Iterators and Sequences
Containers and Iterators

- An **iterator** abstracts the process of scanning through a collection of elements.
- A **container** is an abstract data structure that supports element access through iterators:
  - `begin()`: returns an iterator to the first element
  - `end()`: return an iterator to an imaginary position just after the last element
- An iterator behaves like a pointer to an element:
  - `*p`: returns the element referenced by this iterator
  - `++p`: advances to the next element
- Extends the concept of **position** by adding a traversal capability.
Containers

- Data structures that support iterators are called containers
- Examples include Stack, Queue, Vector, List
- Various notions of iterator:
  - (standard) iterator: allows read-write access to elements
  - const iterator: provides read-only access to elements
  - bidirectional iterator: supports both \( ++p \) and \( --p \)
  - random-access iterator: supports both \( p+i \) and \( p-i \)
Iterating through a Container

- Let C be a container and p be an iterator for C
  
  \[
  \text{for } (p = C.\text{begin}(); p != C.\text{end}(); ++p) \]
  
  loop_body

- Example: (with an STL vector)
  
  typedef vector<int>::iterator Iterator;
  int sum = 0;
  for (Iterator p = V.\text{begin}(); p != V.\text{end}(); ++p)
    sum += *p;
  return sum;
Implementing Iterators

- **Array-based**
  - array $A$ of the $n$ elements
  - index $i$ that keeps track of the cursor
  - $\text{begin}() = 0$
  - $\text{end}() = n$ (index following the last element)

- **Linked list-based**
  - doubly-linked list $L$ storing the elements, with sentinels for header and trailer
  - pointer to node containing the current element
  - $\text{begin}() = \text{front node}$
  - $\text{end}() = \text{trailer node}$ (just after last node)
STL Iterators in C++

- Each STL container type C supports iterators:
  - C::iterator – read/write iterator type
  - C::const_iterator – read-only iterator type
  - C.begin(), C.end() – return start/end iterators

- This iterator-based operators and methods:
  - *p: access current element
  - ++p, --p: advance to next/previous element
  - C.assign(p, q): replace C with contents referenced by the iterator range [p, q) (from p up to, but not including, q)
  - insert(p, e): insert e prior to position p
  - erase(p): remove element at position p
  - erase(p, q): remove elements in the iterator range [p, q)
Sequence ADT

- The **Sequence ADT** is the union of the Array List and Node List ADTs
- **Elements accessed by**
  - Index, or
  - Position
- **Generic methods:**
  - size(), empty()
- **ArrayList-based methods:**
  - at(i), set(i, o), insert(i, o), erase(i)
- **List-based methods:**
  - begin(), end()
  - insertFront(o), insertBack(o)
  - eraseFront(), eraseBack()
  - insert(p, o), erase(p)
- **Bridge methods:**
  - atIndex(i), indexOf(p)
Applications of Sequences

- The Sequence ADT is a basic, general-purpose, data structure for storing an ordered collection of elements

- Direct applications:
  - Generic replacement for stack, queue, vector, or list
  - Small database (e.g., address book)

- Indirect applications:
  - Building block of more complex data structures
Linked List Implementation

- A doubly linked list provides a reasonable implementation of the Sequence ADT
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes
- Position-based methods run in constant time
- Index-based methods require searching from header or trailer while keeping track of indices; hence, run in linear time

© 2010 Goodrich, Tamassia
Array-based Implementation

- We use a circular array storing positions.
- A position object stores:
  - Element
  - Index
- Indices $f$ and $l$ keep track of first and last positions.
Comparing Sequence Implementations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Array</th>
<th>List</th>
</tr>
</thead>
<tbody>
<tr>
<td>size, empty</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>atIndex, indexOf, at</td>
<td>1</td>
<td>(n)</td>
</tr>
<tr>
<td>begin, end</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>set(p,e)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>set(i,e)</td>
<td>1</td>
<td>(n)</td>
</tr>
<tr>
<td>insert(i,e), erase(i)</td>
<td>(n)</td>
<td>(n)</td>
</tr>
<tr>
<td>insertBack, eraseBack</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>insertFront, eraseFront</td>
<td>(n)</td>
<td>1</td>
</tr>
<tr>
<td>insert(p,e), erase(p)</td>
<td>(n)</td>
<td>1</td>
</tr>
</tbody>
</table>