CSCI 335 Software Design and Analysis III

Lists/Stacks/Queues and the STL

Chapter 3

List/Stack/Queue ADTs

- Already covered in 235
- We will emphasize the STL and use of iterators

Vector vs. List in the STL

- STL vector:
  - Constant time indexing
  - Slow to add data in the “middle”
  - Fast to add data at the end (not front)
- STL list:
  - Implemented as a doubly linked list
  - No indexing
  - Fast insertion/removal of items in any position

Vector vs. List in the STL

- STL vector and list:

```
void push_back(const Object &x)
void pop_back()
const Object &back() const  // returning non-const reference also provided
const Object &front() const // returning non-const reference also provided
```

- STL list only:

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

- STL vector only:

```
Object & operator[](int idx) // returning const ref. also provided
Object &at(int idx) // returning const ref. also provided
int capacity() const
void reserve(int new_capacity)
```

Iterators (STL)

- In STL position represented by iterator

```
list<string>::iterator itr1;
vector<int>::iterator itr2;
```

  - … note that book will use *iterator* as a shorthand

- Basic operations

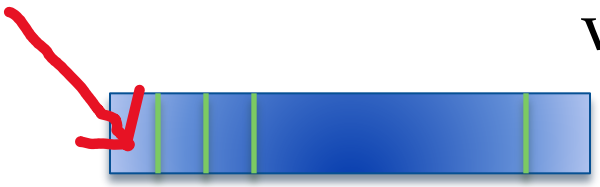
```
iterator begin();    // first item
iterator end();      // position after last item
```

```
for (int i=0; i != v.size(); ++i) //no iterators
      cout << v[i] << endl;
```

Iterator semantics

vector<int> v; // Same semantics for list, etc.

v.begin()



v.end()

v.begin() == v.end()

Empty vector

Iterators (STL)

Basic operations

```
iterator begin();  // first item
iterator end();    // position after last item

// Print out a vector using iterators.
for (vector<int>::iterator itr = v.begin();
     itr != b.end(); itr.??)
     cout << itr.?? << endl;
```

Iterator Methods

```
itr++
++itr
*itr // returns a reference to the object stored at itr's location
itr1 == itr2
itr1 != itr2
for (vector<int>::iterator itr = v.begin(); itr != v.end(); ++itr)
        cout << *itr << endl;
// or
vector<int>::iterator itr = v.begin();
while (itr != v.end())
      cout << *itr++ << endl;
```

Example (optional)

```
vector<string> v{“hi”, “there”, “ok”};
vector<string>::iterator itr = v.begin();
while (itr != v.end()) {
    string value1 = *itr;  // copy ?
    const string &value2 = *itr; // copy ?
    *itr = “a”;
    ++itr;
}
```

Operations that require iterators

```
// Insert prior
iterator insert(iterator pos, const Object &x);

// Delete at, return next
iterator erase(iterator pos);
iterator erase(iterator start, iterator end);
```

- Example: Routine that erases every other item on list or vector. How would we program this?

Example: Using erase on a list

```
// @ lst: A given list, or any object type that supports iterators and
//        erase.
// The function deletes every other element from lst, starting from
// the first item.
template <typename Container>
void RemoveEveryOtherItem(Container &lst) {
    typename Container::iterator itr = lst.begin();
    while (itr != lst.end()) {
      itr = lst.erase(itr);
      if (itr != lst.end()) ++itr;
    }
}
```

Example: Using erase on a list

```
// @ lst: A given list, or any object type that supports iterators and
//        erase.
// The function deletes every other element from lst, starting from
// the first item.
template <typename Container>
void RemoveEveryOtherItem(Container &lst) {
    auto itr = lst.begin();
    while (itr != lst.end()) {
      itr = lst.erase(itr);
      if (itr != lst.end()) ++itr;
    }
}
```

- List: empty

itr ^

- List: 10 -> 5 -> 6 -> 100 -> 7

  itr ^

  List: 5 -> 6 -> 100 -> 57

  itr ^ (after erase)

  itr ^ (after ++itr)

  List: 5 -> 100 -> 7

  itr ^ (afrer erase)

  itr ^ (after ++itr)

  List: 5 -> 100

  itr ^ (after erase)

  itr (++itr?)

Operations that require iterators

- Erase every 2nd item
- More efficient for list or vector?

Operations that require iterators

- Erase every 2nd item
- More efficient for list or vector?
  - 0.062 sec for 400,000 list of integers, 0.125 sec for 800,000
  - 2.5 min for 400,000 vector of integers, >10min for 800,000
  - Running time?

const_iterators

- Is there a need for a const iterator?
  - Note: *itr is a reference to the object at iterator's position
- Check generic routine that runs for both vector and list:

```
template <typename Container, typename Object>
void Change(Container &c, const Object &value) {
    auto itr = c.begin();
    while (itr != c.end())
        *itr++ = value;
}
```

const_iterators

- Any problems on next one?

```
void Print(const list<int> &lst, ostream &out = cout) {
    list<int>::iterator itr = lst.begin();
    while (itr != lst.end()) {
        out  << *itr << endl;
        *itr = 0;
        itr++;
    }
}
```

const_iterators

- Any problems?

```
void Print(const list<int> &lst, ostream & out = cout) {
    list<int>::iterator itr = lst.begin(); // ERROR
    while (itr != lst.end()) {
        out  << *itr << endl;
        *itr = 0;
        itr++;
    }
}
```

const_iterators

- Life is good.

```
void Print(const list<int> &lst, ostream & out = cout) {
    list<int>::const_iterator itr = lst.begin();
    while (itr != lst.end()) {
        out  << *itr << endl;
        // *itr = 0; can't change value
        itr++;
    }
}
```

const_iterators

- STL provides both iterator and const_iterator
- `operator*()` for const_iterator returns *constant reference*
  - `iterator begin();`
  - `const_iterator begin() const;`
  - `iterator end();`
  - `const_iterator end() const;`

