




Copyright © 2011 IGATE Corporation (a part of Capgemini Group). All rights reserved.
No part of this publication shall be reproduced in any way, including but not limited to photocopy, photographic, magnetic, or other record, without the prior written permission of IGATE Corporation (a part of Capgemini Group).
IGATE Corporation (a part of Capgemini Group) considers information included in this document to be confidential and proprietary.

Document History				
Date	Course Version No.	Software Version No.	Developer / SME	Change Record Remarks
Mar-2009	1.0		Vaishali Kunchur	Content Creation
28-May-2009	1.0		CLS team	Review

January 19, 2016

Proprietary and Confidential

< 3 >

Capgemini
INNOVATING. TRANSFORMING. SUSTAINING.


Course Goals and Non Goals

➤ **Course Goals**

- Understand Standard Template Library using C++
- Understand containers, iterators, and algorithms

➤ **Course Non Goals**


- Templates will not be covered as a part of this course



January 19, 2016

Proprietary and Confidential

< 3 >



Capgemini

ENSAUING THE WORLD OF SOFTWARE


Pre-requisites

- > C++
- > Templates in C++

January 19, 2016

Proprietary and Confidential

< 4 >




Capgemini

ENSAUVE TROUVER LE BON CONSEILLER

Intended Audience

> Programmers




© Copyright Capgemini, Inc.
All Rights Reserved

January 19, 2016

Proprietary and Confidential

< 5 >



Capgemini
ENSAUING THE WORLD OF BUSINESS


Day Wise Schedule

- **Day 1**
 - Lesson 1: Introduction to STL and Containers
- **Day 2**
 - Lesson 2: Iterators and Algorithms

January 19, 2016

Proprietary and Confidential

» 6 «



Capgemini
INNOVATIVE TECHNOLOGY SOLUTIONS

Table of Contents

1.

Lesson 1: Introduction to STL and Containers

1.1. Introduction to STL

1.2. Containers

1.3. Types of Containers

1.4. Sequence Containers – list and vectors


1.5. Associative Containers – set and map

1.6. Container Adapters – queue and stack

January 19, 2016

Proprietary and Confidential

< 3 >

Capgemini

ENSAUTING THE WORLD OF BUSINESS

Table of Contents

2.

Lesson 2: Iterators and Algorithms


2.1. Types of Iterators

2.2. Algorithms

January 19, 2016

Proprietary and Confidential

+ 8 +

Capgemini

ENSAUING TECHNOLOGICAL SOLUTIONS

References


➤ Books:

– Thinking in C++; by Bruce Eckel

➤ URLs:

– http://www.sgi.com/tech/stl/table_of_contents.html


– msdn.microsoft.com/en-us/magazine/cc301955.aspx



January 19, 2006

Proprietary and Confidential


< 9 >

 **Capgemini**
ENSAUTUNE TOWNSHED SPOTSWOOD

Page 0-9

Next Step Courses (if applicable)


➤ STL Extensions



January 19, 2016

Proprietary and Confidential

> 10 <



Capgemini

ENSAUING TECHNOLOGY SOLUTIONS


C++ STL (Standard Template Library)

Lesson 1: Introduction to STL and Containers

January 19, 2018

Proprietary and Confidential

> 1 <



Capgemini
CONSULTING | TECHNOLOGY | BUSINESS

Lesson Objectives

➤ To understand the following topics:


- Introduction to STL
- Containers
- Types of Containers:
 - Sequence Containers – List and Vectors
 - Associative Containers – Set and Map
 - Container Adapters – Queue and Stack



1.1: Introduction to STL

Practical Limitations of C++

- **Limitation of C++:**
 - Lack of a common set of generic data structures.
 - Lack of sufficient language mechanism to enable developers to implement container classes that were type safe, flexible, and efficient for general use.
- **Templates were incorporated into C++ language.**
- **Standard Template Library (STL) provides C++ programmers with a library of common data structures.**

January 19, 2006
Proprietary and Confidential
3


Introduction to STL:

Practical limitations of C++:

Limitation of C++ has been the lack of common set of generic data structures like “containers” or “collections” for using in programs.

C++ programmers have to create their own data structures whenever they need them. However, this undertaking is tedious and error prone.

This was because, until recently C++ lacked sufficient language mechanisms to enable developers to implement “container classes” that were type safe, flexible, and efficient for general use.

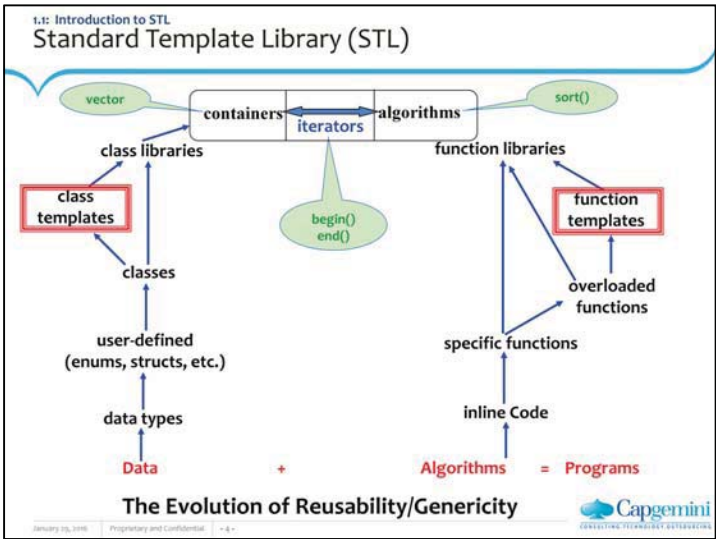
Hence Stroustrup and others from Extension Working Group (EWG) came up with “templates for the C++ language”. However, the approach used prior to templates could not provide the required efficiency for the components in a standard library that would be put to general use.

Standard Template Library (STL) provides C++ programmers with:

a library of “common data structures” – for example: linked lists, vectors, dequeues, sets, and maps

a “set of fundamental algorithms” that operate on them


Now, C++ programmers no longer need to implement these basic data structures and can be assured that the data structures provided by the STL perform efficiently and correctly.



STL Overview

- STL is based on the use of C++ templates.
- Class and function templates are used throughout the STL.
- Templates allow the STL to work with built-in types and user-defined types in a seamless way.
- Three core components of STL are:
 - Containers
 - Algorithms
 - Iterators

January 19, 2006 Proprietary and Confidential 5-1



Introduction to STL (contd.):

STL Overview:

The STL is based on the use of C++ templates; class and function templates are used throughout the STL. Templates provide the efficiency that is required for a generic component library, and also make the library extensible. Templates allow the STL to work with built-in types and user-defined types in a seamless way. Templates are still a recent addition to C++. The STL depends heavily on many of these advanced features, and in some cases, relies on workarounds to accommodate the current generation of compilers.

There are six components in the STL organization. Three components, in particular, can be considered the core components of the library, namely:

Containers: They are data structures that manage a set of memory locations.

Algorithms: They are computational procedures.

Iterators: They provide a mechanism for traversing and examining the elements in a container.

An “STL data structure” or “container”, unlike traditional ones, does not contain many member functions. STL containers contain:

a minimal set of operations for creating, copying, and destroying the container operations for adding and removing elements


You will not find container member functions for examining the “elements” in a container, or sorting them. Instead, algorithms have been decoupled from the container, and can only interact with a container via traversal by an “iterator”.

Advantages of STL

➤ Following are the advantages of STL:

- Flexibility
 - STL allows algorithms to be applied to many different structures
- Efficiency
 - Containers are close to efficiency of hand-coded, type-specific containers.
- Easy to learn structure
 - Library is small due to high degree of generality
- Theoretical foundation

January 19, 2006 Proprietary and Confidential > 8 <



Introduction to STL (contd.):

Advantages of STL:

Following is a list of the advantages provided by STL:

Flexibility

The use of generic algorithms allows algorithms to be applied to many different structures.

STL's generic algorithms work on native C++ data structures such as strings and vectors.

Efficiency

STL containers are very close to the efficiency of hand-coded, type-specific containers.

