Abstract Data Types

Topics

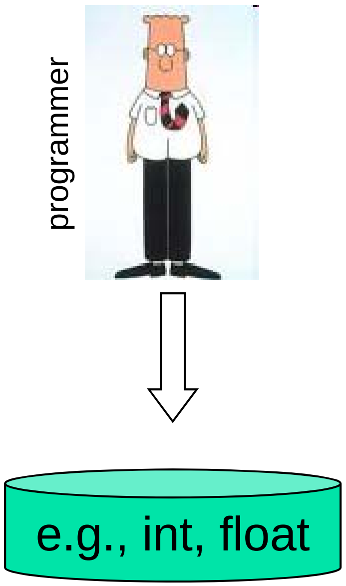
- Abstract Data Types (ADTs)

Some basic ADTs:

- Lists
- Stacks
- Queues

Primitive Data Type vs. Abstract Data Types

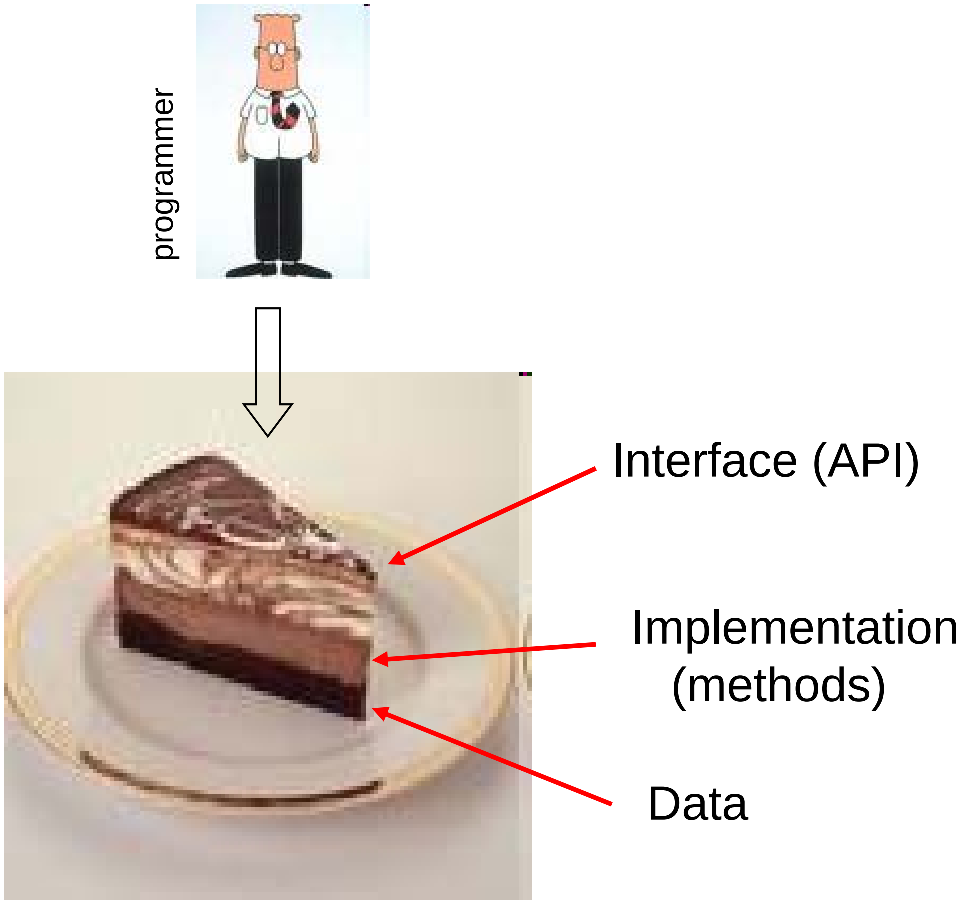
Primitive DT:



programmer

e.g., int, float

ADT:



programmer

Interface (API)

Implementation (methods)

Data

Abstract Data Types (ADTs)

- ADT is a set of objects together with a set of operations.
    - "Abstract" in that implementation of operations not specified in ADT definition
    - E.g., List
    - Insert, delete, search, sort

- C++ classes are perfect for ADTs
- Can change ADT implementation details without breaking code using ADT

Specifications of basic ADTs

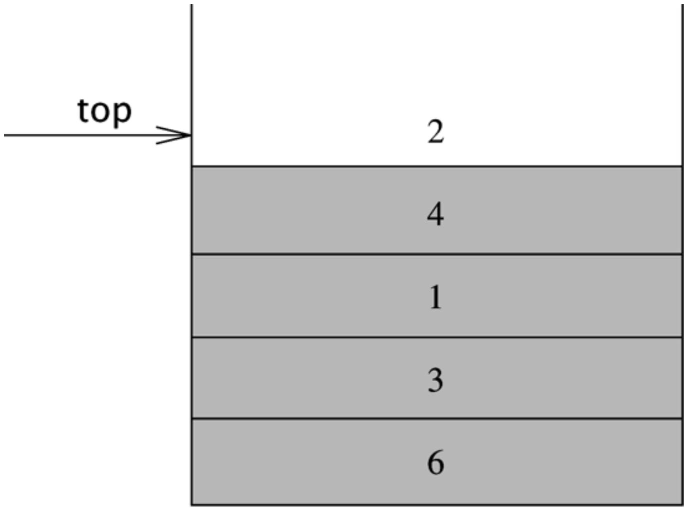
List, Stack, Queue

The List ADT

- List of size N: A_0, A_1, ..., A_{N-1}
- Each element A_k has a unique position k in the list
- Elements can be arbitrarily complex
- Operations
  - insert(X,k), remove(k), find(X), findKth(k), printList()

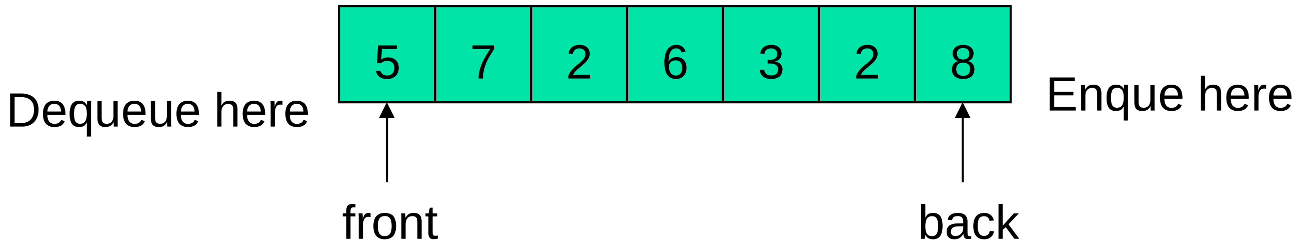
Stack ADT

- Stack = a list where insert and remove take place only at the "top"
- Operations
  - Push (insert) element on top of stack
  - Pop (remove) element from top of stack
  - Top: return element at top of stack
- LIFO (Last In First Out)



Queue ADT

- Queue = a list where insert takes place at the back, but remove takes place at the front
- Operations
  - Enqueue (insert) element at the back of the queue
  - Dequeue (remove and return) element from the front of the queue
  - FIFO (First In First Out)