Printing a container

```
// @ c: a given container.
// @ out: an output stream.
// Prints out the container on the output stream.
template <typename Container>
void Print(const Container &c, ostream &out = cout)  {
    out << "[ ";
    if (!c.empty()) {
        auto itr = begin(c);
        out << *itr++;
        while (itr != end(c))
            out << ", " << *itr++;
     }
     out << " ]" << endl;
}
```

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 a List

- How?
- Use a doubly-linked list (pointers)
- 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

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} {}

   };
  …
```

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

List class

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

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

```
44        Object & front( )
45          { return *begin( ); }
46        const Object & front( ) const
47          { return *begin( ); }
48        Object & back( )
49          { return *--end( ); }
50        const Object & back( ) const
51          { return *--end( ); }
52        void push_front( const Object & x )
53          { insert( begin( ), x ); }
54        void push_back( const Object & x )
55          { insert( end( ), x ); }
56        void pop_front( )
57          { erase( begin( ) ); }
58        void pop_back( )
59          { erase( --end( ) ); }
60
61        iterator insert( iterator itr, const Object & x )
62          { /* See Figure 3.18 */ }
63
64        iterator erase( iterator itr )
65          { /* See Figure 3.20 */ }
66        iterator erase( iterator start, iterator end )
67          { /* See Figure 3.20 */ }
```

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

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

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

```
69    private:
70      int   theSize;
71      Node *head;
72      Node *tail;
73
74      void init( )
75        { /* See Figure 3.16 */ }
76  };
```

Data members

Initialization

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} {}
      friend class List<Object>;  // Why friend??
}
```

How would you implement begin() in list?

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

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);
}
```

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>;
}
```

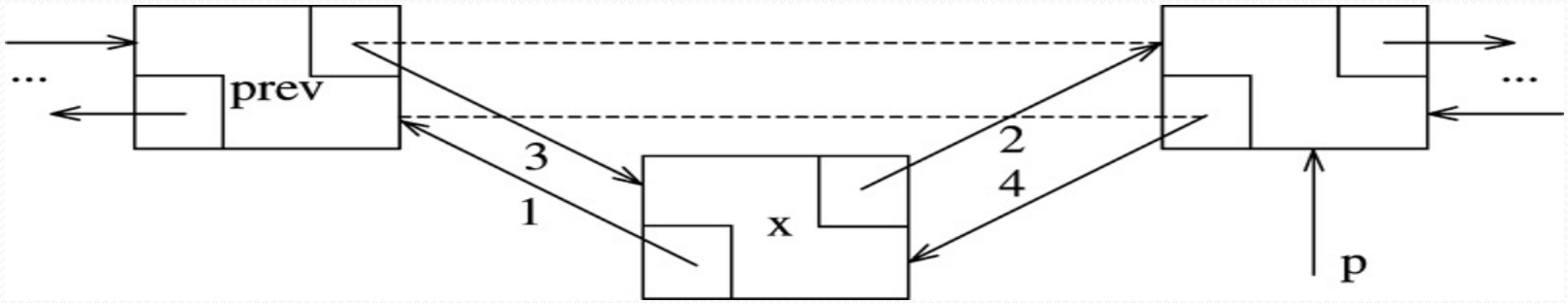
```
List() { init(); }

~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

Inserting before iterator




```
{ … // In List class:

// Insert x before itr.
iterator insert(iterator itr, const Object &x) {
    Node *p = itr.current_;
    ++size_;
    return iterator(p->prev_ = p->prev_->next_ = new Node(x, p->prev_, p));
}
```

4 3 1, 2

Why does it work?

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 !!
  --size_;
 return return_itr;
}
```

Possible error conditions?

- No error checking in code presented…

Possible error conditions?

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

Modified iterator class

```
1     protected:
2       const List<Object> *theList;
3       Node *current;
4
5       const_iterator( const List<Object> & lst, Node *p )
6         : theList( &lst ), current( p )
7       {
8       }
9
10      void assertIsValid( ) const
11      {
12          if( theList == NULL || current == NULL || current == theList->head )
13              throw IteratorOutOfBoundsException( );
14      }
```

Why ?

New constructor

Insert with some checks

```
1       // Insert x before itr.
2       iterator insert( iterator itr, const Object & x )
3       {
4           itr.assertIsValid( );
5           if( itr.theList != this )
6               throw IteratorMismatchException( );
7
8           Node *p = itr.current;
9           theSize++;
10          return iterator( *this,
11                           p->prev = p->prev->next = new Node( x, p->prev, p ) );
12      }
```

Exercise: 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) {

    // ….
}
```

Exercises

- Given two sorted lists L1 and L2, write a procedure to compute their intersection using the basic list operations. (3.4)
- Implement: const_iterator operator+(int k) const; (3.14)
- 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. (3.15)