Easy-to-learn structure

The library is quite small owing to the high degree of generality.


Theoretical foundation

The library bases its theoretical foundation on a "semi-formal" specification of the library components.

C++ Standard Template Library

- C++ STL is a generic collection of class templates and algorithms.
- Standard data structures can be easily implemented.
- C++ Standard Template Library includes:
 - STL Containers
 - Iterators
 - Algorithms
 - Functions objects

January 19, 2016 Proprietary and Confidential > 2 <



Introduction to STL (contd.):

C++ Standard Template Library:

The Standard Template Library (STL) sets standards for the application of “iterators” applied to STL containers or other sequences defined by us, by STL algorithms or other user defined functions.

The STL provides a set of common classes for C++, such as “containers” and “associative arrays”. These can be used with any “built-in type” and with any “user-defined type” that supports some elementary operations.

STL algorithms are independent of containers, which significantly reduces the complexity of the library.

The STL uses templates to achieve its results. This approach provides “compile-time polymorphism”, which is more efficient than traditional “run-time polymorphism”.

Modern C++ compilers are tuned to minimize any “abstraction penalty” arising from heavy use of the STL.

The STL was created as the library of “generic algorithms” and “data structures” for C++, with three ideas in mind:

- generic programming
- abstractness without loss of efficiency, and
- value semantics

Algorithms in STL are like function templates.

1.1: Introduction to STL

Standard Template Library (STL)

➤ A library of class and function templates based on work in generic programming done by Alex Stepanov and Meng Lee of the Hewlett Packard Laboratories in the early 1990s. It has three components:

1. **Containers:** Generic "off-the-shelf" class templates for storing collections of data
2. **Algorithms:** Generic "off-the-shelf" function templates for operating on containers
3. **Iterators:** Generalized "smart" pointers that allow algorithms to operate on almost any container

The diagram illustrates the interaction between Container Classes and Algorithms. On the left, a blue starburst shape contains the text "Container Classes" and "vector". On the right, another blue starburst shape contains the text "Algorithms" and "sort()". A double-headed blue arrow connects the two starbursts, with the word "Iterators" written above it. Below the arrow, the functions "begin()" and "end()" are listed in red text, indicating the range of elements that iterators traverse.

January 26, 2016 Proprietary and Confidential v. 8.0

Capgemini
CONSTRUCTIVE TECHNOLOGY SOLUTIONS

1.2: Containers


Concept

- Container is an object that can keep and administer other objects.
- They are implemented as class templates.
- They allow a great flexibility in the types supported as element.
- Containers manage the storage space for its elements and provides member functions to access them.

January 19, 2016

Proprietary and Confidential

> 9 <


CONSTRUCTING TECHNOLOGY DIFFERENTIAL

Containers:

A container is a “holder object” that stores a collection of other objects (its “elements”). Containers are implemented as “class templates”, which allow a great flexibility in the types supported as elements.

The container manages the “storage space” for its “elements”, and provides member functions to access them, either directly or through iterators (reference objects with similar properties to pointers).

Containers replicate the structures that are very commonly used in programming.

For example: dynamic arrays (vector), queues (queue), stacks (stack), heaps (priority_queue), linked lists (list), trees (set), associative arrays (map).

Many containers have several member functions in common, and share functionalities, as well.

The decision of the type of container that has to be used for a specific need does not generally depend only on the “functionality” offered by the container, but also on the “efficiency of some of its members (complexity)”.

This is especially true for “sequence containers”, which offer different trade-offs in complexity between inserting / removing elements and accessing them. stack, queue and priority_queue are implemented as “container adaptors”. “Container adaptors” are not full container classes, but classes that provide a specific interface relying on an object of one of the container classes (such as “deque” or “list”) to handle the elements.

The underlying container is encapsulated in such a way that its elements are accessed by the members of the container class independently of the underlying container class used.

1.1: Introduction to STL

STL Containers

➤ In 1994, STL was adopted as a standard part of C++.


➤ There are 10 containers in STL:

Kind of Container	STL Containers
Sequential:	deque, list, vector
Associative:	map, multimap, multiset, set
Adapters:	priority_queue, queue, stack
Non-STL:	bitset, valarray, string

January 19, 2016

Proprietary and Confidential

10

Capgemini
CONSTRUCTING TECHNOLOGY SOLUTIONS

Containers (contd.):
The Standard Template Library (STL) provides several containers for storing collections of related objects. The containers are all template classes, allowing you to specify what objects are allowed in the containers.
Containers in STL can be divided into three categories:


- sequence containers
- associative containers, and
- container adapters

1.2: Containers > 1.2.1: Sequence Containers

Concept

- **Sequence containers:**
 - Sequence containers maintain the original ordering of inserted elements.
 - They allow us to specify the location for inserting the element in the container.
- **The different types of Sequence containers are:**
 - Deque
 - List
 - Vector

January 19, 2016 Proprietary and Confidential 11



Sequence Containers:

Sequence containers maintain the original ordering of inserted elements. This allows you to specify where to insert the element in the container.

Types of Sequence containers:

The different types of Sequence containers are:

The deque (double-ended queue) container allows for fast insertions and deletions at the beginning and end of the container. You can also randomly access any element quickly.

The list container allows for fast insertions and deletions anywhere in the container, but you cannot randomly access an element in the container.

The vector container behaves like an array, but will automatically grow as required.

Vector Container

- **Generalized array that stores a collection of elements of the same data type**
- **Vector – similar to an array**
 - Vectors allow access to its elements by using an index in the range from 0 to n-1 where n is the size of the vector
- **Vector vs array**
 - Vector has operations that allow the collection to grow and contract dynamically at the rear of the sequence

Vector Container

Example:

```
#include <vector>
```

```
...
```

```
vector<int> scores(100);           //100 integer scores
```

```
vector<Passenger>passengerList(20); //list of 20 passengers
```

➤ Constructors:

```
vector<T> v,           // empty vector
```

```
v1(100),              // contains 100 elements of type T
```

```
v2(100, val),          // contains 100 copies of val
```

```
v3(fp1r,lp1r);         // contains copies of elements in  
                        // memory locations fp1r up to lp1r
```

➤ Copy constructor

➤ Destructor

Vector Container - operations

- `v.capacity()` Number of elements `v` can contain without growing
- `v.max_size()` Upper limit on the size and capacity
- `v.size()` Number of elements `v` actually contains
- `v.reserve(n)` Increase capacity (but not size) to `n`
- `v.empty()` Check if `v` is empty
- `v.push_back(val)` Add `val` at end
- `v.pop_back()` Remove value at end
- `v.front()`, `v.back()`, Access first value, last value,
 `v[i]`, `v.at(i)` `i`-th value without / with range checking
- Relational operators Lexicographic order is used
- Assignment (`=`) e.g., `v1 = v2;`
- `v.swap(v1)` Swap contents with those of vector `v1`
-

Vector Container

- Allows direct access to the elements via an index operator
- Indices for the vector elements are in the range from 0 to size()-1
- Example:

```
#include <vector>
vector<int> v(20);
v[5]=15;
```

STL's stack container

- STL includes a stack container.
- Actually, it is an **adapter**, as indicated by the fact that one of its type parameters is a **container type**.
- Sample declaration:
`stack<int, vector<int> > st;`
- Basically, it is a class that acts as a wrapper around another class, providing a **new user interface** for that class.
- A container adapter such as **stack** uses the members of the encapsulated container to implement what looks like a new container.
- For a `stack<T, C<T> >`, `C<T>` may be any container that supports `push_back()` and `pop_back()` in a LIFO manner.

In particular `C` may be a **vector**, a **deque**, or a **list**.

Basic Operations

Constructor

`stack< T, C<T> > st;` creates an empty stack `st` of elements of type `T`; it uses a container `C<T>` to store the elements.

