We’ve emphasized the importance of software reuse. Recognizing that many data structures and algorithms are commonly used, the C++ standard committee added the Standard Template Library (STL) to the C++ Standard Library. The STL defines powerful, template-based, reusable components that implement many common data structures and algorithms used to process those data structures.

**Standard Template Library (STL)**

- is a library of generic container classes which are both efficient and functional
- C++ STL developed in early 90’s at Hewlett Packard Laboratories
  - Alex Stepanov and Meng Lee
- became part of C++ standard in 1994
  - implementations available by late 90’s

**Main Ideas**

- General purpose: generic data structures & algorithms, templates
- Flexibility: Allows for many combinations of algorithm-container
- Simple & uniform interface: interface through templates (not inheritance)
- Efficiency

**STL components**

- **Containers**: templates for classes which hold a collection of elements
- **Algorithms**: templates for functions which operate on a range of elements from a container
  - range is specified by iterators
- **Iterators**: allow access to the elements in a container
  - allow for movement from one element to another

**Components**

- **Containers**: Each STL container has associated member functions. A subset of these member functions is defined in all STL containers. The containers are divided into three major categories—sequence containers, associative containers and container adapters.
- **Iterators**: allow access into containers. They have properties similar to those of pointers, and are used by programs to manipulate the STL-container elements.
- **Algorithms**: STL algorithms are functions that perform such common data manipulations as searching, sorting and comparing elements (or entire containers). The STL provides approximately 70 algorithms. Most of them use iterators to access container elements. Each algorithm has minimum requirements for the types of iterators that can be used with it.
Containers

Containers
Containers in the STL fall into two categories: sequential and associative.

The sequential containers are `vector`, `list`, and `deque`.

The associative containers are `map`, `multimap`, and `set`.

In addition, several containers are called abstract data types, which are specialized versions of other containers. These are `stack`, `queue`, and `priority_queue`.

Sequential Containers: Elements of the sequential containers can be accessed by position, for example, by using an index. An ordinary C++ array is an example of a sequential container.

One problem with a ordinary C++ array is that you must specify its size at compile time. That is, in the source code. You must specify an array large enough to hold what you guess is the maximum amount of data.

When the program runs, you will either waste space in memory by not filling the array or run out of space.

The STL provides the `vector` container to avoid these difficulties.

The STL provides the `list` container, which is based on the idea of a linked list.

The third sequence container is the `deque`, which can be thought of as a combination of a stack and a queue. A deque combines these approaches so you can insert or delete data from either end. The word "deque" is derived from Double-Ended Queue.

Basic Sequential Containers

<table>
<thead>
<tr>
<th>Container</th>
<th>Description</th>
<th>Characteristics</th>
<th>Advantages and Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>vector</td>
<td>Replaces the standard C++ array</td>
<td>Increasing memory and access time</td>
<td>Quick random access (by index), easy to use</td>
</tr>
<tr>
<td>deque</td>
<td>Replaces the standard C++ array</td>
<td>Increasing memory and access time</td>
<td>Quick to insert or delete at any location</td>
</tr>
</tbody>
</table>

STL Containers (part 1 of 2)

<table>
<thead>
<tr>
<th>Standard Library container class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vector</code></td>
<td>Sequential container that stores elements of a common type.</td>
</tr>
<tr>
<td><code>deque</code></td>
<td>Stacks and queues are combinations of stacks and queues.</td>
</tr>
<tr>
<td><code>stack</code></td>
<td>Simple stack that supports the LIFO (last-in, first-out) discipline.</td>
</tr>
<tr>
<td><code>queue</code></td>
<td>Simple queue that supports the FIFO (first-in, first-out) discipline.</td>
</tr>
<tr>
<td><code>priority_queue</code></td>
<td>Simple priority queue that supports the HPQ (highest-priority-first) discipline.</td>
</tr>
</tbody>
</table>

