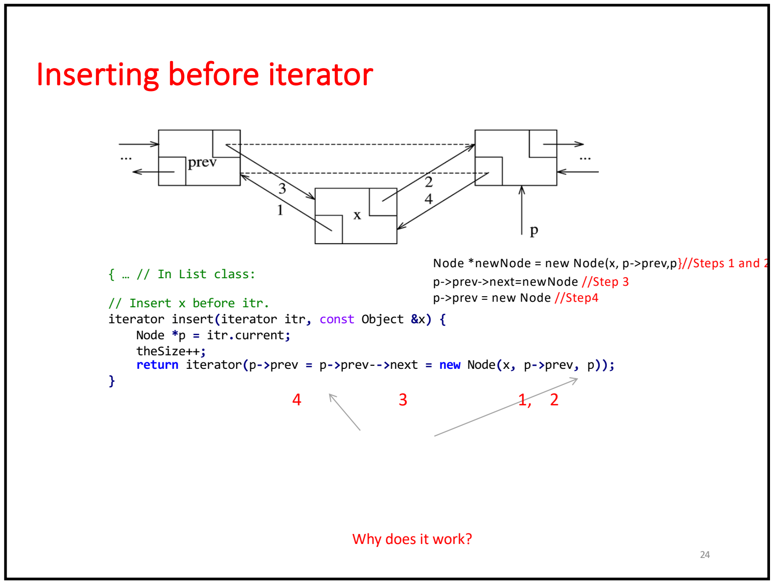
22



```
List() { init(); }

 ~List() {
     clear();
     delete head;
     delete tail;
 }

 List(const List &rhs) {
     init();
     for(auto &x: rhs)
         push_back(x);   // Need to provide this function.
 }
 List & operator=(const List & rhs) {
     List copy = rhs;
     std::swap(*this, copy);
     return *this;
 }
 List(List &&rhs): size_{rhs.size}, head_{rhs.head}, tail_{rhs.tail} {
     rhs.size = 0;
     rhs.head_= nullptr;
     rhs.tail = nullptr;
 }
 List & operator=(List && rhs) {
     std::swap(size, rhs.size_);
     std::swap(head, rhs.head_);
     std::swap(tail, rhs.tail_);
     return *this;
 }
```

BIG
FIVE
IMPLEMENTATIONS

23



Inserting before iterator

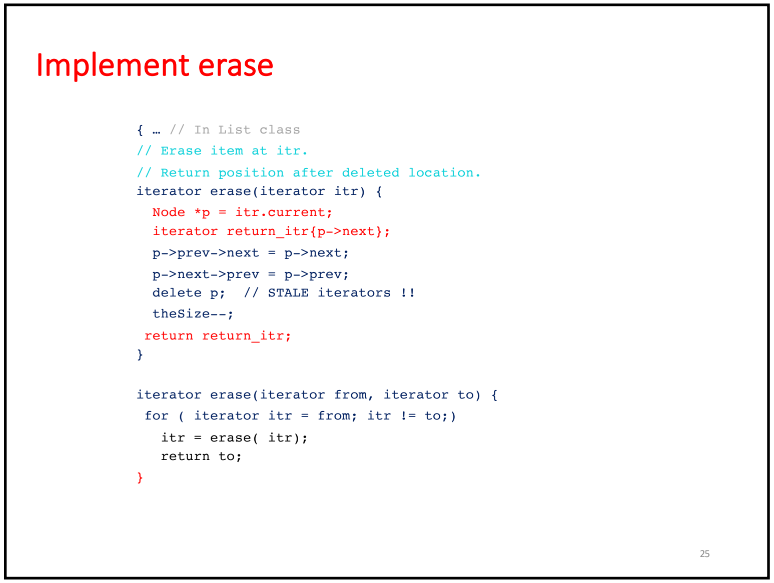
Node *newNode = new Node(x, p->prev,p}//Steps 1 and 2
p->prev->next=newNode //Step 3
p->prev = new Node //Step4

```
{ … // In List class:

// Insert x before itr.
iterator insert(iterator itr, const Object &x) {
    Node *p = itr.current;
    theSize++;
    return iterator(p->prev = p->prev-->next = new Node(x, p->prev, p));
}
```

4   3   1, 2

Why does it work?