Note 1: The space between the two `>`s must be there to avoid confusing the compiler (else it treats it as `>>`); for example,
`stack< int, vector<int> > s;`
not `stack< int, vector<int>> s;`

Note 2: The default container is `deque`; that is, if `C<T>` is omitted as in `stack<T> st;` a `deque<T>` will be used to store the stack elements. Thus `stack<T> st;` is equivalent to `stack< T, deque<T> > st;`

Destructor

Assignment, relational Operators

`size()`, `empty()`, `top()`, `push()`, `pop()`

Example: Converting to base two (where our whole discussion of stack began).

January 19, 2008 Proprietary and Confidential ~ 17 ~



Example: convert base 10 to base 2

```
#include <iostream>
// #include <deque> not needed for default container,
// but do need if some other container is used
#include <stack>
using namespace std;

int main()
{
    unsigned number,          // number to be converted
        remainder;           // remainder of number/2
    stack<unsigned> stackOfRemainders;
    char response;             // user response
```

Example: convert base 10 to base 2

```
do{
    cout << "Enter positive integer to convert: ";
    cin >> number;
    while (number != 0)
    {
        remainder = number % 2;
        stackOfRemainders.push(remainder);
        number /= 2;
    }
    cout << "Base two representation: ";
    while (!stackOfRemainders.empty() )
    {
        remainder = stackOfRemainders.top();
        stackOfRemainders.pop();
        cout << remainder;
    }
    cout << endl << "\nMore (Y or N)? ";
    cin >> response;
} while (response == 'Y' || response == 'y');
```

January 19, 2006 Proprietary and Confidential - 19 -



STL's queue container

In `queue<T, C<T>>`, container type `C` may be `list` or `deque`.

Why not `vector`? You can't remove from the front efficiently!

The default container is `deque`.

`queue` has same member functions and operations as `stack` except:

- `push()` adds item at back (our `addQ()`)
- `front()` (instead of `top()`) retrieves front item
- `pop()` removes front item (our `removeQ()`)
- `back()` retrieves rear item

queue Example

```
#include <string>
#include <queue>
using namespace std;

int main()
{
    queue<int>    qint;
    queue<string> qstr;

    // Output number of values stored in qint
    cout << qint.size() << endl;

    for (int i = 1; i <= 4; i++)
        qint.push(2*i);

    qint.push(123);

    cout << qint.size() << endl;
```

Contd

```
while (!qint.empty())// Dump contents of qint
{
    cout << qint.front() << " ";
    qint.pop();
}
cout << endl;

qstr.push("STL is"); qstr.push("impressive!\n");
while (!qstr.empty())
{
    cout << qstr.front() << ' ';
    qstr.pop();
}
}
```

Output:

```
0
5
2 4 6 8 123
STL is impressive!
```


Deque

As an ADT, a **deque** — an abbreviation for **double-ended queue** — is a sequential container that functions like a queue (or a stack) on both ends.

It is an ordered collection of data items with the property that **items can be added and removed only at the ends**.

Basic operations are:

Construct a deque (usually empty):

Check if the deque is empty

Push_front: Add an element at the front of the deque

Push_back: Add an element at the back of the deque

Front: Retrieve the element at the front of the deque

Back: Retrieve the element at the back of the deque

Pop_front: Remove the element at the front of the deque

Pop_back: Remove the element at the back of the deque

STL's deque Class Template

Has the same operations as **vector<T>** except that there is no **capacity()** and no **reserve()**

Has two new operations:

d.push_front(value); Push copy of **value** at front of **d**

d.pop_front(value); Remove **value** at the front of **d**

Like STL's **vector**, it has several operations that are not defined for deque as an ADT:

[]
insert and delete at arbitrary points in the list,
same kind of iterators.

But insertion and deletion are not efficient and, in fact, take longer than for **vectors**.

vector vs. deque

Capacity of a **vector** must be increased

⇒ it must copy the objects from the old **vector** to the new **vector**

⇒ it must destroy each object in the old **vector**

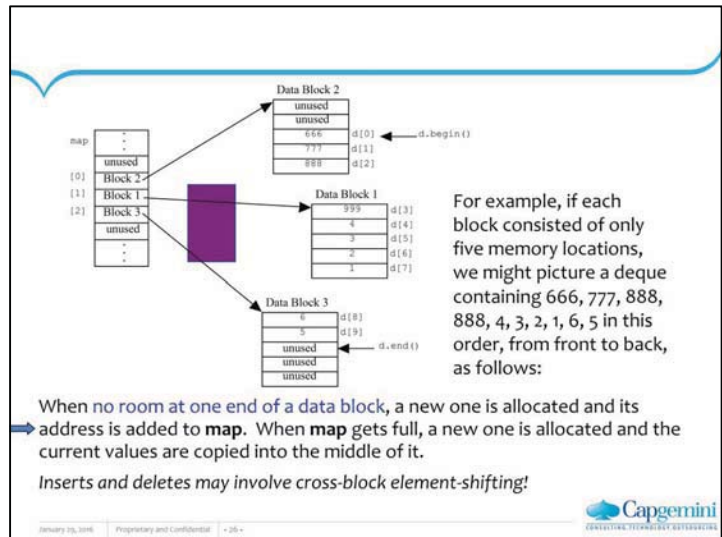
⇒ a lot of overhead!

With **deque** this copying, creating, and destroying is avoided.

Once an object is constructed, it can **stay in the same memory locations as long as it exists** (if insertions and deletions take place at the ends of the **deque**).

Unlike **vectors**, a **deque** isn't stored in a single varying-sized block of memory, but rather in a collection of fixed-size blocks (typically, 4K bytes).

One of its data members is essentially an array **map** whose elements point to the locations of these blocks.



1.2: Containers > 1.2.1: Sequence Containers > 1.2.1.1: Types > 1.2.1.1.2: list Class


Explanation

➤ **list Class:**

- list class is a template class of sequence containers.
- It maintains the elements in a linear order and allows efficient insertions and deletions at any location within the sequence.
- The sequence is stored as a bidirectional linked list of elements.
- Each member is of some type *Type*.

```
template
< class Type, class Allocator=allocator<Type> > class list
```

January 19, 2016 Proprietary and Confidential 32


list Class:

The STL list class is a “template class” of “sequence containers” that maintain their elements in a linear arrangement. They allow efficient insertions and deletions at any location within the sequence. The sequence is stored as a “bidirectional linked list” of elements, each containing a member of some type *Type*.

```
template < class Type, class Allocator=allocator<Type> > class
list
```

Here the parameters are:

Type – It is the element data type to be stored in the list.

Allocator – It is the type that represents the stored allocator object that encapsulates details about the list’s allocation and deallocation of memory. This argument is optional, and the default value is `allocator<Type>`

The list member functions, namely `merge`, `reverse`, `unique`, `remove`, and `remove_if`, have been optimized for operation on list objects. They offer a high-performance alternative to their generic counterparts.

List reallocation occurs when a “member function” must insert or erase elements of the list. In all such cases, only “iterators” or “references” that point at erased portions of the controlled sequence become invalid.

Include the STL standard header `<list>` to define the container template class `list` and several supporting templates.

list Class - Member Functions (Modifiers)

Summary of list class member functions:

- assign Assign new content to container
- push_front Insert element at beginning
- pop_front Delete first element
- push_back Add element at the end
- pop_back Delete last element
- insert Insert elements
- erase Erase elements

January 19, 2016 Proprietary and Confidential - 28 -



list Class – Member Function (Modifiers):

This provides the following advantages to list containers:

Efficient insertion and removal of elements anywhere in the container (constant time).

Efficient moving of elements and block of elements within the container or even

between different containers (constant time).

Iterating over the elements in forward or reverse order (linear time).