STL Containers - Common Member Functions (Part 1 of 3)

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>insert</code></td>
<td>Inserts an item in the container.</td>
</tr>
<tr>
<td><code>erase</code></td>
<td>Removes the member from the container.</td>
</tr>
<tr>
<td><code>assignment</code></td>
<td>Assigns an object to another.</td>
</tr>
<tr>
<td><code>swap</code></td>
<td>Swaps the elements of two containers.</td>
</tr>
<tr>
<td><code>max_element</code></td>
<td>Returns the maximum element of the container.</td>
</tr>
<tr>
<td><code>min_element</code></td>
<td>Returns the minimum element of the container.</td>
</tr>
<tr>
<td><code>begin</code></td>
<td>Returns the first element of the container.</td>
</tr>
<tr>
<td><code>end</code></td>
<td>Returns the past-the-end iterator.</td>
</tr>
<tr>
<td><code>reverse</code></td>
<td>Reverses the elements of a container.</td>
</tr>
<tr>
<td><code>equal_range</code></td>
<td>Returns the range of elements that are equal to a given element.</td>
</tr>
</tbody>
</table>

STL Containers (part 2 of 2)

<table>
<thead>
<tr>
<th>Standard Library container class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>list</code></td>
<td>Sequence container that supports the FIFO (first-in, first-out) discipline.</td>
</tr>
<tr>
<td><code>set</code></td>
<td>Association container that stores elements of a common type.</td>
</tr>
<tr>
<td><code>map</code></td>
<td>Association container that stores key-value pairs.</td>
</tr>
<tr>
<td><code>unordered_set</code></td>
<td>Association container that stores elements of a common type.</td>
</tr>
<tr>
<td><code>unordered_map</code></td>
<td>Association container that stores key-value pairs.</td>
</tr>
</tbody>
</table>

STL Containers - Common Member Functions (Part 2 of 3)

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>begin</code></td>
<td>Returns the first element of the container.</td>
</tr>
<tr>
<td><code>end</code></td>
<td>Returns the past-the-end iterator.</td>
</tr>
<tr>
<td><code>reverse</code></td>
<td>Reverses the elements of a container.</td>
</tr>
<tr>
<td><code>equal_range</code></td>
<td>Returns the range of elements that are equal to a given element.</td>
</tr>
</tbody>
</table>

Common member functions for most STL containers.
STL Containers - Common Member Functions

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>resize</code></td>
<td>The two versions of this function move or resize the container.</td>
</tr>
<tr>
<td><code>clear</code></td>
<td>Erases all elements from the container.</td>
</tr>
</tbody>
</table>

Common member functions for most STL containers.

STL Containers header files

- The standard library header files for each of the Standard Library containers are shown below.

<table>
<thead>
<tr>
<th>Header File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vector</code></td>
<td>The type of element stored in the container.</td>
</tr>
<tr>
<td><code>list</code></td>
<td>A container reference to the type of element stored in the container.</td>
</tr>
<tr>
<td><code>map</code></td>
<td>An iterator that points to an element of the container's element type.</td>
</tr>
<tr>
<td><code>set</code></td>
<td>A container reference to the type of element stored in the container.</td>
</tr>
<tr>
<td><code>deque</code></td>
<td>The type of element stored in the container.</td>
</tr>
<tr>
<td><code>pair</code></td>
<td>A container reference to the type of element stored in the container.</td>
</tr>
<tr>
<td><code>map</code></td>
<td>A container reference to the type of element stored in the container.</td>
</tr>
<tr>
<td><code>set</code></td>
<td>An iterator that points to an element of the container's element type.</td>
</tr>
</tbody>
</table>

Containers – typedefs

- The typedefs shown in the next two tables are used in generic declarations of variables, parameters to functions and return values from functions. For example, `value_type` in each container is always a typedef that represents the type of value stored in the container.
- The first-class containers are vectors, deques, lists, sets, multisets, maps and multimeaps.