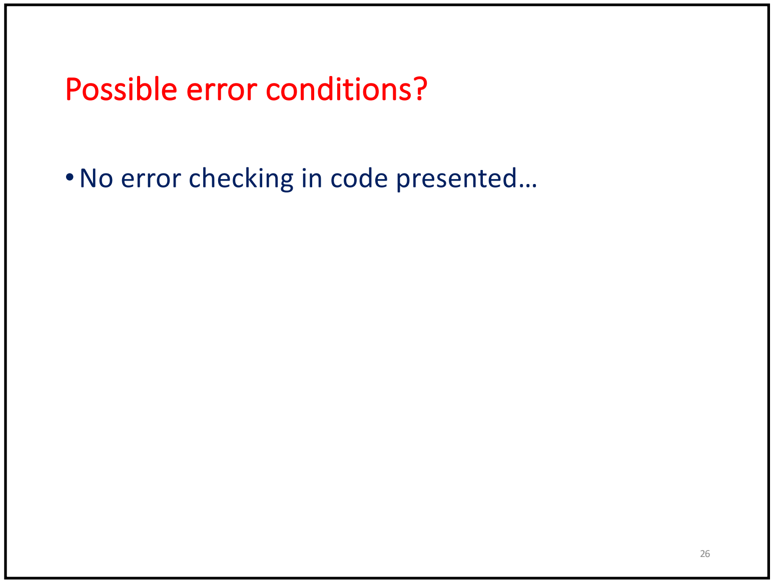
24



Implement erase

```
{ … // In List class
// Erase item at itr.
// Return position after deleted location.
iterator erase(iterator itr) {
  Node *p = itr.current;
  iterator return_itr{p->next};
  p->prev->next = p->next;
  p->next->prev = p->prev;
  delete p;  // STALE iterators !!
  theSize--;
 return return_itr;
}

iterator erase(iterator from, iterator to) {
 for ( iterator itr = from; itr != to;)
   itr = erase( itr);
   return to;
}
```

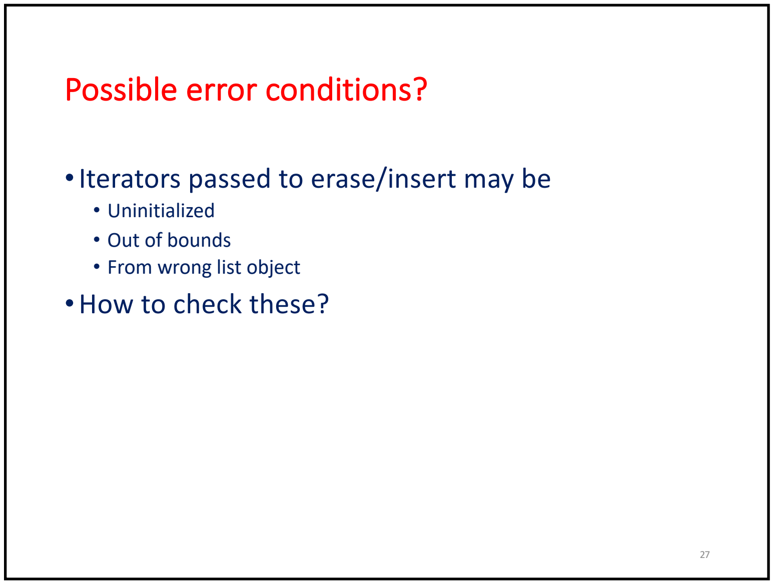
25



Possible error conditions?