Member functions:

(constructor) Construct list - Example

```
list<int> first; // empty list of ints
```

```
list<int> second (4,100); // four ints with value 100
```

```
list<int> third (second.begin(),second.end()); // iterating through second
```

```
list<int> fourth (third); // a copy of third (destructor)
```

List destructor

```
operator=                      Copy container content
```

Iterators:

```
begin                      Return iterator to beginning
```

```
End                      Return iterator to end
```

```
rbegin                      Return reverse iterator to reverse beginning
```


```
rend                      Return reverse iterator to reverse end
```

list Class - Member Functions (Modifiers)

➤ **Summary of list class member functions (contd.):**

- swap Swap content
- clear Clear content

January 19, 2016 Proprietary and Confidential 29



list Class – Member Function (Modifiers) (contd.):

Capacity:

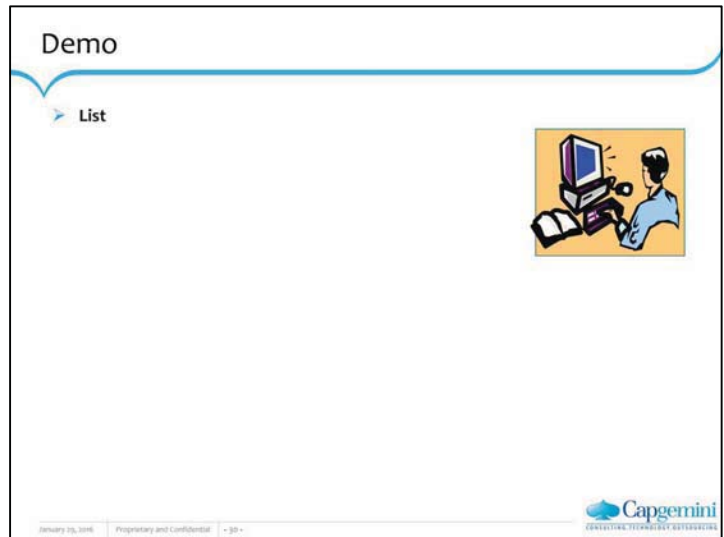
empty	Test whether container is empty
size	Return size
max_size	Return maximum size
resize	Change size

Element access:

front	Access first element
back	Access last element

Operations:

splice	Move elements from list to list
remove	Remove elements with specific value
remove_if	Remove elements fulfilling condition template)
unique	Remove duplicate values
merge	Merge sorted lists
sort	Sort elements in container
Reverse	Reverse the order of elements



Sample code:

```
#include <iostream>
#include <list>
using namespace std;
// Simple example uses type int
int main()
{
    list<int> L;
    L.push_back(0);
    // Insert a new element at the end
    L.push_front(0);
    // Insert a new element at the beginning
    L.insert(++L.begin(),2);
    // Insert "2" before position of first argument
    // (Place before second argument)
    L.push_back(5);
    L.push_back(6);
    list<int>::iterator i;
    for(i=L.begin(); i != L.end(); ++i)
        cout << *i << " ";
    cout << endl;
    return 0;
}
```



1.2: Containers > 1.2.2: Associative Containers

Concept

➤ **Associative containers:**

- Associative containers can be grouped into two subsets, namely maps and sets.
 - **Map** – It has a key / value pair. Key is used to order the sequence, and the value is associated with that key.
 - **Set** – It is an ascending container of unique elements.
- Both “maps” and “sets” allow only “one instance” of a key or element to be inserted into the container.
- Both “maps” and “sets” support “bidirectional iterators”.

January 19, 2016
Proprietary and Confidential
31



CAPGEMINI

CONSULTING TECHNOLOGY SERVICES

Associative Containers:

The defining characteristic of “associative containers” is that elements are inserted in a pre-defined order, such as “sorted ascending”.

The associative containers can be grouped into two subsets, namely maps and sets.

Map: It is sometimes referred to as a “dictionary”, consists of a key / value pair.

The key is used to order the sequence, and the value is somehow associated with that key.

For example: A map might contain keys representing every unique word in a text and values representing the number of times that word appears in the text.

Set: It is simply an ascending container of unique elements.

Both map and set allow only one instance of a key or element to be inserted into the container.

If multiple instances of elements are required, use “multimap” or “multiset”.

Both maps and sets support “bidirectional iterators”.

While not officially part of the STL standard, hash_map, and hash_set are commonly used to improve searching times.

These containers store their elements as a “hash table”, with each table entry containing a bidirectional linked list of elements.

To ensure the fastest search times, make sure that the hashing algorithm for your elements return evenly distributed hash values.

1.2: Containers > 1.2.2: Associative Containers > 1.2.2.1: Types

Explanation

The different types of Associative Containers are:

– map

– set


– multimap

– multiset

January 19, 2016

Proprietary and Confidential

< 32 >

Capgemini
CONSULTING TECHNOLOGY SERVICES


1.2: Containers > 1.2.2: Associative Containers > 1.2.2.1: Types > 1.2.2.1.1: map Class

Explanation

➤ **map Class:**

- It is an associative container.
- It is reversible.
- It can be sorted.
- It is unique in the sense that each of its elements must have a unique key.
- It is a pair associative container.
- It is a template class.

January 19, 2016 Proprietary and Confidential 33



map Class:

The characteristics of a STL map class are:

It is an associative container, which is a variable size container that supports the efficient retrieval of “element values” based on an associated “key value”.

It is reversible, because it provides bidirectional iterators to access its elements.

It is sorted, because its elements are ordered by key values within the container in accordance with a specified comparison function.

It is unique in the sense that each of its elements must have a unique key.

It is a pair associative container, because its element data values are distinct from its key values.

It is a template class, because the functionality it provides is generic and so is independent of the specific type of data contained as “elements” or “keys”. The data types to be used for “elements” and “keys” are, instead, specified as parameters in the class template along with the “comparison function” and “allocator”.

Explanation

- It is used for the storage and retrieval of data from a collection.
- each element is a pair that has both a data value and a sort key.
- template < class Key, class Type, class Traits = less<Key>, class Allocator=allocator<pair <const Key, Type> > > class map

January 19, 2016 Proprietary and Confidential ~ 34 ~



map Class (contd.):

The STL map class is used for the storage and retrieval of data from a collection in which each element is a pair that has both a “data value” and a “sort key”.

The value of the key is unique, and is used to automatically order the data.

The value of an element in a map, but not its associated key value, may be changed directly. Instead, key values associated with old elements must be deleted and new key values associated with new elements inserted.

```
template < class Key, class Type, class Traits = less<Key>,
class Allocator=allocator<pair <const Key, Type> > > class
map
```

Here the parameters are:

Key – It is the key data type to be stored in the map.

Type – It is the element data type to be stored in the map.


Traits – It is the type that provides a function object that can compare two element values as sort keys to determine their relative order in the map. This argument is optional and the binary predicate less<Key> is the default value.

Allocator – It is the type that represents the stored allocator object that encapsulates details about the map’s allocation and deallocation of memory. This argument is optional and the default value is allocator<pair <const Key, Type> >.

map Class – Member functions

➤ Here is a summary of map Class member functions:

- Capacity:
 - empty Test whether container is empty
 - size Return container size
 - max_size Return maximum size
- Modifiers:
 - insert Insert element
 - erase Erase elements
 - swap Swap content
 - clear Clear content



January 19, 2016 Proprietary and Confidential ~ 35 ~

map Class – Member functions:

Member functions

(constructor) Construct map

map<char,int> first;

first['a']=10;

first['b']=30;

first['c']=50;

first['d']=70;

map<char,int> second (first.begin(),first.end());

map<char,int> third (second);

map<char,int,classcomp> fourth;

(destructor)

Map destructor

operator=

Copy container content

Element access:

operator[]

Access element

Modifiers:

insert

Insert element

erase

Erase elements

swap

Swap content

clear

Clear content



Sample Code:

```
struct ltstr
{
    bool operator()(const char* s1, const char* s2) const
    {
        return strcmp(s1, s2) < 0;
    }
};

int main()
{
    map<const char*, int, ltstr> months;

    months["january"] = 31;
    months["february"] = 28;
    months["march"] = 31;
    months["april"] = 30;
    months["may"] = 31;
    months["june"] = 30;
    months["july"] = 31;
```

contd.