Implementation for basic ADTs

List ADT using Arrays

A_0	A_1	A_2	A_3	…

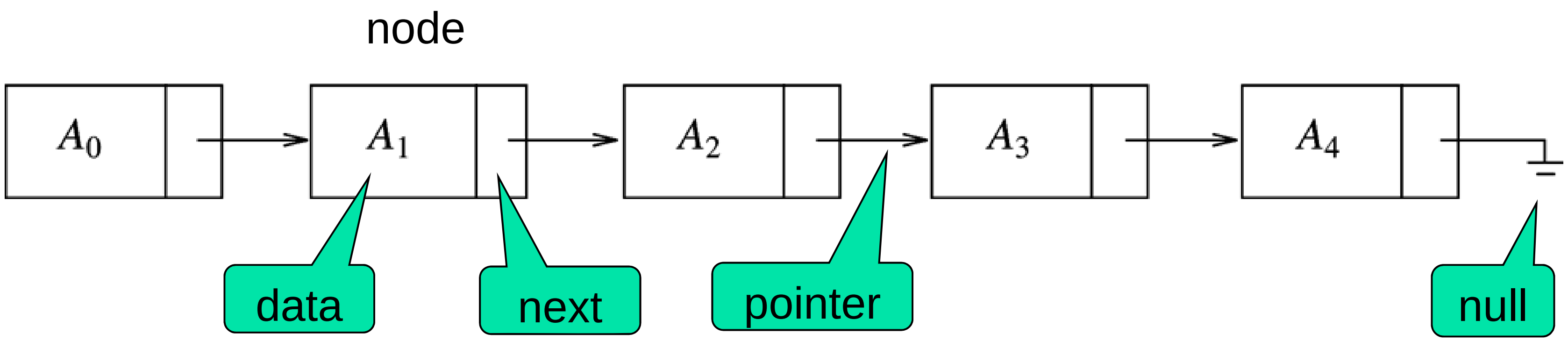
- Operations
  - insert(X,k) : O(N)
  - remove(k) : O(N)
  - find(X) : O(N)
  - findKth(k) : O(1)
  - printList() : O(N)

• Read as “order N” (means that runtime is proportional to N)

• Read as “order 1” (means that runtime is a constant – i.e., not dependent on N)

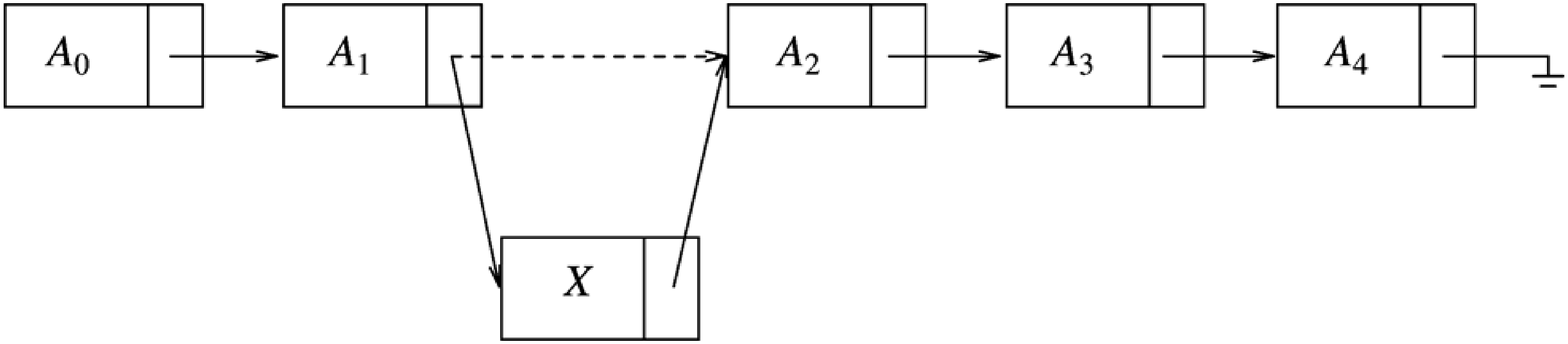
List ADT using Linked Lists

- Elements not stored in contiguous memory
- Nodes in list consist of data element and next pointer

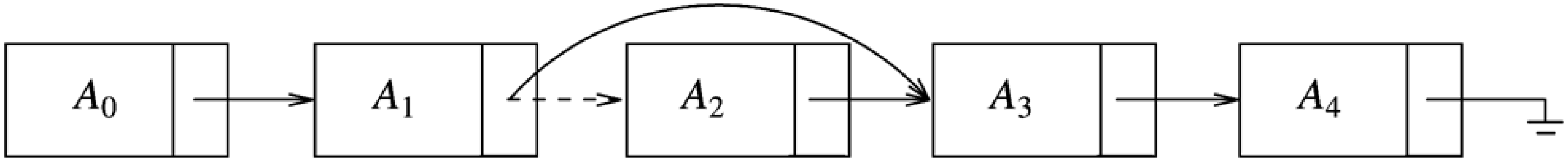


Linked Lists

- Operations
  - Insert(X,A) – O(1)



  - Remove(A) – O(1)

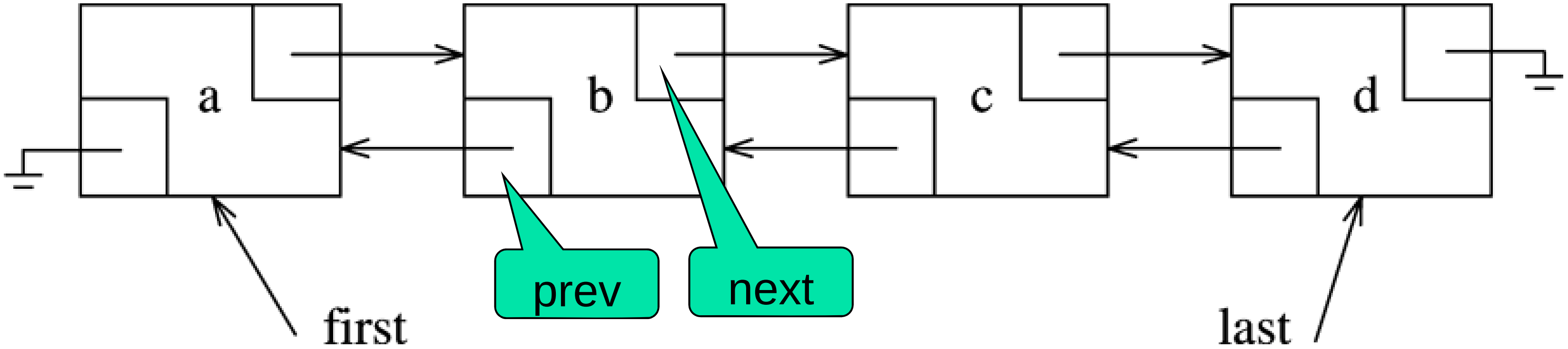


Linked Lists

- Operations
  - find(X) – O(N)
  - findKth(k) – O(N)
  - printList() – O(N)

Doubly-Linked List

- Singly-linked list
  - insert(X,A) and remove(X) require pointer to node just before X
- Doubly-linked list
  - Also keep pointer to previous node



Doubly-Linked List

- Insert(X,A)

```
newA = new Node(A);
newA->prev = X->prev;
newA->next = X;
X->prev->next = newA;
X->prev = newA;
```

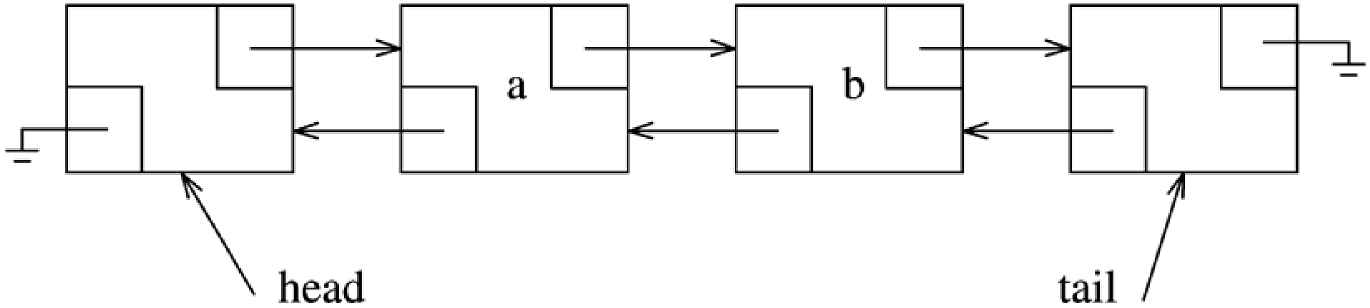
- Remove(X)

```
X->prev->next = X->next;
X->next->prev = X->prev;
```