- No error checking in code presented…

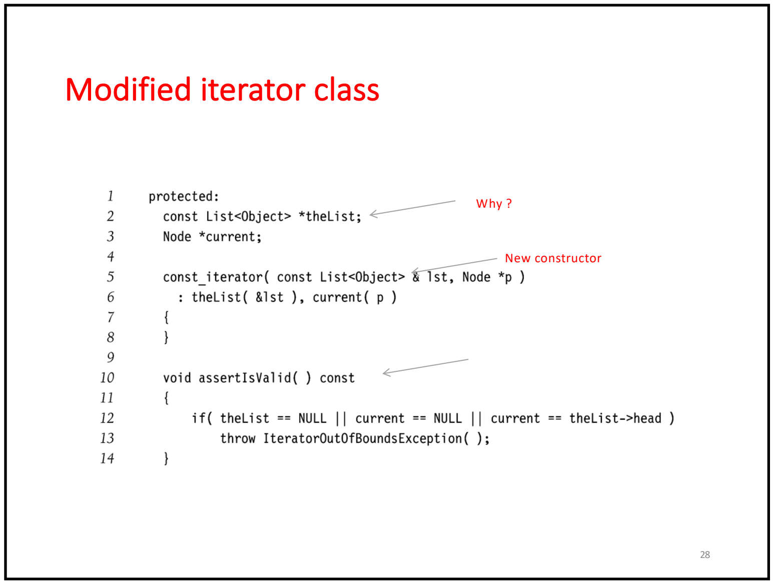
26



Possible error conditions?

- Iterators passed to erase/insert may be
  - Uninitialized
  - Out of bounds
  - From wrong list object
- How to check these?

27

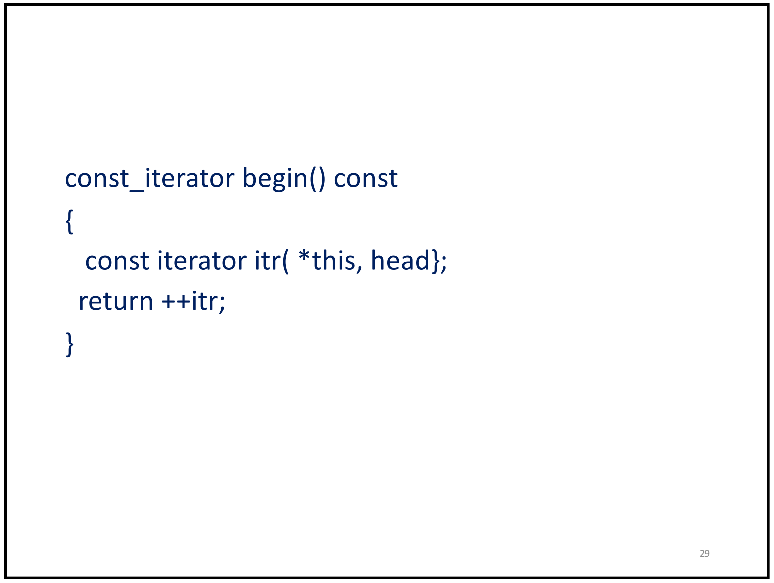


Modified iterator class

```
protected:
  const List<Object> *theList;
  Node *current;

  const_iterator( const List<Object> & lst, Node *p )
    : theList( &lst ), current( p )
  {
  }

  void assertIsValid( ) const
  {
      if( theList == NULL || current == NULL || current == theList->head )
          throw IteratorOutOfBoundsException( );
  }
```