Sample Code (contd.):

```
months["august"] = 31;
months["september"] = 30;
months["october"] = 31;
months["november"] = 30;
months["december"] = 31;

cout << "june -> " << months["june"] << endl;
map<const char*, int, ltstr>::iterator cur =
months.find("june");
map<const char*, int, ltstr>::iterator prev = cur;
map<const char*, int, ltstr>::iterator next = cur;
++next;
--prev;
cout << "Previous (in alphabetical order) is " <<
(*prev).first << endl;
cout << "Next (in alphabetical order) is " <<
(*next).first << endl;
}
```


1.2: Containers > 1.2.2: Associative Containers > 1.2.2.1: Types > 1.2.2.1.2: set Class

Explanation

set Class:

- A “set” is used for the storage and retrieval of data from a collection in which the values of the elements contained are unique.
- These values serve as the key values according to which the data is automatically ordered.
- Value of an element in a set may not be changed directly.
- A “set” is an associative container, reversible, sorted, and unique.

January 19, 2016 Proprietary and Confidential 38


set Class:

The STL container class set is used for the storage and retrieval of data from a collection in which the values of the elements contained are “unique”. These values serve as the key values according to which the data is automatically ordered. The value of an element in a set may not be changed directly. Instead, you must delete old values and insert elements with new values.

```
template < class Key, class Traits=less<Key>, class
Allocator=allocator<Key> > class set
```

The characteristics of an STL set are:

It is an associative container, which is a variable size container that supports the efficient retrieval of element values based on an associated key value. Further, it is a simple associative container because its element values are its key values. It is reversible, because it provides a “bidirectional iterator” to access its elements.

It is sorted, because its elements are ordered by key values within the container in accordance with a specified comparison function.


It is unique in the sense that each of its elements must have a unique key. Since “set” is also a “simple associative container”, its elements are also unique.

set Class – member functions

➤ Here is a summary of set Class member functions:

- Observers:
 - * key_comp Return comparison object
 - * value_comp Return comparison object
- Operations:
 - * find Get iterator to element
 - * count Count elements with a specific key
 - * lower_bound Return iterator to lower bound
 - * upper_bound Return iterator to upper bound
 - * equal_range Get range of equal elements

January 19, 2016 Proprietary and Confidential < 39 >



set Class – member functions:

operator= Copy vector content

Capacity:

empty Test whether container is empty

size Return container size

max_size Return maximum size

Modifiers:

insert Insert element

erase Erase elements

swap Swap content

clear Clear content

Set Constructor:

```
set<int> first; // empty set of ints
```

```
int myints[] = {10,20,30,40,50};
```

```
set<int> second (myints,myints+5); // pointers used as iterators
```

```
set<int> third (second); // a copy of second
```

```
set<int> fourth (second.begin(), second.end()); // iterator ctor.
```

```
set<int,classcomp> fifth; // class as Compare
```



Sample code:

```
#include <iostream>
#include <set>
#include <vector>

using namespace std;

template <class T>
void print(T& c){
    for( typename T::iterator i = c.begin(); i != c.end(); i++){
        std::cout << *i << endl;
    }
}

int main(){
    const int num_grades = 11;
    const int grade[num_grades] = { 2, 5, 3, 8, 9, 9, 6, 3, 5, 9, 10 };

    set<int> unique( grade, grade+num_grades );
    multiset<int> all( grade, grade+num_grades );
    print( unique );
    print( all );
}
```


1.2: Containers > 1.2.2: Associative Containers > 1.2.2.1: Types > 1.2.2.1.3: multiset

Explanation

multisets:

- They are similar to “set” containers, however they allow for multiple keys with equal values.
- It is a “Sorted Associative Container” that stores objects of type Key.
- Its value type and its key type, is Key.

January 19, 2016 Proprietary and Confidential ~ 61 ~



Multiple-key set:

“multisets” are associative containers with the same properties as set containers. However, they allow for multiple keys with equal values.

In their implementation in the C++ Standard Template Library, “multiset” containers take the same three template parameters as set containers:

```
template < class Key, class Compare = less<Key>, class  
Allocator = allocator<Key> > class multiset;
```

It is a “Sorted Associative Container” that stores objects of type “Key”.

It is a “Simple Associative Container”. It means that its value type, as well as its key type, is “Key”.

It is also a “Multiple Associative Container”. It means that two or more elements may be identical.

Demo

 multiset



January 19, 2016 Proprietary and Confidential 42

 Capgemini
CONSULTING TECHNOLOGY SERVICES

Sample code:

```
#include <set>
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;
void main()
{
    const int N = 10;
    int a[N] = {3, 4, 4, 1, 1, 1, 0, 5, 1, 0};
    int b[N] = {4, 4, 2, 4, 2, 4, 0, 1, 5, 5};

    multiset<int> A(a, a + N);
    multiset<int> B(b, b + N);
    multiset<int> C;
```

Explanation

January 19, 2016 Proprietary and Confidential 43



Sample code (contd.):

```
cout << "Set A: ";
copy(A.begin(), A.end(), ostream_iterator<int>(cout, " "));
cout << endl;
cout << "Set B: ";
copy(B.begin(), B.end(), ostream_iterator<int>(cout, " "));
cout << endl;

cout << "Union: ";
set_union(A.begin(), A.end(), B.begin(), B.end(),
          ostream_iterator<int>(cout, " "));
cout << endl;

cout << "Intersection: ";
set_intersection(A.begin(), A.end(), B.begin(), B.end(),
                 ostream_iterator<int>(cout, " "));
cout << endl;

set_difference(A.begin(), A.end(), B.begin(), B.end(),
               inserter(C, C.begin()));
cout << "Set C (difference of A and B): ";
copy(C.begin(), C.end(), ostream_iterator<int>(cout, " "));
cout << endl;
}
```


1.2: Containers > 1.2.2: Associative Containers > 1.2.2.1: Types > 1.2.2.1.4: Multimap

Explanation

➤ **Multimaps:**

- They are similar to maps, and store elements formed by the combination of a “key value” and a “mapped value”.
- There is no limit on the number of elements with the same key.
- Inserting a new element into a multimap does not invalidate iterators pointing to existing elements.
- Difference between set (multiset) and map (multimap):
 - set (map) stores data which inherently contains the key expression.
 - map (multimap) stores the key expression and the appropriate data separately, i.e. the key is not a part of the data stored.

January 19, 2016 Proprietary and Confidential ~ 44 ~



Multiple-key map:

Maps are a kind of associative containers that store elements formed by the combination of a key value and a mapped value, much like map containers, but allowing different elements to have the same key value.

In their implementation in the C++ Standard Template Library, multimap containers take four template parameters:

```
template < class Key, class T, class Compare = less<Key>, class
Allocator = allocator<pair<const Key,T> > > class multimap;
```

Multimap is also a “Multiple Associative Container”. It means that there is no limit on the number of elements with the same key.

Multimap has the important property that inserting a new element into a multimap does not invalidate “iterators” that point to existing elements.

Erasing an element from a multimap also does not invalidate any iterators, except for iterators that actually point to the element that is being erased.

“set” and “map” support unique keys. It means that those containers may contain at the most one element (data record) for each key.

“multiset” and “multimap” support equal keys, so more than one element can be stored for each key.

The difference between set (multiset) and map (multimap) is that a “set” (map) stores data which inherently contains the key expression. “map” (multimap) stores the key expression and the appropriate data separately, i.e. the key has not to be part of the stored data.