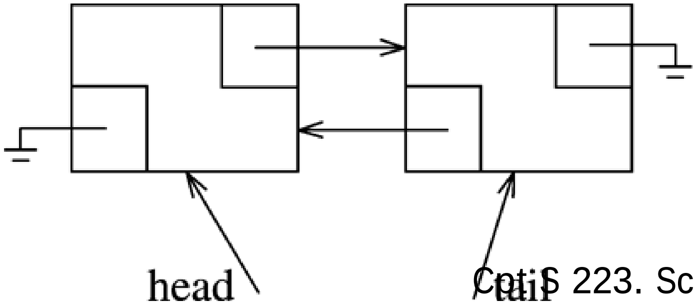
- Problems with operations at ends of list

Sentinel Nodes

- Dummy head and tail nodes to avoid special cases at ends of list
- Doubly-linked list with sentinel nodes



- Empty doubly-linked list with sentinel nodes



C++ Standard Template Library (STL)

- Implementation of common data structures
  - List, stack, queue, …
  - Generally called *containers*
- WWW references for STL
  - www.sgi.com/tech/stl/
  - http://www.cplusplus.com/reference/stl/
  - www.cppreference.com/cppstl.html

Implementing Lists using STL

- **`vector<Object>`**
  - Array-based implementation
  - **`findKth`** – O(1)
  - **`insert`** and **`remove`** – O(N)
    - Unless change at end of vector
- **`list<Object>`**
  - Doubly-linked list with sentinel nodes
  - **`findKth`** – O(N)
  - **`insert`** and **`remove`** – O(1)
    - If position of change is known
- Both require O(N) for search

Container Methods

- `int size() const`
  - Return number of elements in container
- `void clear()`
  - Remove all elements from container
- `bool empty()`
  - Return true is container has no elements, otherwise returns false

Vector and List Methods

- `void push_back (const Object & x)`
  - Add x to end of list
- `void pop_back ()`
  - Remove object at end of list
- `const Object & back () const`
  - Return object at end of list
- `const Object & front () const`
  - Return object at front of list

List-only Methods

- `void push_front (const Object & x)`
  - Add x to front of list
- `void pop_front ()`
  - Remove object at front of list

Vector-only Methods

- **`Object & operator[] (int idx)`**
  - Return object at index idx in vector with no bounds-checking
- **`Object & at (int idx)`**
  - Return object at index idx in vector with bounds-checking
- **`int capacity () const`**
  - Return internal capacity of vector
- **`void reserve (int newCapacity)`**
  - Set new capacity for vector (avoid expansion)

Iterators

- Represents position in container
- Getting an iterator
  - **`iterator begin ()`**
    - Return appropriate iterator representing first item in container
  - **`iterator end ()`**
    - Return appropriate iterator representing end marker in container
    - Position ***after*** last item in container

Iterator Methods

- **`itr++`** and **`++itr`**
  - Advance iterator **`itr`** to next location
- **`*itr`**
  - Return reference to object stored at iterator **`itr`**'s location
- **`itr1 == itr2`**
  - Return true if **`itr1`** and **`itr2`** refer to same location; otherwise return false
- **`itr1 != itr2`**
  - Return true if **`itr1`** and **`itr2`** refer to different locations; otherwise return false

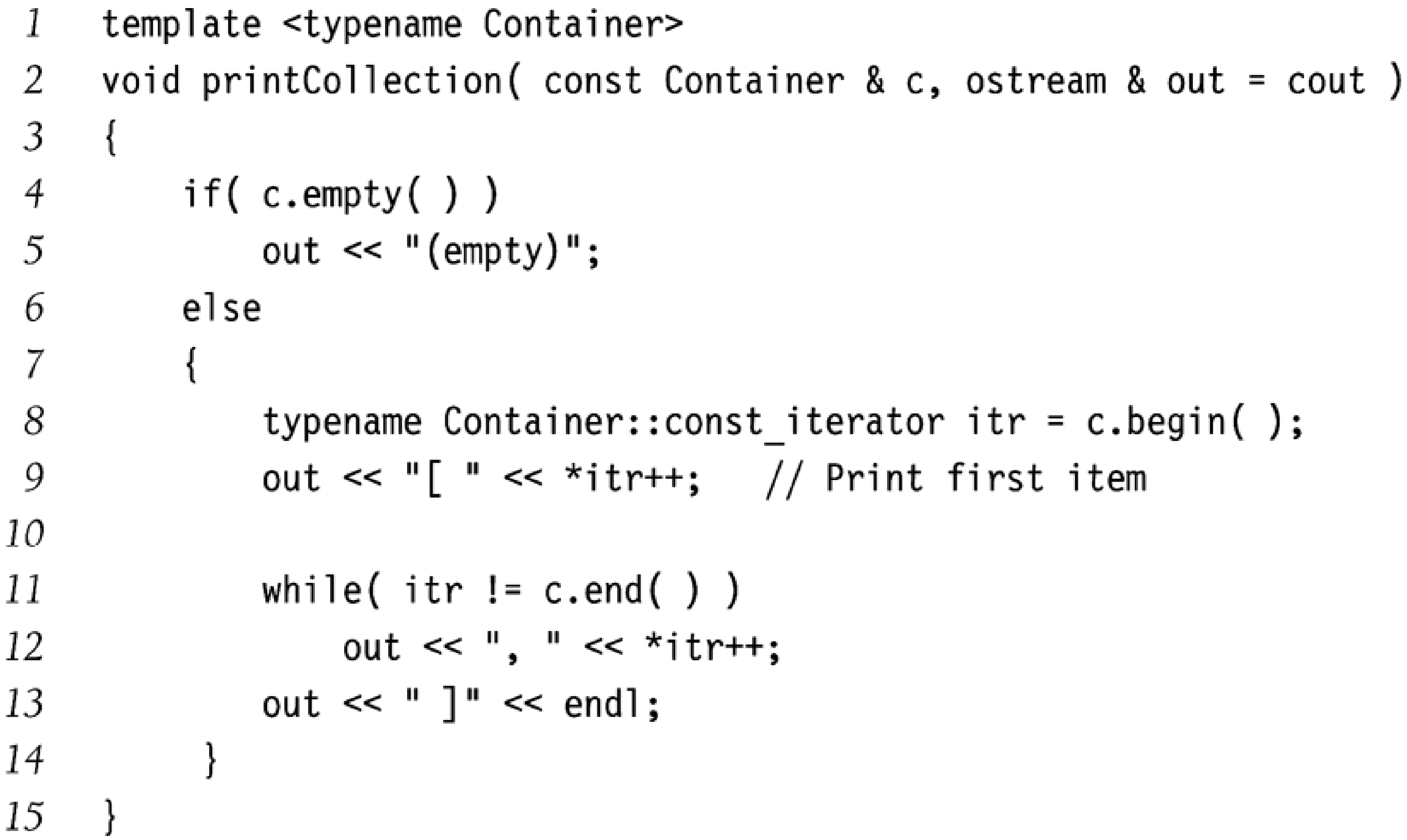
Example: printList

```
template <typename Container>
void printList (const Container & lst)
{
    for (typename Container::const_iterator itr = lst.begin();
         itr != lst.end();
         ++itr)
    {
        cout << *itr << endl;
    }
}
```