Containers – typedefs

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>value_type</code></td>
<td>The type of element stored in the container.</td>
</tr>
<tr>
<td><code>reference</code></td>
<td>A container reference to the type of element stored in the container.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>front()</code></td>
<td>A container reference to the type of element stored in the container.</td>
</tr>
<tr>
<td><code>back()</code></td>
<td>An iterator that points to an element of the container's element type.</td>
</tr>
<tr>
<td><code>begin()</code></td>
<td>A container reference to the type of element stored in the container.</td>
</tr>
<tr>
<td><code>end()</code></td>
<td>An iterator that points to an element of the container's element type.</td>
</tr>
</tbody>
</table>

typedefs used in first-class containers.

vector<T>

- growable, self-contained, type-independent array
- element type specified when a vector is declared
- `vector<double>` numbers;
- `vector<cashier>` checkoutStations;
- has a set of operations (methods)
- A vector changes size dynamically.
- A vector supports random-access.
- All STL algorithms can operate on a vector.
- Examples
  - `vector<int> v1;` (capacity is 0)
  - `vector<float> v2 (10);` (capacity is 10; size is 10)
  - `vector<string> v3 (5, “C++”);` (capacity is 5; size is 5)
The Vector

- An enhancement of the array.
- Can be used like an array:
  - Access a value via an index
    \[ x = v[i]; \]
    \[ v[j] = z; \]
- Additional capabilities:
  - Increase or decrease its length
  - Insert an element at a specified position
  - Remove an element from a specified position

Vector Example

```cpp
// Create a vector to hold strings.
vector<string> my_vector;
// Add some entries.
my_vector.push_back("Bashful");
my_vector.push_back("Awful");
my_vector.push_back("Jumpy");
my_vector.push_back("Happy");
```

Vector Example (2)

```cpp
my_vector[0] [1] [2] [3] [4]
"Bashful" "Awful" "Jumpy" "Happy"
```

```cpp
my_vector.insert(2, "Doc");
```

After insertion of "Doc" before the third element

```cpp
Had to slide [2][3] to [3][4]
```

Vector Example (3)

```cpp
my_vector[0] [1] [2] [3] [4]
"Bashful" "Awful" "Doc" "Jumpy" "Happy"
```

After insertion of "Doc" before the third element

```cpp
Cheap to add at end
```

```cpp
my_vector.push_back("Dopey"); // add at end
```

After insertion of "Dopey" at the end

Size and capacity Functions with vector Container

- Initially returns 0 for size and capacity of vector v.
- Function size—available in every container—returns the number of elements currently stored in the container.
- Function capacity returns the number of elements that can be stored in the vector before the vector needs to dynamically resize itself to accommodate more elements.

Some vector<T> methods

- V.capacity() //size of array currently allocated
- V.size() //number of values V contains
- V.empty() //true iff V.size() is 0
- V.reserve(n) //grow V so its capacity is n
- V.push_back(val) //add val at end of V
- V.pop_back() //erase V's last element
- V[i] //access element of V whose index is i
- V.at(i) //access element of V whose index is i
```cpp
#include <iostream>
#include <vector>
using namespace std;

bool Search(const vector<int> & V, int item) {
    int p = 0;
    while (p < V.size()) {
        if (item == V[p]) // or V.at(p)
            return true;
        else p++;
    }
    return false;
}

int main () {
    vector<int> numbers;
    int number;
    while (cin >> number) { // enter <control> D to stop the loop
        if (Search(numbers, number))
            cout << "Duplicate" << endl;
        else numbers.push_back(number);
    }
    cout << "number of unique values: " << numbers.size();
    return 0;
}
```

```
#include <list>
#include <iostream>
using namespace std;

int main() {
    list<int>L;
    L.push_back(9);
    L.push_back(7);
    L.push_back(5);
    L.push_back(3);
    list<int>::iterator p;
    for (p = L.begin(); p != L.end(); p++)
        cout << *p << endl;
    for (p = L.begin(); p != L.end(); p++)
        (*p)++;
    for (p = L.begin(); p != L.end(); p++)
        cout << *p << endl;
    return 0;
}
```

