Lists - I

The List ADT
List ADT

- A list is a dynamic ordered tuple of homogeneous elements
  \[ A_0, A_1, A_2, ..., A_{N-1} \]
  where \( A_i \) is the \( i \)-th element of the list
- Definition: The *position* of element \( A_i \) is \( i \); positions range from 0 to \( N-1 \) inclusive
- Definition: The *size* of a list is \( N \) (a list with no elements is called an “empty list”)
Operations on a List

- create an empty list
- destroy a list
- construct a (deep) copy of a list
- find(x) – returns the position of the first occurrence of x
- remove(x) – removes x from the list if present
- insert(x, position) – inserts x into the list at the specified position
- isEmpty() – returns true if the list has no elements
- makeEmpty() – removes all elements from the list
- findKth(position) – returns the element in the specified position

- The implementations of these operations in a class may have different names than the generic names above
STL vector

- The STL vector class template provides an array implementation of the List ADT
- The vector also support operations that are not specific to List
List operations using a vector

- vector supports constant time insertion at the “end” of the list using
  ```
  void push_back( const Object& x);
  ```
- vector supports constant time deletion from the “end” of the list using
  ```
  void pop_back( );
  ```
- vector supports constant time access to the element at the “end” of the list using
  ```
  const Object& back( ) const;
  ```
- vector supports constant time access to the element at the “front” of the list using
  ```
  const Object& front( ) const;
  ```
- vector also supports these List operations
  - bool empty( ) const; -- returns true if the vector is empty
  - void clear( ) -- removes all elements of the vector
  - Default constructor
  - Copy constructor
  - Assignment operator
iterators

- Some List operations (notably those that remove and insert from the middle of the list) require the notion of position.
- All STL containers use iterators and const_iterators to represent position within the container. Doing so provides a uniform interface for all STL container templates.
- For example, a position within a vector<int> is represented by the type vector<int>::iterator
- Questions
  - How does an application get an iterator?
  - What can iterators do?
  - What methods of vector require an iterator?
Creating an iterator

- The STL vector (and all STL containers) define the following methods for creating iterators:
  - `iterator begin()` — returns an iterator representing the first element of the vector
  - `iterator end()` — returns an iterator representing the end marker of the vector (i.e. the position after the last element of the vector).
Using iterators

We can now loop using iterators to access and print all elements of the vector of ints, v

```cpp
vector<int>::iterator itr;
for (itr = v.begin(); itr != v.end(); itr.xxx)
{
    cout << itr.zzz << endl; // element at itr’s position
}
```

itr.xxx must move the iterator to the next element
itr.zzz must retrieve the element at itr’s position
Iterator Operations

- Iterators are often considered an abstraction of a pointer and so support pointer semantics

`++itr` and `itr++` advance the iterator to the next element in the container

`*itr` returns the element stored at `itr`’s position

`itr1 == iter2` returns true if both iterators refer to the same position

`iter1 != iter2` returns true if the iterators refer to different positions

So, how do we complete the loop on the previous slide?
List remove and insert

- Now armed with iterators, we can insert and remove elements from the middle of a vector
  - `iterator insert (iterator pos, const Object & x)` inserts x into the vector prior to the position specified by the iterator
  - `iterator erase (iterator position)` removes the object at the position specified by the iterator
  - `iterator erase (iterator start, iterator end)` removes all objects beginning at position “start”, up to (but NOT including) position “end”
**const_iterator**

Since *itr* represents the element in a container, the statement *itr* = x; can be used to change the contents of the container.

This should not be allowed for const containers (typically passed as parameters), therefore containers also support const_iterators.

A const_iterator behaves the same as an iterator except the * operator returns a const reference to the element in the container and hence cannot be used to change the element.

Think of *iterator* as a mutator and *const_iterator* as an accessor.

Also note that the iterator class is derived from const_iterator via inheritance. Therefore an iterator can be used anywhere a const_iterator can be used but not vice-versa. However, good programming practice dictates that const_iterators be used with const containers.
const_iterator (cont’d)

The compiler forces you to use const_iterator for const containers by supplying two versions of begin() and end()

• iterator begin( );
• const_iterator begin( ) const;

• iterator end( );
• const_iterator end( ) const;

The “constness” of the methods is part of their signature.
Scanning a container

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

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