Constant Iterators

- `iterator begin ()`
- `const_iterator begin () const`
- `iterator end ()`
- `const_iterator end () const`
- Appropriate version above returned based on whether container is `const`
- If `const_iterator` used, then `*itr` cannot appear on left-hand side of assignment (e.g., `*itr=0`)

Better printList

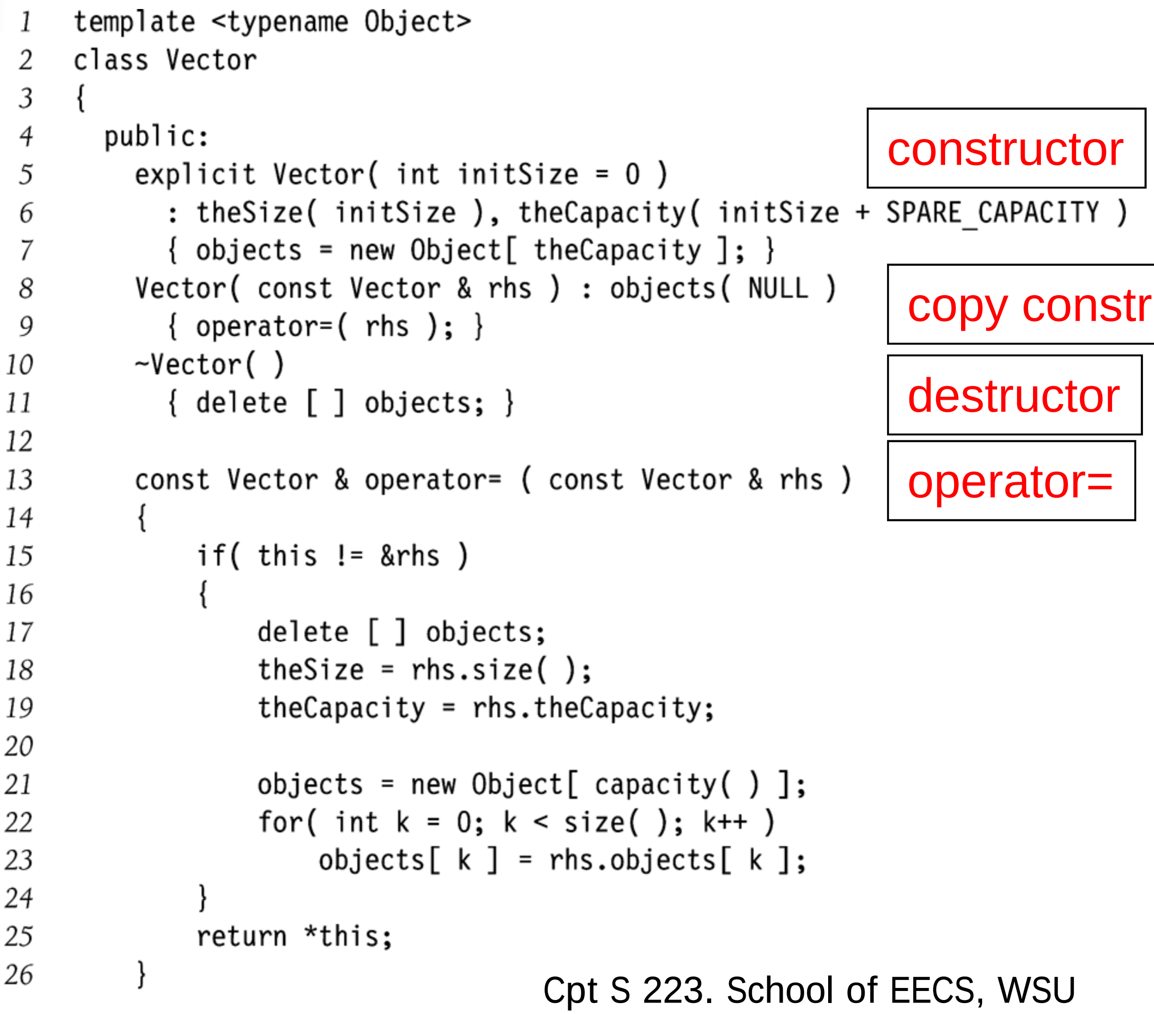


```
template <typename Container>
void printCollection( const Container & c, ostream & out = cout )
{
    if( c.empty( ) )
        out << "(empty)";
    else
    {
        typename Container::const_iterator itr = c.begin( );
        out << "[ " << *itr++;   // Print first item

        while( itr != c.end( ) )
            out << ", " << *itr++;
        out << " ]" << endl;
     }
}
```

Container Operations Requiring Iterators

- **`iterator insert (iterator pos, const Object & x)`**
  - Add **x** into list, prior to position given by iterator **pos**
  - Return iterator representing position of inserted item
  - O(1) for lists, O(N) for vectors
- **`iterator erase (iterator pos)`**
  - Remove object whose position is given by iterator **pos**
  - Return iterator representing position of item following **pos**
  - This operation invalidates **pos**
  - O(1) for lists, O(N) for vectors
- **`iterator erase (iterator start, iterator end)`**
  - Remove all items beginning at position **`start`**, up to, but not including **end**

Implementation of Vector



```
template <typename Object>
class Vector
{
  public:
    explicit Vector( int initSize = 0 )
      : theSize( initSize ), theCapacity( initSize + SPARE_CAPACITY )
      { objects = new Object[ theCapacity ]; }
    Vector( const Vector & rhs ) : objects( NULL )
      { operator=( rhs ); }
    ~Vector( )
      { delete [ ] objects; }

    const Vector & operator= ( const Vector & rhs )
    {
        if( this != &rhs )
        {
            delete [ ] objects;
            theSize = rhs.size( );
            theCapacity = rhs.theCapacity;

            objects = new Object[ capacity( ) ];
            for( int k = 0; k < size( ); k++ )
                objects[ k ] = rhs.objects[ k ];
        }
        return *this;
    }
```