---

**list<T> class**

- another STL container class
- used for storing a linear collection of like items
- comparison to a vector?
  - linked list vs array is the underlying data structure
  - no indexing (iterators are bidirectional)
  - inserts and deletes anywhere are done in a constant amount of time

---

**Basic list class methods**

- list(); // construct an empty list
- list(const list<T> & aList); // copy constructor
- ~list(); // destructor
- list<T> operator=(const list<T> & aList); // assignment operator
- bool empty();
- int size();

---

**Some more list methods**

- L.push_back(value) // append value to L
- L.push_front(value) // insert value at front of L
- L.insert(pos, value) // insert value into L at position indicated by iterator pos
- L.front() // return L's first element
- L.back() // return L's last element
- L.begin() // return an iterator positioned at start
- L.end() // return the past the end iterator
- L.sort() // sort L's elements using <
list in C++ (STL)

- Syntax
  ```
  #include <list>
  using namespace std;
  list<T> collection;
  // T: datatype stored in vector
  ```

list Operations

- Adding elements
  - push_back - At the end
  - push_front - At the beginning
  - insert - Anywhere

- Removing an element
  - pop_back - From the end
  - pop_front - From the beginning
  - erase, remove - Anywhere

list Operations (2)

- Information accessors/inspectors
  - size - How many?
  - empty - Are there none?

- Modification
  - sort - NOT part of <algorithm>
  - reverse - change the order

Member Functions of Sequential Container

- Vectors:
  - Smart array.
  - They manage storage allocation for you, expanding and contracting the size of the vector as you insert or erase data.
  - You can use vectors much like arrays, accessing elements with the [] operator.
  - They allow fast access to and manipulation of elements.
  - They are also fast to add (or push) new data items onto the end (or back) of the vector.
  - When this happens, the vector's size is automatically increased to hold the new data.

- Some member functions:
  - `size()`: Returns the number of items in the container.
  - `empty()`: Returns true if container is empty.
  - `max_size()`: Returns size of the largest possible container.
  - `begin()`: Returns an iterator to the start of the container for iterating forward through the container.
  - `end()`: Returns an iterator to the end of the container, used to end forward iteration.
  - `rbegin()`: Returns a reverse iterator to the end of the container for iterating backward through the container.
  - `rend()`: Returns a reverse iterator to the beginning of the container, used to end backward iteration.

Examples

- Lists:
  - Doubly linked list.
  - Each element contains a pointer to the next element and to the previous element.
  - The `size()` function returns the number of elements in the list.
  - The `empty()` function checks if the list is empty.
  - The `push_back()` function adds an element to the end of the list.
  - The `push_front()` function adds an element to the beginning of the list.
  - The `erase()` function removes an element from the list.

- Doubles:
  - A `double` is a variable that supports floating-point numbers with a high degree of precision.
  - Unlike a vector or list, a double can be accessed on the front as well as the back.
  - It supports double-ended access, allowing `push_front()`, `push_back()`, and `pop_front()`, `pop_back()` operations.
  - Memory is allocated dynamically for vectors and queues. A vector always allocates a contiguous block of memory, while a stack is implemented using a linked list.
Iterators have many features in common with pointers and are used to point to the elements of first-class containers (vectors, deque's, lists, sets, multisets, maps and multimeaps).

Iterators hold state information sensitive to the particular containers on which they operate; thus, iterators are implemented appropriately for each type of container.

Certain iterator operations are uniform across containers.

For example, the dereferencing operator (*) dereferences an iterator so that you can use the element to which it points.

The ++ operation on an iterator moves it to the next element of the container (much as incrementing a pointer into an array aims the pointer at the next element of the array).

