Containers

- We call any data structure or ADT that stores any collection of elements a container.
- A container may contain elements in a sequence or an unordered collection, such as a set.
- However, it is assumed that the elements of a container can be arranged in some linear order.
Position ADT

- The **Position** ADT models the notion of the place within a container data structure where a single object is stored.
- It gives a unified view of diverse ways of storing data, such as:
  - a cell of an array
  - a node of a linked list
- Just one method:
  - object p.element(): returns the element at position
  - In C++ it is convenient to implement this as *p
Position ADT

- **Positions** are always defined with respect to their neighbors. Unless it is the first or last element in the container, a position \( q \) is always “after” some position \( p \) and “before” some position \( r \). This is all relative to the linear ordering we assumed for the container.

- A position does not change if the element moves within the container, or if the element is swapped or replaced by another.

- A position is **invalidated** if the element it is associated with is explicitly removed.
Iterator ADT

- Extends the concept of *position* by adding a traversal capability
- An *iterator* abstracts the process of scanning through a collection of elements
- An iterator behaves like a pointer to an element
  - *p*: returns the element referenced by this iterator
  - ++p*: advances to the next element
Containers, refined

- We re(de)fine container to mean a data structure that stores a collection of elements and 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
- 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
  for (p = C.begin(); p != C.end(); ++p)
  
  *loop_body*

- Example: (with an STL vector V)
  typedef vector<int>::iterator Iterator;
  int sum = 0;
  for (Iterator p = V.begin(); p != V.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)
List ADT

- The List ADT models a sequence of positions storing arbitrary objects
- It establishes a before/after relation between positions
- Generic methods:
  - size(), empty()
- Iterators:
  - begin(), end()
- Update methods:
  - insertFront(e), insertBack(e)
  - eraseFront(), eraseBack()
- Iterator-based update:
  - insert(p, e)
  - remove(p)
List ADT

- The update methods are for convenience. For example, `insertFront(e)` is short for `insert(L.begin(), e)`.
- An error condition occurs if an invalid iterator is passed as an argument to an iterator-based update method.
Doubly Linked List Implementation

- A doubly linked list provides a natural implementation of the List ADT
- Nodes implement Iterator and store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes
Insertion

- We visualize operation $\text{insert}(p, x)$, which inserts $x$ before $p$.
Insertion Algorithm

Algorithm $\text{insert}(p, e)$: \{insert $e$ before $p$\}
Create a new node $v$
$v \rightarrow \text{element} = e$
$u = p \rightarrow \text{prev}$
$v \rightarrow \text{next} = p; \quad p \rightarrow \text{prev} = v$ \{link in $v$ before $p$\}
$v \rightarrow \text{prev} = u; \quad u \rightarrow \text{next} = v$ \{link in $v$ after $u$\}
Deletion

- We visualize $\text{remove}(p)$
Deletion Algorithm

Algorithm \texttt{remove}(p):
\begin{itemize}
    \item $u = p\rightarrow\text{prev}$
    \item $w = p\rightarrow\text{next}$
    \item $u\rightarrow\text{next} = w$ \{linking out p\}
    \item $w\rightarrow\text{prev} = u$ \{invalidate p\}
\end{itemize}
The text shows a doubly-linked list implementation where there are separate Node and Iterator classes.

In this approach, the Node class remains simple:

```cpp
class Node {
    Elem elem;
    Node* prev;
    Node* next;
};
```
The iterator class requires a number of its own functions, which would water down Node’s interface if the two were combined.

```cpp
class Iterator {
public:
    Elem& operator*();
    bool operator==(const Iterator& p) const;
    bool operator!=(const Iterator& p) const;
    Iterator& operator++();
    Iterator& operator--();
friend class NodeList;
private:
    Node* v;
    Iterator(Node* v);
};
```
NodeList class

The NodeList class includes the Node class and the Iterator class as nested classes.

class NodeList {
private:
   // insert Node declaration here
public:
   // insert Iterator declaration here

    NodeList();
    int size() const;
    bool empty() const;
    Iterator begin() const;
    Iterator end() const;
    void insertFront(const Elem& e);
    void insertBack(const Elem& e);

    void insert(const Iterator& p, const Elem& e);
    void eraseFront();
    void eraseBack();
    void erase(const Iterator& p);
   // ...
private:
    int n;
    Node* header;
    Node* trailer;

};
The implementation of the Iterator must refer to the Iterator as `NodeList::Iterator`.

```cpp
class NodeList::Iterator {
public:
  NodeList::Iterator(Node* u) { v = u; }
  Elem& operator*() { return v->elem; }
  bool operator==(const Iterator& p) const { return v == p.v; }
  NodeList::Iterator& operator++() { v = v->next; return *this }
};
```

// (operator!= and operator-- are similar)
Performance

- In the implementation of the List ADT by means of a doubly linked list
  - The space used by a list with $n$ elements is $O(n)$
  - The space used by each position of the list is $O(1)$
  - All the operations of the List ADT run in $O(1)$ time
  - Operation `element()` of the Position ADT runs in $O(1)$ time