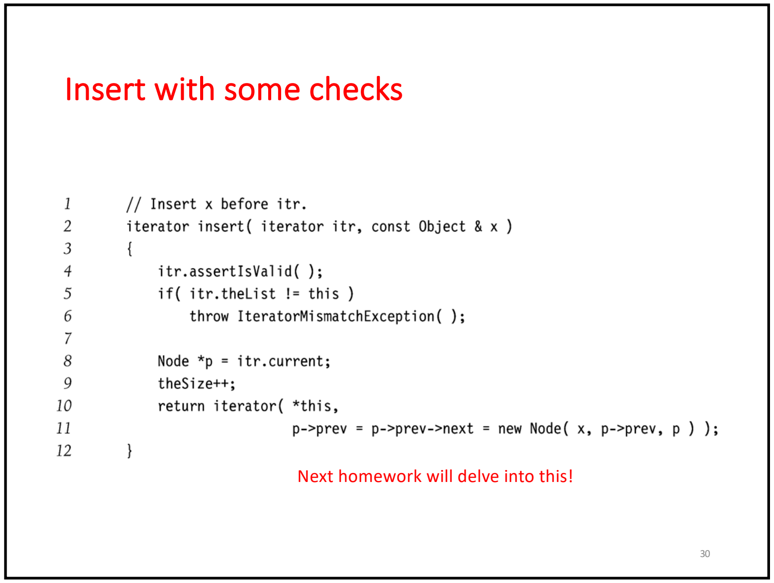
Why ?
New constructor

28



```
const_iterator begin() const
{
   const iterator itr( *this, head};
  return ++itr;
}
```

29

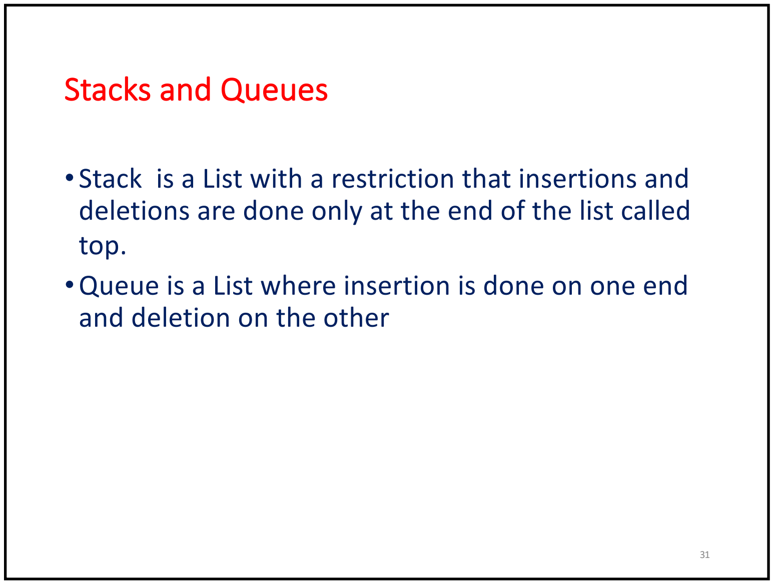


Insert with some checks

```
// Insert x before itr.
iterator insert( iterator itr, const Object & x )
{
    itr.assertIsValid( );
    if( itr.theList != this )
        throw IteratorMismatchException( );

    Node *p = itr.current;
    theSize++;
    return iterator( *this,
                     p->prev = p->prev->next = new Node( x, p->prev, p ) );
}
```

Next homework will delve into this!

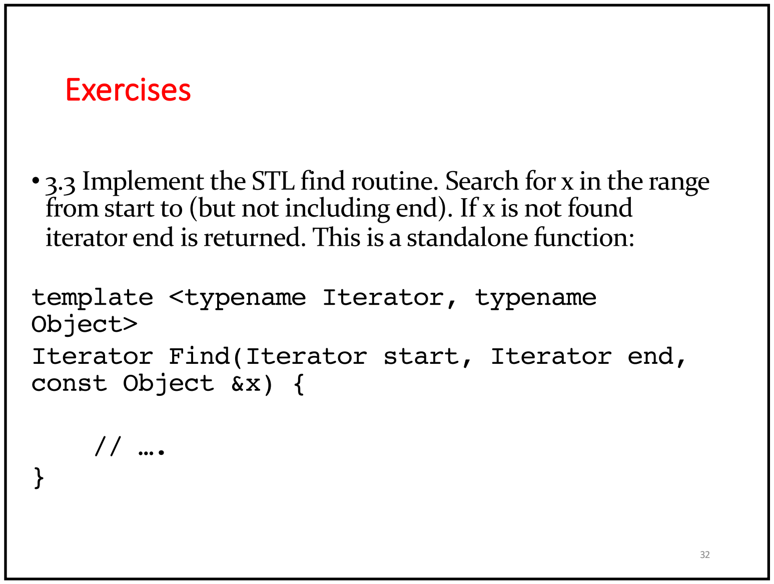
30



Stacks and Queues

- Stack is a List with a restriction that insertions and deletions are done only at the end of the list called top.
- Queue is a List where insertion is done on one end and deletion on the other