Sample code:

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

int main ()
{
    multimap<char,int> mymultimap;
    multimap<char,int>::reverse_iterator rit;

    mymultimap.insert (pair<char,int>('x',10));
    mymultimap.insert (pair<char,int>('y',20));
    mymultimap.insert (pair<char,int>('y',150));
    mymultimap.insert (pair<char,int>('z',9));

    // show content:
    for ( rit=mymultimap.rbegin(); rit != mymultimap.rend();
        rit++)
        cout << rit->first << " => " << rit->second << endl;

    return 0;
}
```

Lab -

➤ Lab 1

➤ Lab 2




Hands On

January 19, 2016

Proprietary and Confidential

< 45 >



Summary

- C supports development of all type of applications using one high-level language.
- Variable names are the names (labels) given to the memory location where different constants are stored.
- Expressions can contain constants, variable or function calls.
- Preprocessor directive statements begin with a # symbol.




Review Question

- Question 1: Character occupies _____ bytes.
- Question 2: _____ converts source code to object code.
- Question 3: String constants ends with _____ special character.
- Question 4: Object code is output of _____.



Review Question: Match the Following

1. File inclusion directives	Links the object codes to form a single Executable Code.
2. getchar()	Replace tokens in the current file.
3. Macro definition directives	Embed files within the current file.
4. Link Editor	Returns ASCII value range from 0 to 255




Knowledge Check

January 19, 2018

Proprietary and Confidential

< 49 >



Capgemini

CONSTRUCTIVE TECHNOLOGY PARTNERSHIP


C++ STL (Standard Template Library)

Lesson 2: Iterators and Algorithms

January 19, 2016

Proprietary and Confidential

< 1 >



Capgemini
ENJOYING TECHNOLOGY SERVICES

Lesson Objectives

➤ In this lesson, you will learn the following:

- Concept of Iterator
- Types of Iterators
- Use of Iterators
- Algorithms
- Association between container, iterator, and algorithms
- Function Objects



2.1: Iterators

Concept

➤ **Iterator is a pointer-like object, which is able to point to the specific element in the container.**

- They are objects that point to other objects.
- They are used to iterate over a range of objects.
- If an iterator points to one element in a range, then it is possible to increment it so that it points to the next element.

January 19, 2016 Proprietary and Confidential 3

**Iterators:**

- The Standard Template Library (STL) takes advantage of the “operator overloading” feature of the C++ language in order to provide class definitions for the “iterators” to represent “pointer-like objects”.
- In general, the basic operations required for pointers are as follows:
 - assigning a value (to make it point to a specific data item)
 - dereferencing it (to access the data element - in read or write mode)
 - pointer arithmetic (in particular, incrementing or decrementing the pointer with the ++ or -- unary operators), and
 - comparison of the values (in particular, for equality or inequality)
- Suppose we are given class definitions that provide the following:
 - assignment operator
 - comparison operators
 - the unary * operator, and
 - the (also unary) ++ and -- operators

Then we can indeed write code that uses these types of objects as if they were pointers (provided that each operator's definition performs an operation that corresponds to its intuitive equivalent for pointers).

2.1: Iterators > 2.1.1: Types of Iterators

Explanation

➤ The basic forms of iterators that are used in the standard library are:

- **input iterator** - read only, forward moving
- **output iterator** - write only, forward moving
- **bidirectional iterator** - read and write, forward and backward moving
- **forward iterator** - both read and write, forward moving
- **random access iterator** - read and write, random access

January 19, 2016 Proprietary and Confidential 4



Types of Iterators:

- Iterator is not a single concept, but there are “six concepts” that form a “hierarchy”. Some of these define a restricted set of operations, while the others define additional functionality.
- They are categorized into “five types” based on the algorithms that they use. They are Input Iterator, Output Iterator, Forward Iterator, Bidirectional Iterator, and Random Access Iterator. One more trivial Iterator is introduced to clarify the definitions of the other iterator concepts.
- Input and Output Iterators are most restricted iterators, both of which allow single pass.
- Let us read about the different types of iterators in detail.
- **Input Iterators** only guarantees read access. On dereferencing the Input iterator, it is possible to obtain the value it points to. It is not possible to assign a new value through Input Iterator.
Input iterators are the simplest form of iterator. They perform a “simple linear search”, looking for a specific value being held within a container. The contents of the container are described using two iterators, referred here as “first” and “last”. While “first” is not equal to “last”, the element denoted by “first” is compared to the “test value”. If equal, the iterator, which now denotes the located element, is returned. If not equal, the first iterator is incremented, and the loop then cycles once more. If the entire region of memory is examined without finding the desired value, then the algorithm returns the end-of-range iterator.
- **Output Iterators** only guarantees write access. It is possible to assign a value through Output Iterator. It is not possible to refer to that value. This means Output iterators can be used to assign values in a sequence, but cannot be used to access values.

For example: We can use an output iterator in a generic algorithm that copies values from one sequence into another.

Example code

```
void print(vector<int>&v)
{
    for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
        cout << *it << " ";
    cout << endl;
}
```

January 19, 2016 Proprietary and Confidential 15



Types of Iterator (contd.):

- Forward Iterators** support both Input Iterator and Output Iterator operations, and they also provide additional functionality.
 Multi-pass algorithm with Forward Iterators : It may be constant, in which case it is possible to access the object it points to or assign new value through it.
 It permits values to both be accessed and modified. One function that uses forward iterators is the "replace() generic algorithm", which replaces occurrences of specific values with other values.
- Bidirectional Iterators** also allows multi-pass algorithms. They support motion in both directions. A Bidirectional Iterator may be incremented to obtain the next element or decremented to obtain previous element. This iterator can be used to traverse a doubly linked list.
 It is similar to a forward iterator, except that bidirectional iterators support the decrement operator (operator --), permitting movement in either a forward or a backward direction through the elements of a container.
 For example: We can use bidirectional iterators in a function that reverses the values of a container, placing the results into a new container.
- Random access Iterator** allows operations of pointer arithmetic. It helps in addition of offsets, subscripting, and subtraction of one iterator from another to find a distance.
 Some algorithms require more functionality than the ability to access values in either a forward or backward direction. Random access iterators permit values to be accessed by subscript, subtracted one from another (to yield the number of elements between their respective values) or modified by arithmetic operations, all in a manner similar to conventional pointers.

2.1: Iterators > 2.1.2: Use of Iterators

Demo

➤ Iterator



January 19, 2016 Proprietary and Confidential > 6 <

 Capgemini
 ENJOYING THE JOURNEY OF INNOVATION

Sample code:

```
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>
#include <iterator>
using namespace std;
int main()
{
    vector<int> V;
    V.push_back(11);
    V.push_back(22);
    V.push_back(33);
    copy(V.begin(), V.end(),
    ostream_iterator<int>(cout, " "));
    cout << endl;
    list<int> L(V.size());
    reverse_copy(V.begin(), V.end(), L.begin());
    copy(L.begin(), L.end(),
    ostream_iterator<int>(cout, " "));
    cout << endl;
    vector<int> V2;
    //copy the elements from V that are not <22 into V2
    remove_copy_if(V.begin(), V.end(),
    back_inserter(V2), bind2nd(less<int>(), 22));
    copy(V2.begin(), V2.end(),
    ostream_iterator<int>(cout, " "));
    cout << endl;
    exit(0);
}
```

2.1: Iterators > 2.1.3: Stream Iterators

Explanation

- C++ Standard Library provides **iostreams** for reading and writing data to and from input / output streams.
- Two iterator templates work with I/O stream, namely:
 - **istream_iterators** - for reading data from an input stream
 - **ostream_iterators** - for writing data to an output stream

January 19, 2016 | Proprietary and Confidential | 33



Stream Iterators:

- The C++ Standard Library provides “**iostreams**” to facilitate the reading and writing of data to and from input / output streams. The STL provides two iterator templates so that algorithms may work directly with I/O streams:
 - **istream_iterators** for reading data from an input stream.
 - **ostream_iterators** for writing data to an output stream.
- For example: We can read data into our list from the standard input as follows:

```
istream_iterator<int, ptrdiff_t> in(cin);
istream_iterator<int, ptrdiff_t> eos;
copy(in, eos, back_inserter(l));
and standard output:
ostream_iterator<cout, " , " > out(cout, " , ");
copy(l.begin(), l.end(), out);
```

2.1: Iterators > 2.1.4: Iterator Adaptors

Explanation

➤ STL provides three iterator adaptors:

- **Reverse iterators** – They can be applied to a “random access iterator” or a “bi-directional iterator” that will traverse the container in the reverse direction.
- **Insert iterators** – They are of the following types:
 - **back_insert_iterators** - add new elements to the end of the container.
 - **front_insert_iterators** - add new elements to the front of a container.
 - **insert_iterators** - add new elements to the container at the location specified when the iterator is constructed.

January 19, 2016 Proprietary and Confidential - 8 -



Iterator Adaptors:

- Adaptors can also be used to extend the functionality of an existing iterator. The STL provides three iterator adaptors:
 - **Reverse iterators:** Random access iterators and bi-directional iterators can be traversed forwards and backwards. By applying the `reverse_iterator` adaptor to either a random access iterator or a bi-directional iterator an iterator is obtained that will traverse the same container in the reverse direction.
 - **Insert iterators:** Iterators behave a lot like pointers. If you assign a new value to a container through its iterator, you will overwrite any value that is already at the location specified by the iterator. Sometimes, it is desirable to be able to **insert** a new element at that location in the container. The STL provides three different **insert iterators** that instead of overwriting the value at that location, actually insert a value there. Each type of adaptor performs the insertion into the container in a different place.
 - **back_insert_iterators** add new elements to the end of the container. Since the algorithm uses the `back_insert` function to do the insertions, it can only be used with vectors, lists, and deques.
 - **front_insert_iterators** add new elements to the front of a container. Since `push_front` is used for insertions, it can only be used on lists, and deques.
 - **insert_iterators** add new elements to the container at the location specified when the iterator is constructed. This iterator can be used to insert new elements anywhere within a container, not just at the front or the end. Since it uses `insert` to add the elements, it can be used with any of the STL containers.

Explanation

- **STL provides three iterator adaptors (contd.):**
 - Raw storage iterators: Raw storage iterators allow algorithms to use raw, uninitialized memory during their execution
- **STL also provides “const iterators” so that iterators may be used with “const containers”:**

```
list::const_iterator iter;
```

January 19, 2016 | Proprietary and Confidential | 9



Iterator Adaptors (contd.):

- **Raw storage iterators:** Raw storage iterators allow algorithms to use raw, uninitialized memory during their execution. The `raw_storage_iterator` is used by several internal algorithms for partitioning and merging elements in a container.
- Example of constant iterator:

```
template <class T>
void PrintList (const list<T>& l)
{
    list::const_iterator iter;
    for (iter = l.begin(); iter != l.end(); iter++)
    {
        cout << *iter << endl;
    }
}
```

2.2: Algorithms

Explanation

- STL includes algorithms to perform most commonly used operations on group / sequence of elements like searching, sorting, insertion / removal of elements.
- Algorithms are standard methods and generic routines.
- Algorithms are decoupled from particular containers and are instead parameterized by the type of iterator.
- Algorithms are written to work on iterators rather than components.

Algorithms:

- The STL also includes a large collection of *algorithms* that manipulate the data stored in containers. You can reverse the order of elements in a vector, for example, by using the “reverse algorithm”. They are routines to find, count, sort, search elements in container classes.
- Algorithms define a collection of functions, especially designed to be used on ranges of elements.
- STL algorithms are decoupled from the particular containers they operate on, and are instead parameterized by iterator types. This means that all containers within the same iterator category can utilize the same algorithms.
- Since algorithms are written to work on “iterators” rather than “components”, the software development effort is drastically reduced. Instead of writing a “search routine” for each kind of container, one needs to only write one for each iterator of interest. Since different components can be accessed by the same iterators, just a few versions of the search routine must be implemented.

Algorithms (contd.):

- For example, here is the implementation of the STL count algorithm, which counts the number of elements in a container with a particular value.

```
template <class InputIterator,
          class T,
          class Size>
void count(InputIterator first,
          InputIterator last,
          const T& value, Size& n) {
    while (first != last)
        if (*first++ == value)
            ++n;
}
```

- An “algorithm” which uses “InputIterators” can be used with:
 - any “container” that works with “InputIterators”, or
 - any of the iterator categories above it in the hierarchy
- To count the number of elements equal to 10 in our list, we can write:

```
int countTen = 0;
count(l.begin(), l.end(), 10, countTen);
```

- The semantics of the InputIterator first and InputIterator last in the STL count algorithm look like regular C pointers. In fact, all STL algorithms will work with regular C/C++ pointers, as well.
- For example, we can use the same “count algorithm” to count values in a regular int array:

```
int a[100]; // put some values into array a
int countTen = 0;
count(&a[0], &a[100], 10, countTen);
```

- Most C/C++ programmers are so familiar with the “pointer programming paradigm” that they quickly become familiar with the “STL algorithms”.
- The pointer-like semantics of STL algorithms guarantee that there is an efficient implementation for them, often resulting in code that is nearly as efficient as hand-written assembly code.

2.2: Algorithms > 2.2.1: Fundamental categories of Algorithm

Explanation

➤ Given below is a list of categories of algorithm:

- **Non-Mutating Sequence Operations:** count and search
- **Mutating Sequence Operations:** copy, reverse, or swap
- **Searching and Sorting:** stable_sort, binary_search, and merge
- **Set Operations:** set_union and set_intersection
- **Heap Operations**
- **Numeric Operations:** accumulate, inner_product, partial_sum, and adjacent_difference

January 19, 2016 Proprietary and Confidential 12



Fundamental Categories of Algorithm:

Given below is a list of categories of algorithm:

- **Non-Mutating Sequence Operations:** They consist of algorithms like count and search, which do not modify the iterator or its associated container.
- **Mutating Sequence Operations:** They consist of algorithms like copy, reverse, or swap which may modify a container or its iterator.
- **Searching and Sorting:** They consist of sorting, searching, and merging algorithms, such as stable_sort, binary_search, and merge.
- **Set Operations:** They consist of mathematical set operations, such as set_union and set_intersection. These algorithms only work on sorted containers.
- **Heap Operations:** Heaps are a very useful and efficient data structure that is often used to implement priority queues. The STL provides facilities for making heaps and using them.
- **Numeric Operations:** The STL provides a few numerical routines to show how the STL might be used to provide a template-based numeric library. The STL provides algorithms for: accumulate, inner_product, partial_sum, and adjacent_difference.
- **Miscellaneous Operations:** This final category is for algorithms like min and next_permutation that don't quite fit in the above categories.

STL algorithms

- do not access containers directly
- stand-alone functions that operate on data by means of *iterators*
- can work with regular C-style arrays as well as containers.

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

// Add our Display() template for arrays

int main()
{
    int ints[] = {555, 33, 444, 22, 222, 777, 1, 66};
    // must supply start and "past-the-end" pointers
    sort(ints, ints + 8);
    cout << "Sorted list of integers:\n";
    Display(ints, 8);
}
```



```
double dubs[] = {55.5, 3.3, 44.4, 2.2, 22.2, 77.7, 0.1};
sort(dubs, dubs + 7);
cout << "\nSorted list of doubles:\n";
Display(Dubs, 7);

string strs[] = {"good", "morning", "cpsc", "186", "class"};
sort(strs, strs + 5);
cout << "\nSorted list of strings:\n";
Display(strs, 5);
}
//--- OUTPUT -----
Sorted list of integers:
1 22 33 66 222 444 555 777

Sorted list of doubles:
0.1 2.2 3.3 22.2 44.4 55.5 77.7

Sorted list of strings:
186 class cpsc good morning
```

Supply own comparison operator

```
#include <iostream.h>
#include <string>
#include <algorithm>

bool IntLessThan(int a, int b)
{ return a > b; }

bool DubLessThan(double a, double b)
{ return a > b; }

bool StrLessThan(string a, string b)
{ return !(a < b) && !(a == b); }
```

```
int main()
{
    int ints[] = {555, 33, 444, 22, 222, 777, 1, 66};
    sort(ints, ints + 8, IntLessThan);
    cout << "Sorted list of integers:\n";
    Display(ints, 8);

    double dubs[] = {55.5, 3.3, 44.4, 2.2, 22.2, 77.7, 0.1};
    sort(dubs, dubs + 7, DubLessThan);
    cout << "Sorted list of doubles:\n";
    Display(dubs, 7);

    string strs[] =
    {"good", "morning", "cpsc", "186", "class"};
    sort(strs, strs + 5, StrLessThan);
    cout << "Sorted list of strings:\n";
    Display(strs, 5);
}

//-----
Sorted list of integers:
777 555 444 222 66 33 22 1
Sorted list of doubles:
77.7 55.5 44.4 22.2 3.3 2.2 0.1
Sorted list of strings:
morning good cpsc class 186
```

Bitsets and ValArrays

The C++ standard includes **bitset** as a container, but it is not in STL. A **bitset** is an array whose elements are bits. It is much like an array whose elements are of type **bool**, but unlike arrays, it does provide operations for manipulating the bits stored in it. They provide an excellent data structure to use to implement sets.