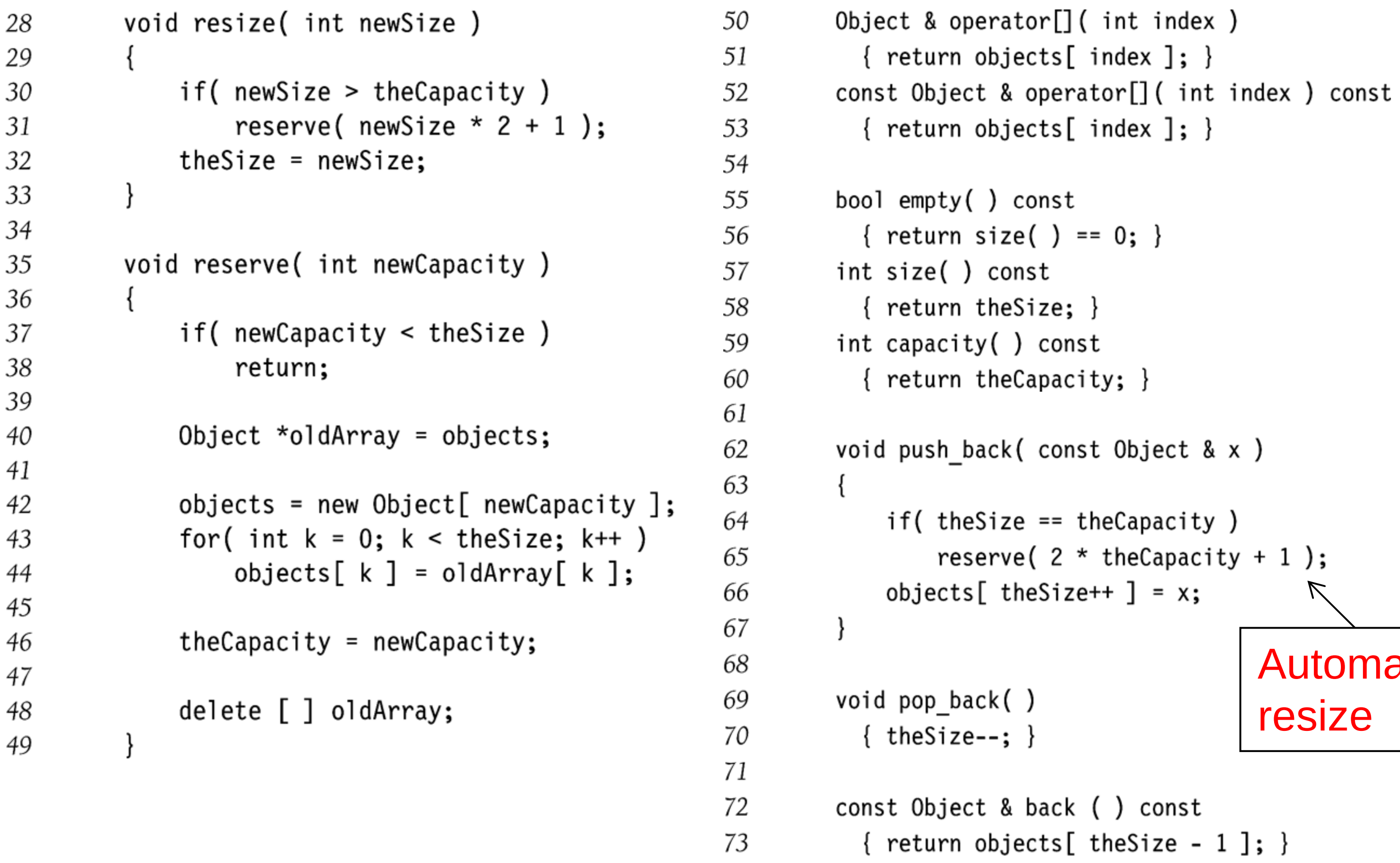
constructor

copy constructor

destructor

operator=

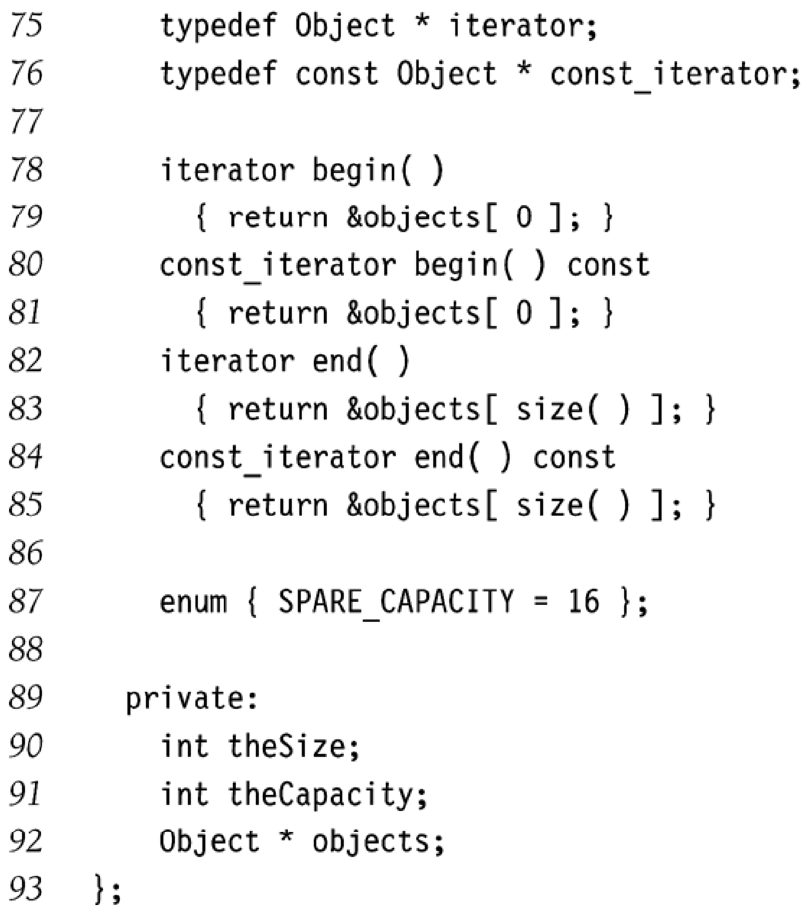
Implementation of Vector



```
void resize( int newSize )
{
    if( newSize > theCapacity )
        reserve( newSize * 2 + 1 );
    theSize = newSize;
}

void reserve( int newCapacity )
{
    if( newCapacity < theSize )
        return;

    Object *oldArray = objects;

    objects = new Object[ newCapacity ];
    for( int k = 0; k < theSize; k++ )
        objects[ k ] = oldArray[ k ];

    theCapacity = newCapacity;

    delete [ ] oldArray;
}
Object & operator[]( int index )
  { return objects[ index ]; }
const Object & operator[]( int index ) const
  { return objects[ index ]; }

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

void push_back( const Object & x )
{
    if( theSize == theCapacity )
        reserve( 2 * theCapacity + 1 );
    objects[ theSize++ ] = x;
}

void pop_back( )
  { theSize--; }

const Object & back ( ) const
  { return objects[ theSize - 1 ]; }
```