STL first-class containers provide member functions begin and end.

Function begin returns an iterator pointing to the first element of the container.

Function end returns an iterator pointing to the first element past the end of the container (an element that doesn’t exist).

If iterator i points to a particular element, then ++i points to the “next” element and *i refers to the element pointed to by i.

The iterator resulting from end is typically used in an equality or inequality comparison to determine whether the “moving iterator” (i in this case) has reached the end of the container.

An object of type iterator refers to a container element that can be modified.

An object of type const_iterator refers to a container element that cannot be modified.

Iterators are allow to traverse sequences.

Methods
- operator*
- operator++, and operator--

Different types of iterators - to support read, write and random access.

Containers define their own iterator types.

Changing the container can invalidate the iterator.
STL iterators

- Iterators are "pointer-like" objects
  - Provide a generic way to access the elements of any container class
  - Many STL algorithms require iterators as arguments
  - Some STL algorithms return an iterator
- Each STL container class has an iterator class associated with it
  - Vector<T>::iterator
  - List<T>::iterator
- Stack<T> and Queue<T> don't have iterators

Instantiating STL Container

Instantiating an STL container object is easy. First, you need include an appropriate header file. Then you use the template format with the kind of elements to be stored as the parameter. Examples might be:

```cpp
#include <vector>

vector<ComplexT> vec; // create a vector of complex numbers
```

Iterators

- Iterators are "pointer-like" objects to access elements in a container.

In general, the following holds true of iterators:

- A reference to iterator iter refers to the element that the iterator points to (differentially. *iter refers to the object. The iterator points to the element in the sequence to which the element at the index corresponds. The iterator represents the element at the index. The iterator refers to the value of the element at the index. The iterator provides an interface to access the value of the element at the index.

For the containers that have their elements stored in a contiguous manner, only pointers are available. For example, the list, deque, and set use pointers for these containers. The STL containers include header files that define iterators. They also provide iterators for the standard template library containers (stack, priority_queue, map, and set).

Iterator Types

<table>
<thead>
<tr>
<th>Output</th>
<th>Input</th>
<th>Forward</th>
<th>Bi-directional</th>
<th>Random</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>x = *i</td>
<td>x = *i</td>
<td>x = *i</td>
<td>x = *i</td>
</tr>
<tr>
<td>Write</td>
<td>x = *i</td>
<td>x = a</td>
<td>x = a</td>
<td>a = x</td>
</tr>
<tr>
<td>Iteration</td>
<td>++ i</td>
<td>++ i</td>
<td>++ i, -- i</td>
<td>++ i, -- i, + i, - i</td>
</tr>
<tr>
<td>Comparison</td>
<td>x == y</td>
<td>x == y</td>
<td>x == y, !x = y</td>
<td>x &lt; y, x &gt; y</td>
</tr>
</tbody>
</table>

- Output: write only and can write only once
- Input: read many times each item
- Forward supports both read and write
- Bi-directional support also decrement
- Random supports random access (just like C pointer)

Data Access

Data access allows us to access elements in a container. The container has a function called "begin" that returns an iterator pointing to the first element of the container. The function "end" returns an iterator pointing to the next element of the container.

```cpp
int main() {
    int arr[] = {2, 4, 6, 8};
    vector<int> v;
    sort(v.begin(), v.end());
    for (auto i : v) {
        cout << i << " ";
    }
    return 0;
}
```

An equivalent approach to iterating through the elements of a container is using a for loop instead of a while loop:

```cpp
for (auto i : v) {
    cout << i << " ";
}
```
**Data Insertion**

```c
int main()
{
    int arr[] = {2, 4, 6, 8, 10};
    // array of ints
    int* iter = arr;
    // iterator
    *iter = 12;
    // increment iterator
    *iter = (arr + 5); // pointer to the end
    // character array
    char str[] = "Hello World!"
    // array of chars
    char* ptr = str;
    // pointer
    *ptr = 'Z';
    // character
}
```