31

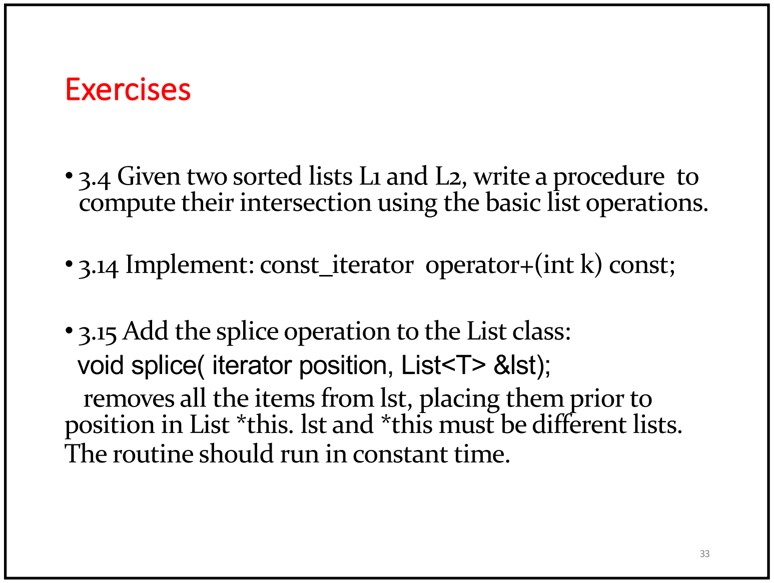


Exercises

- 3.3 Implement the STL find routine. Search for x in the range from start to (but not including end). If x is not found iterator end is returned. This is a standalone function:

```
template <typename Iterator, typename
Object>
Iterator Find(Iterator start, Iterator end,
const Object &x) {

    // ….
}
```

32

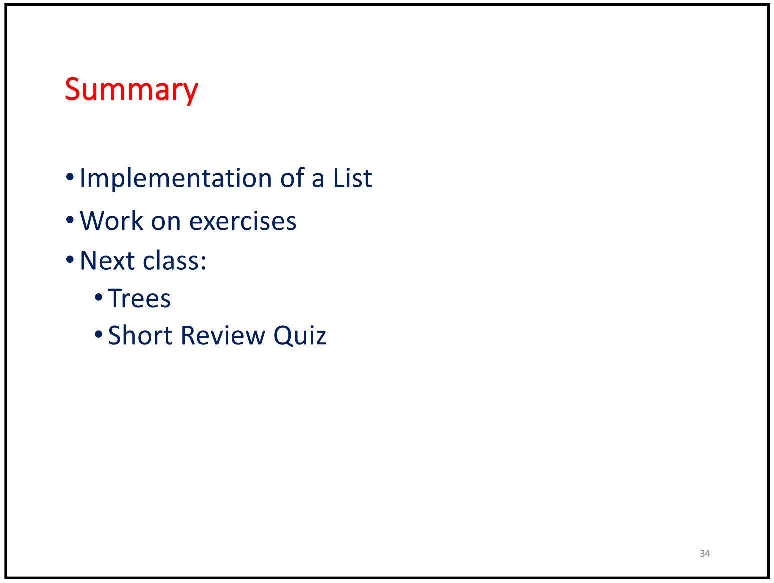


Exercises

- 3.4 Given two sorted lists L1 and L2, write a procedure to compute their intersection using the basic list operations.
- 3.14 Implement: const_iterator operator+(int k) const;
- 3.15 Add the splice operation to the List class:
  void splice(iterator position, List<T> &lst);
  removes all the items from lst, placing them prior to position in List *this. lst and *this must be different lists. The routine should run in constant time.

33

33



Summary

- Implementation of a List
- Work on exercises
- Next class:
  - Trees
  - Short Review Quiz

34

34