Automatic resize

Implementation of Vector

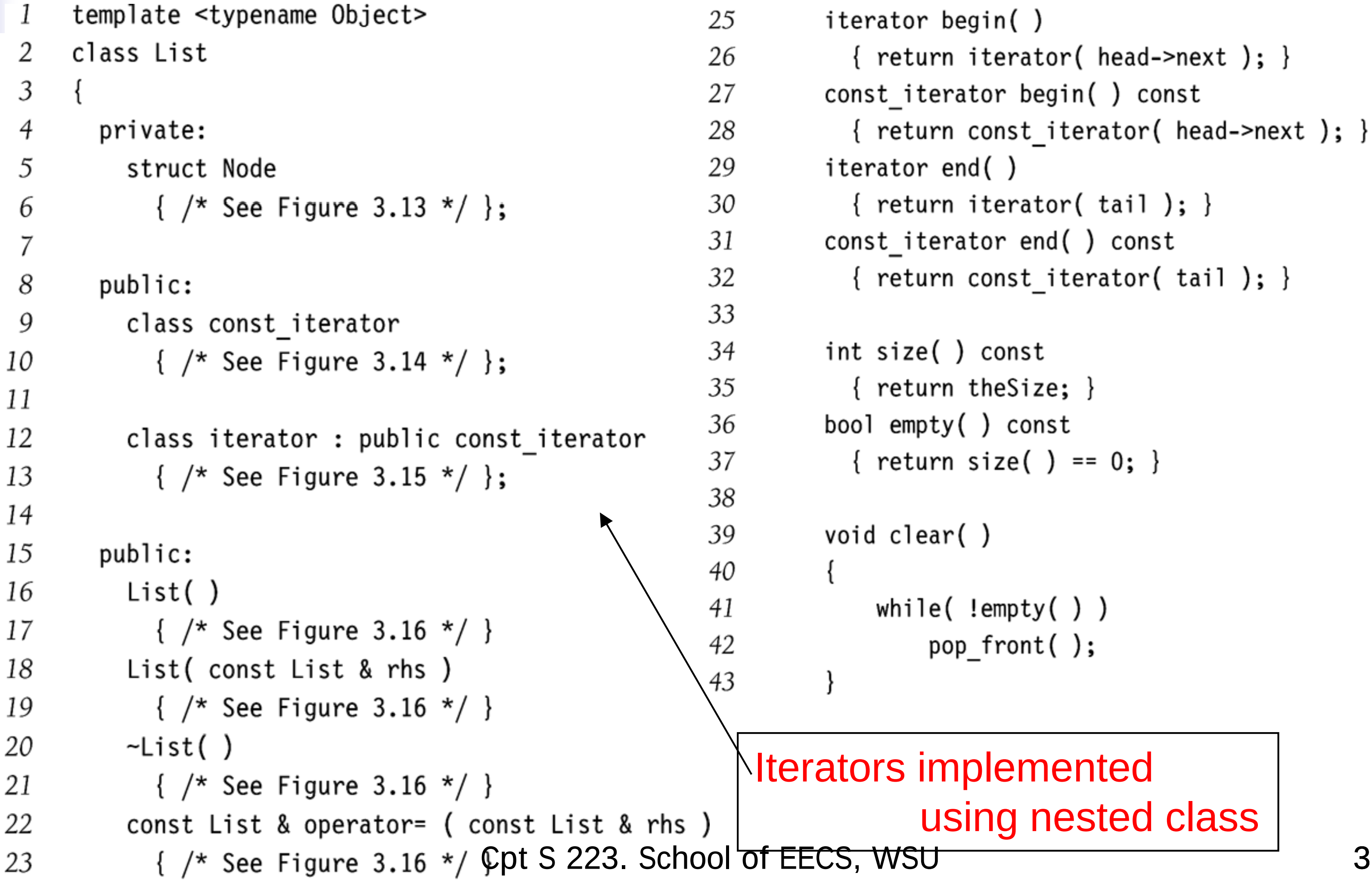


```
    typedef Object * iterator;
    typedef const Object * const_iterator;

    iterator begin( )
      { return &objects[ 0 ]; }
    const_iterator begin( ) const
      { return &objects[ 0 ]; }
    iterator end( )
      { return &objects[ size( ) ]; }
    const_iterator end( ) const
      { return &objects[ size( ) ]; }

    enum { SPARE_CAPACITY = 16 };

  private:
    int theSize;
    int theCapacity;
    Object * objects;
};
```

Iterators (implemented using simple pointers)

Iterator methods

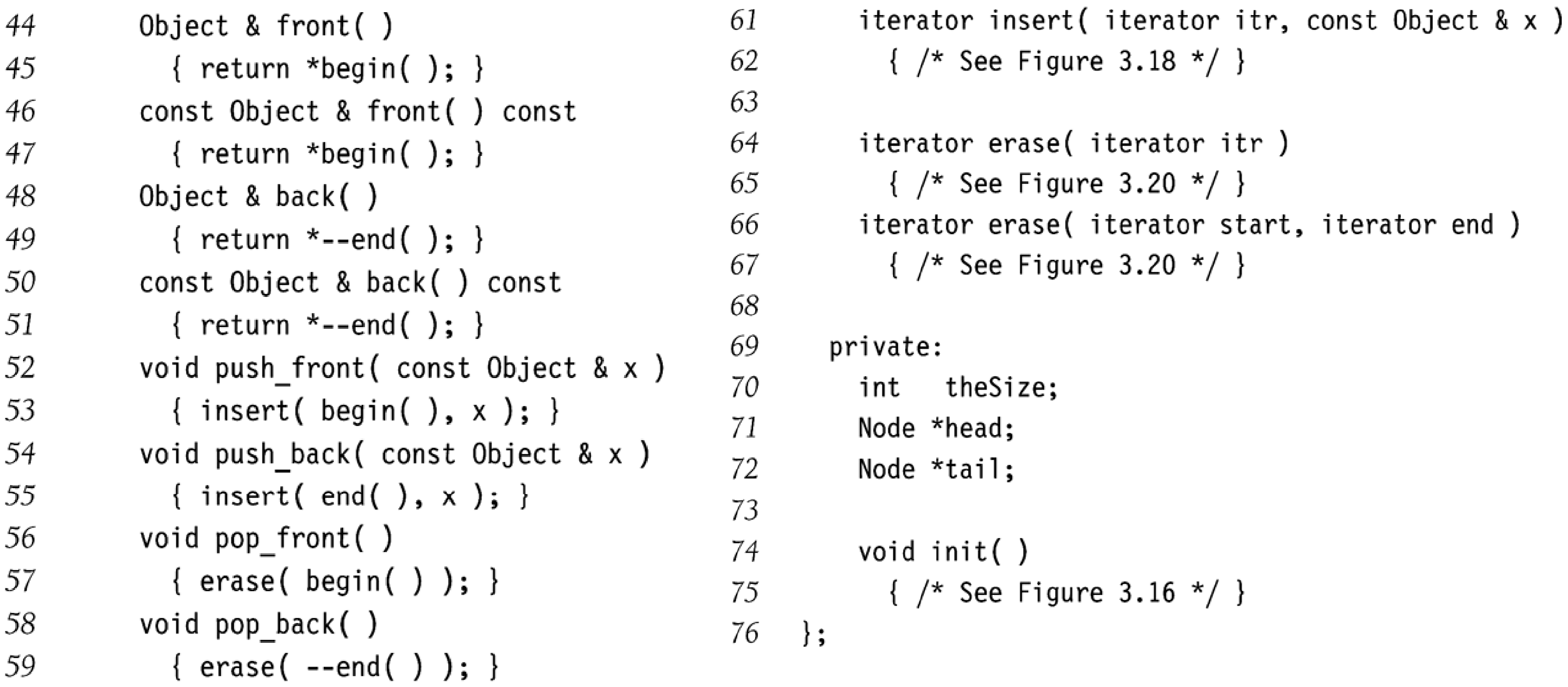
Implementation of List



```
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( );
    }
```

Iterators implemented using nested class

Implementation of List

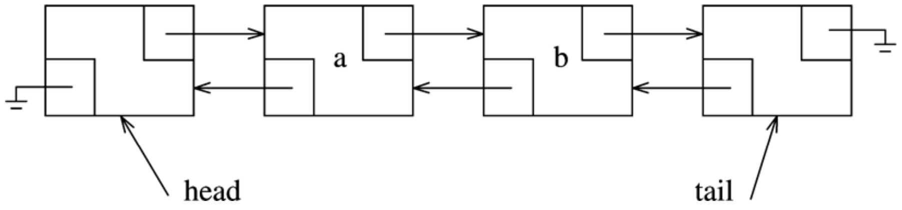


```
    Object & front( )
      { return *begin( ); }
    const Object & front( ) const
      { return *begin( ); }
    Object & back( )
      { return *--end( ); }
    const Object & back( ) const
      { return *--end( ); }
    void push_front( const Object & x )
      { insert( begin( ), x ); }
 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

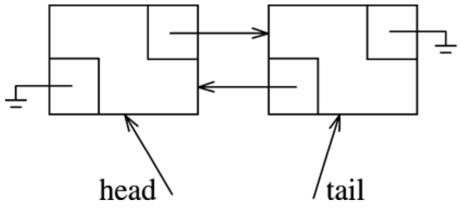
```
    iterator insert( iterator itr, const Object & x )
      { /* See Figure 3.18 */ }

    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 */ }
};
```

Implementation of List