**Constant Iterators**

The STL defines `const_iterator` types to be able to visit a range of the elements of a constant container. Whereas the elements of the list in the previous example could have been stored, the elements of the vector in the next example are `const_iterator` and `const_iterator` are required:

```cpp
int arr[] = {2, 4, 6, 8, 10};
vector<int> vec(arr, arr + 5);
const_iterator iter = vec.begin();
// iterator to vector of ints
cout << *iter << endl;
// display first element
```

**Example**

```cpp
using namespace std;

int main()
{
    int arr[] = {2, 4, 6, 8, 10};
    // array of ints
    int* iter = arr;
    // iterator
    *iter = 12;
    // increment iterator
    *iter = (arr + 5); // pointer to the end
    // character array
    char str[] = "Hello World!"
    // array of chars
    char* ptr = str;
    // pointer
    *ptr = 'Z';
    // character
}
```

**STL Algorithms**

- **An algorithm is an operation (function) that can be applied to many STL containers.**
- **The STL algorithms are generic - they can operate on a variety of data structures.**
- **STL container classes such as `vector` and `list`**
- **Program-defined data structures**
- **Behavior must satisfy the requirements of a particular algorithm.**
- **STL algorithms achieve generality by accessing and traversing the elements of a container indirectly through iterators.**
- **Iterators must be settable and dereferencable.**

**Sort Algorithm**

- The `sort` algorithm is an operation (function) that can be applied to many STL containers.
- Notable exception: list container (has its own method).
- `sort()` orders a container's contents in ascending order.
- As defined by the `operator<()` as applied to the container's elements.
- Programmer-defined types can be sorted just as easily as a built-in type.

**find Algorithm**

- Searches a subrange of the elements in a container (or all the elements).
- Looks for an element that is "equal to" a specified value; stops when it finds the first element equal to the specified value.
- The equality operator `==` must be defined for the type of the container's elements.
- Search value must be of the same type as the elements stored in the container.
- Or of a type that the compiler can automatically convert.
- Return value is an iterator specifying the position of the first matching element.
- If no matching element is found, the return value is equal to the iterator specifying the end of the element subrange.
STL Algorithms

- are function templates designed to operate on a sequence of elements rather than methods
- the sequence is designated by two iterators
- most container classes have the following two methods:
  - `begin()` - returns an iterator positioned at the container’s first element
  - `end()` - returns an iterator positioned past the container's last element (past-the-end)
- `c.begin()`, `c.end()` specifies a sequence which contains all elements of the container `c`

Algorithms cont’d

- Most STL algorithms work on sequences
- Sequences are passed as two iterators:
  - beginning element
  - element one after last
  

- Algoritms depend on iterator type not on container type

Example:

```cpp
#include <iostream>
#include <algorithm>

using namespace std;

int main()
{   // empty list holds 5 vars
    list<int> lt;   // standard
    set<int> st;   // set with order of elements
    for (int i = 0; i < 10; i++)
    {   // for loop with data
        if (i == 2)
        {   // if condition
            auto it = lt.begin();   // it is an iterator
            if (it != lt.end())
            {
                it = it + 1;         // advance it
            }
        }
    }
    return 0;
}
```

```
// with no data
list<int> lt;
list<int> li;
list<int> la;
list<int> lb;
list<int> lc;
list<int> ld;
list<int> le;
list<int> lf;
list<int> lg;
list<int> lh;
list<int> li;
```
Using sort()

Example:
```cpp
void sort(Iterator first, Iterator last);  
void sort(Iterator first, Iterator last, Comp);  
```

Example:
```cpp
#include <iostream>  
#include <algorithm>  
#include <string>  
using namespace std;  

int main()  
{  
    string word[] = {"fahrenheit", "kelvin", "vital", "meas", "pitch", "ozone"};  
    for_each(word, word+7, print)  
    {  
        cout << *it << " ";  
    }  
    cout << endl;  
    return 0;  
}
```

Using sort() cont'd

Example:
```cpp
void sort(Iterator first, Iterator last, Comp);  
```

In this case, the elements in the range (first, last) are sorted in ascending order using the operator() of the underlying type.