```
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 ) { }
};
```



Empty list



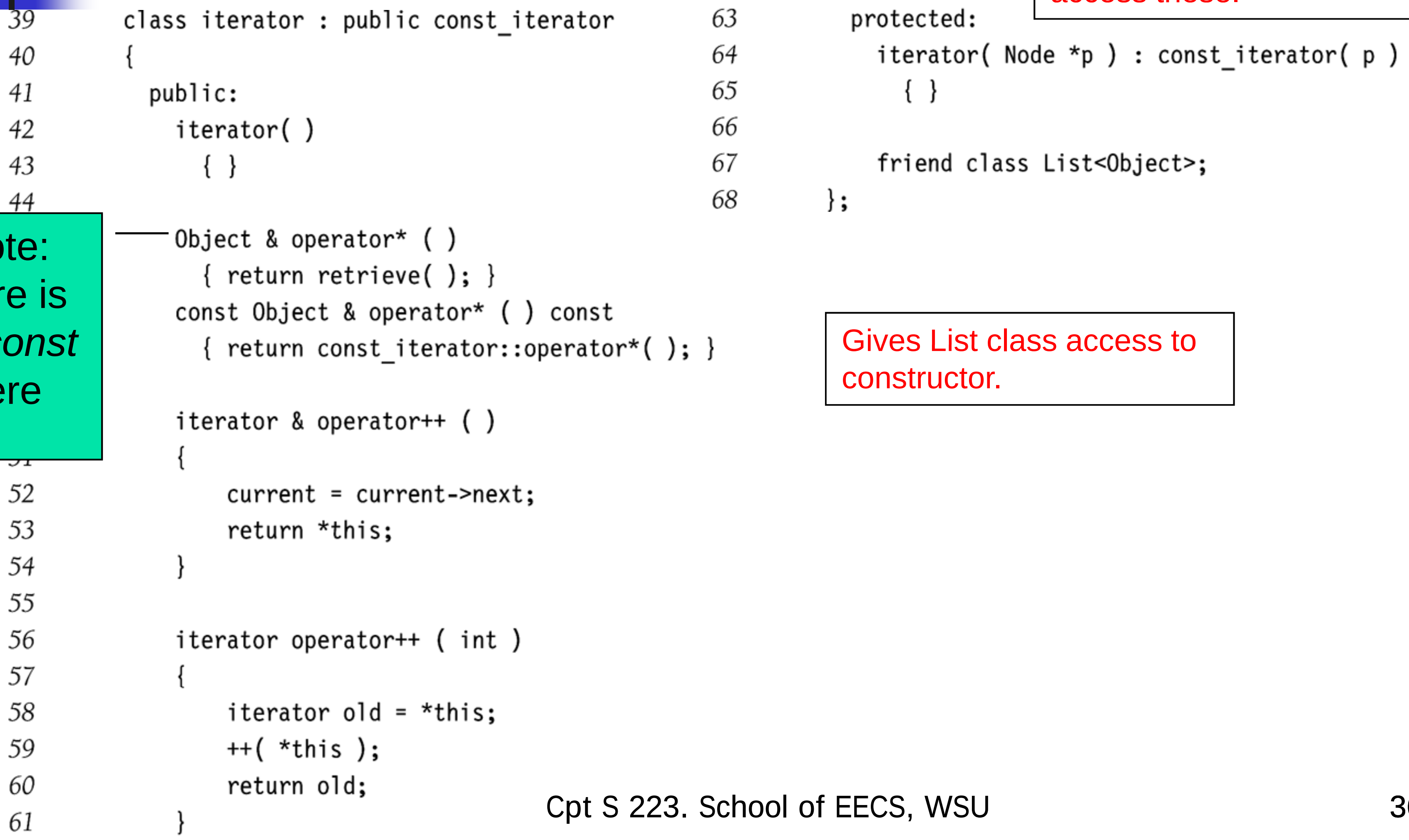
Implementation of List

```
class const_iterator
{
 public:
  const_iterator( ) : current( NULL )
   { }

  const Object & operator* ( ) const
   { return retrieve( ); }

  const_iterator & operator++ ( )
  {
    current = current->next;
    return *this;
  }

  const_iterator operator++ ( int )
  {
    const_iterator old = *this;
    ++( *this );
    return old;
  }

  bool operator== ( const const_iterator & rhs ) const
   { return current == rhs.current; }
  bool operator!= ( const const_iterator & rhs ) const
   { return !( *this == rhs ); }

 protected:
  Node *current;

  Object & retrieve( ) const
   { return current->data; }

  const_iterator( Node *p ) : current( p )
   { }

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

Allows inheriting classes to access these.

Gives List class access to constructor.

Implementation of List



```
    class iterator : public const_iterator
    {
      public:
        iterator( )
          { }

        Object & operator* ( )
          { return retrieve( ); }
        const Object & operator* ( ) const
          { return const_iterator::operator*( ); }

        iterator & operator++ ( )
        {
            current = current->next;
            return *this;
        }

        iterator operator++ ( int )
        {
            iterator old = *this;
            ++( *this );
            return old;
        }

      protected:
        iterator( Node *p ) : const_iterator( p )
          { }

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

Allows inheriting classes to access these.

Note: there is no *const* here

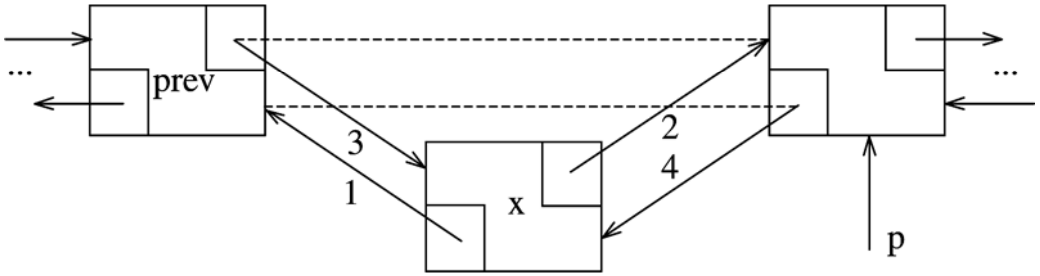
Gives List class access to constructor.

Implementation of List

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

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

List( const List & rhs )
{
    init( );
    *this = rhs;
}

const List & operator= ( const List & rhs )
{
    if( this == &rhs )
        return *this;
    clear( );
    for( const_iterator itr = rhs.begin( ); itr != rhs.end( ); ++itr )
        push_back( *itr );
    return *this;
}

void init( )
{
    theSize = 0;
    head = new Node;
    tail = new Node;
    head->next = tail;
    tail->prev = head;
}
```

Implementation of List

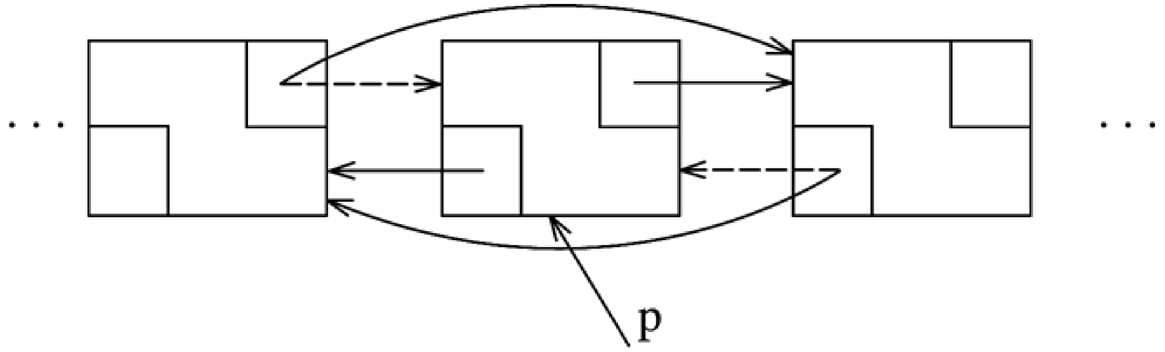
```
// Insert x before itr.
iterator insert( iterator itr, const Object & x )
{
    Node *p = itr.current;
    theSize++;
    return iterator( p->prev = p->prev->next = new Node( x, p->prev, p ) );
}
```



Implementation of List

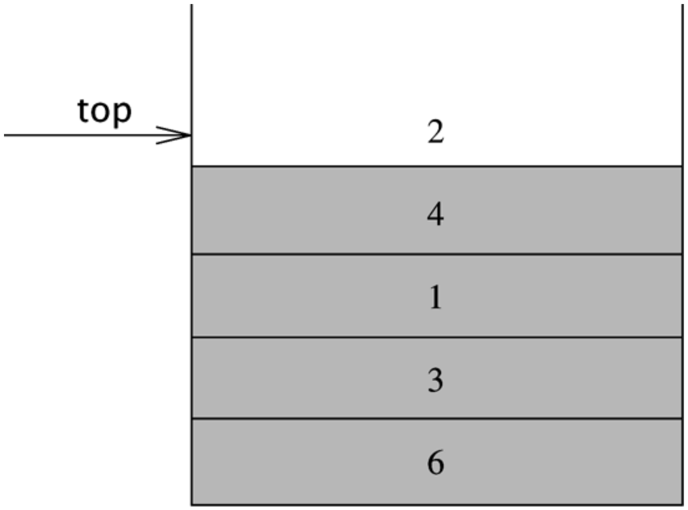
```
// Erase item at itr.
iterator erase( iterator itr )
{
    Node *p = itr.current;
    iterator retVal( p->next );
    p->prev->next = p->next;
    p->next->prev = p->prev;
    delete p;
    theSize--;

    return retVal;
}
```

```
iterator erase( iterator start, iterator end )
{
    for( iterator itr = from; itr != to; )
        itr = erase( itr );

    return to;
}
```



Stack ADT

- Stack is a list where insert and remove take place only at the "top"
- Operations
  - Push (insert) element on top of stack
  - Pop (remove) element from top of stack
  - Top: return element at top of stack
- LIFO (Last In First Out)



Stack Implementation

Linked List

```
template <typename Object>
class stack
{
  public:
    stack () {}
    void push (Object & x)
      { ?                   }
    void pop ()
      { ?
    Object & top ()
      { ?
  private:
    list<Object> s;
}
```

Vector

```
template <typename Object>
class stack
{
  public:
    stack () {}
    void push (Object & x)
      { ?
    void pop ()
      { ?
    Object & top ()
      { ?
  private:
    vector<Object> s;
}
```

Stack Implementation

Linked List

```
template <typename Object>
class stack
{
  public:
    stack () {}
    void push (Object & x)
      { ?                   }
    void pop ()
      { ?
    Object & top ()
      { ?
  private:
    list<Object> s;
}
```

Vector

```
template <typename Object>
class stack
{
  public:
    stack () {}
    void push (Object & x)
      { s.push_back (x); }
    void pop ()
      { s.pop_back (); }
    Object & top ()
      { s.back (); }
  private:
    vector<Object> s;
}
```

Running Times ?

Stack Implementation

Linked List

```
template <typename Object>
class stack
{
  public:
    stack () {}
    void push (Object & x)
      { s.push_front (x); }
    void pop ()
      { s.pop_front (); }
    Object & top ()
      { s.front (); }
  private:
    list<Object> s;
}
```

Vector

```
template <typename Object>
class stack
{
  public:
    stack () {}
    void push (Object & x)
      { s.push_back (x); }
    void pop ()
      { s.pop_back (); }
    Object & top ()
      { s.back (); }
  private:
    vector<Object> s;
}
```

C++ STL Stack Class

- Methods
  - Push, pop, top
  - Empty, size

```
#include <stack>

stack<int> s;

for (int i = 0; i < 5; i++ )
{
   s.push(i);
}
while (!s.empty())
{
   cout << s.top() << endl;
   s.pop();
}
```

Stack Applications

- Balancing symbols: ((()())(()))

```
stack<char> s;
while not end of file
{
  read character c
  if c = ‘(’
  then s.push(c)
  if c = ‘)’
  then if s.empty()
          then error
          else s.pop()
}
if (! s.empty())
then error
else okay
```

How does this work?

Stack Applications

- Postfix expressions
  - 1 2 * 3 + 4 5 * +
    - Means ((1 * 2) + 3) + (4 * 5)
  - HP calculators
  - Unambiguous (no need for paranthesis)
    - Infix needs paranthesis or else implicit precedence specification to avoid ambiguity
    - E.g., try a+(b*c) and (a+b)*c
  - Postfix evaluation uses stack

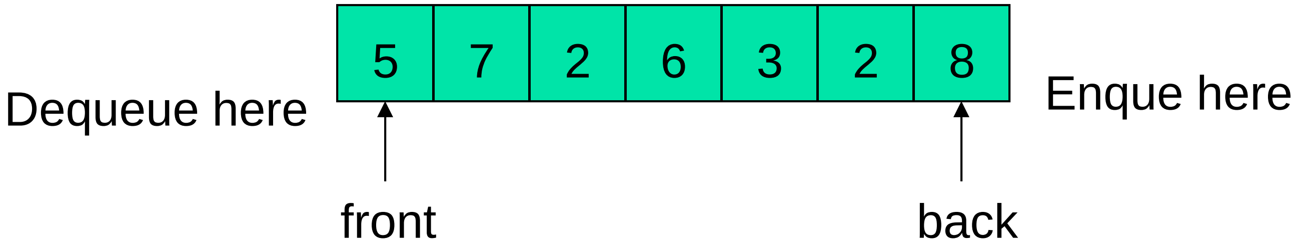
```
Class PostFixCalculator
{
  public:
    ...
    void Multiply ()
    {
      int i1 = s.top();
      s.pop();
      int i2 = s.top();
      s.pop();
      s.push (i1 * i2);
    }
  private:
    stack<int> s;
}
```

Stack Applications

- Function calls
- Programming languages use stacks to keep track of function calls
- When a function call occurs
    - Push CPU registers and program counter on to stack (“activation record” or “stack frame”)
    - Upon return, restore registers and program counter from top stack frame and pop

Queue ADT

- Queue is a list where insert takes place at the back, but remove takes place at the front
- Operations
  - Enqueue (insert) element at the back of the queue
  - Dequeue (remove and return) element from the front of the queue
  - FIFO (First In First Out)



5	7	2	6	3	2	8

Dequeue here

Enque here

front

back

Queue Implementation

Linked List

```
template <typename Object>
class queue
{
  public:
    queue () {}
    void enqueue (Object & x)
      { q.push_back (x); }
    Object & dequeue ()
      {
        Object & x = q.front ();
        q.pop_front ();
        return x;
      }
  private:
    list<Object> q;
}
```

How would the runtime change if vector is used in implementation?

Running time ?

C++ STL Queue Class

- Methods
  - Push (at back)
  - Pop (from front)
  - Back, front
  - Empty, size

```
#include <queue>

queue<int> q;

for (int i = 0; i < 5; i++ )
{
   q.push(i);
}
while (!q.empty())
{
   cout << q.front() << endl;
   q.pop();
}
```

Queue Applications

- Job scheduling
- Graph traversals
- Queuing theory

Summary

- Abstract Data Types (ADTs)
  - Linked list
  - Stack
  - Queue
- C++ Standard Template Library (STL)
- Numerous applications
- Building blocks for more complex data structures