Example:
```cpp
void sort(Iterator first, Iterator last);  
```

The `sort` function can be used with the following:
```cpp
#include <iostream>  
#include <algorithm>  
#include <string>  
using namespace std;  

int main()  
{  
    string word[] = {"fahrenheit", "kelvin", "vital", "meas", "pitch", "ozone"};  
    for_each(word, word+7, print)  
    {  
        cout << *it << " ";  
    }  
    cout << endl;  
    return 0;  
}
```

for_each() algorithm

The `for_each()` algorithm is used to do something to every element in a container. You write your own function to determine what that “something” is. Your function isn’t changed by the algorithm; it is used as is. In our case, we are using an iterator to loop through the container.

```cpp
for_each(SortedObject first, SortedObject last, Function funct)  
```

Each of the elements in the range [first, last) is mapped to the function `funct`. The function may alter the elements it receives. The `for_each()` algorithm is used to:

- Loop through a list of elements.
- Process each element in a list.
- Call a function for each element.

```cpp
for_each(word, word+7, print)  
```

Iterators and Algorithms

Iterators are objects that provide a way to access elements in a container. They are used to navigate through the container and perform operations on the elements. Iterators are lightweight objects that allow you to traverse the elements of a container without knowing its underlying implementation.

- **ForwardIterator**: Can traverse the elements of a container and do more operations like incrementing.
- **BidirectionalIterator**: Can traverse in both directions and do operations like decrementing.
- **RandomAccessIterator**: Can access elements by index and do operations like assigning values.

```cpp
RandomAccessIterator  
```

More Iterators

- **ForwardIterator**: Can traverse the elements of a container and do more operations like incrementing.
- **BidirectionalIterator**: Can traverse in both directions and do operations like decrementing.
- **RandomAccessIterator**: Can access elements by index and do operations like assigning values.

Discussions

If you define or use an iterator, you must specify what kind of container it will be used for. For example, if you define a list containing elements of type 'int',

```cpp
int list[] = {1, 2, 3, 4, 5};  
```

Then you can define an iterator to traverse this list:

```cpp
int iter = list;  
```

When you define or use an iterator, you must specify what kind of container it will be used for. For example, if you’ve defined a list containing elements of type 'int',

```cpp
int list[] = {1, 2, 3, 4, 5};  
```

Then you can define an iterator to traverse this list:

```cpp
int iter = list;  
```
Container Adaptor

It's possible to use basic containers to create another kind of container called a container adaptor. A container adaptor is a sort of simplified or conceptual container that emphasizes certain aspects of a more basic container: it provides a different interface to the programmer.

The adaptors implemented in the STL are stacks, queues, and priority queues. A stack restricts access to pushing and popping a data item on and off the top of the stack. In a queue, you push items at one end and pop them off the other end. In a priority queue, you push data on the front in accordance with the priority you assign to it. When you pop the data off the other end, you always pop the largest item stored. The priority queue automatically sorts the data for you.

Adaptors are template classes that translate functions used in the new container (such as push and pop) to functions used by the underlying container. Stacks, queues, and priority queues can be created from different sequence containers, although the adaptor is often the most obvious choice.

You use a template with a template to instantiate a new container. For example, here's a stack object that holds type int, instantiated from the deque a basic stack:

```cpp
stack<int, std::deque<int>> s; // s is a stack
```

By default, a STL stack adopts a deque. So you can define a stack as follows:

```cpp
stack<int> s; // s is a stack
```

We could force a stack to adopt a vector with the definition:

```cpp
stack<int, std::vector<int>> s; // s is a stack
```