The standard C++ library also provides the **valarray** class template, which is designed to carry out (mathematical) vector operations very efficiently. That is, **valarrays** are (mathematical) vectors that have been highly optimized for numeric computations.

STL's algorithms

Another major part of STL is its collection of more than 80 generic **algorithms**. They are not member functions of STL's container classes and do not access containers directly. Rather they are stand-alone functions that operate on data by means of **iterators**. This makes it possible to work with regular C-style arrays as well as containers. We illustrate one of these algorithms here: **sort**.

Sort 1: Using <

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

// Add a Display() template for arrays

```
int main()
{
    int ints[] = {555, 33, 444, 22, 222, 777, 1, 66};
    // must supply start and "past-the-end" pointers
    sort(ints, ints + 8);
    cout << "Sorted list of integers:\n";
    Display(ints, 8);
}
```

```
template <typename ElemType>
void Display(ElemType arr, int n)
{
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}
```

January 15, 2018 Proprietary and Confidential v1.0

```
double dubs[] = {55.5, 3.3, 44.4, 2.2, 22.2, 77.7, 0.1};
sort(dubs, dubs + 7);
cout << "\nSorted list of doubles:\n";
Display(Dubs, 7);

string strs[] = {"good", "morning", "cpsc", "186", "class"};
sort(strs, strs + 5);
cout << "\nSorted list of strings:\n";
Display(strs, 5);
}

Output:
Sorted list of integers:
1 22 33 66 222 444 555 777

Sorted list of doubles:
0.1 2.2 3.3 22.2 44.4 55.5 77.7

Sorted list of strings:
186 class cpsc good morning
```

Sort 2: Supplying a "less-than" function to use in comparing elements

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

bool IntLessThan(int a, int b)
{ return a > b; }

bool DubLessThan(double a, double b)
{ return a > b; }

bool StrLessThan(string a, string b)
{ return !(a < b) && !(a == b); }
```

```
int main()
{ int ints[] = {555, 33, 444, 22, 222, 777, 1, 66};
  sort(ints, ints + 8, intLessThan);
  cout << "Sorted list of integers:\n";
  Display(ints, 8);

  double dubs[] = {55.5, 3.3, 44.4, 2.2, 22.2, 77.7, 0.1};
  sort(dubs, dubs + 7, DubLessThan);
  cout << "Sorted list of doubles:\n";
  Display(dubs, 7);

  string strs[] = {"good", "morning", "cpsc", "186", "class"};
  sort(strs, strs + 5, StrLessThan);
  cout << "Sorted list of strings:\n";
  Display(strs, 5);
}
Output:
Sorted list of integers:
777 555 444 222 66 33 22 1
Sorted list of doubles:
77.7 55.5 44.4 22.2 3.3 2.2 0.1
Sorted list of strings:
morning good cpsc class 186
```


Sort 3: Sorting a vector of stacks using < (defined for stacks)

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
#include "Stack.h"

/* Add operator<() to our Stack class template as a member
function with one Stack operand or as a friend function with
two Stacks as operands.
Or because of how we're defining < for Stacks here,
st1 < st2 if top of st1 < top of st2
we can use the top() access function and make operator<()
an ordinary function */

template <typename StackElement>
bool operator<(const Stack<StackElement> & a,
              const Stack<StackElement> & b)
{ return a.top() < b.top(); }
```

```
int main()
{
    vector< Stack<int> > st(4); // vector of 4 stacks of ints

    st[0].push(10); st[0].push(20);
    st[1].push(30);
    st[2].push(50); st[2].push(60);
    st[3].push(1); st[3].push(999); st[3].push(3);

    sort(st.begin(), st.end());
    for (int i = 0; i < 4; i++)
    {
        cout << "Stack " << i << ": \n";
        st[i].display();
        cout << endl;
    }
}
```

Output

Stack 0:

3

999

1

Stack 1:

20

10

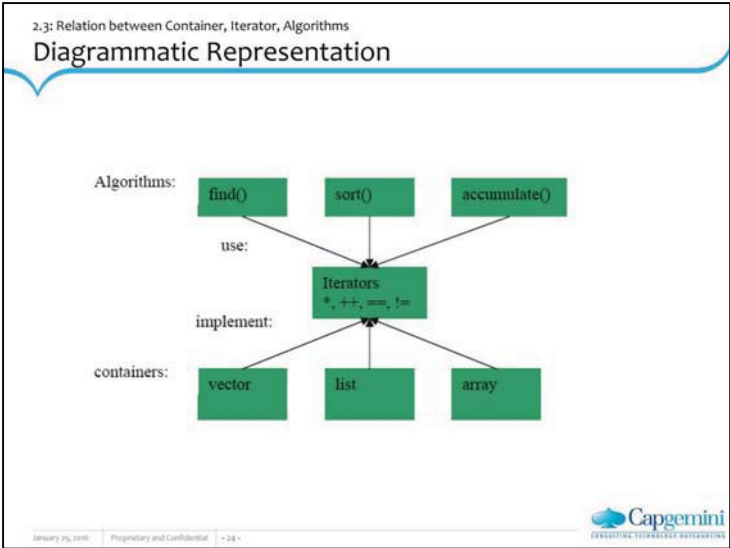
Stack 2:

30

Stack 3:

60

50



2.3: Relation between Container, Iterator, Algorithms > 2.3.1: Function Objects

Concept

- **Function Objects are the class objects that have a function call operator (operator()) defined.**
 - They contain no data members or constructors / destructors.
 - Function objects can be passed to template functions in the same way as pointers to functions are passed to C functions.
 - However, as there is no pointer indirection, and as they can often be expanded inline, they are more efficient.

January 19, 2016 Proprietary and Confidential - 25 -



Function Objects:

- In C programming, you can make a “routine” more generic in nature by passing a “pointer” to a “helper function” that can be used by the “routine” as a part of its algorithm. This is the manner in which the C library quicksort algorithm, qsort, is implemented.
- To sort an array of integers,
 - we define a comparison function

```
static int intcmp(int* i,int *j){
    return(*i - *j);
}
```

- pass a pointer to this function when calling qsort:


```
qsort(a,10,sizeof(int),intcmp);
```
- The problem with passing pointers to functions is that they are less efficient than template functions that can be expanded inline.
- **Function objects** are class objects which have a function call operator (operator()) defined. They contain no data members or constructors / destructors (except the default ones provided by the compiler). Function objects can be passed to template functions in much the same way as pointers to functions are passed to C functions. However, because there is no pointer indirection and because they can often be expanded inline, they are much more efficient.

2.3: Relation between Container, Iterator, Algorithms > 2.3.2: Use of Function Objects

Demo

➤ Function Object



January 19, 2016 Proprietary and Confidential 28

 Capgemini
ENJOYING THE JOURNEY OF INNOVATION**Sample code:**

```
#include <iostream.h>

template < class T >
struct square {
    T operator()(T n) {return n*n;}
};

template < class T, class funcObj >
void printEval(T val,funcObj f) {
    cout << f(val) << endl;
};

main() {
    printEval(1.4, square < float > ());
    printEval(1.4, square < int > ());
    return 0;
}
```

2.3: Relation between Container, Iterator, Algorithms > 2.3.3: Lab for Iterators and Algorithms

Lab

> Lab3

> Lab4



Hands On

January 19, 2016

Proprietary and Confidential

> 27 <



Capgemini

ENSAUING. THROUGHTS. SOLUTIONS.

Summary

➤ **In this lesson, you have learnt:**

- Iterator helps to iterate through a range of objects.
- Most of the containers support iterators.
- Different types of iterators help in iterating through the objects in different manner.
- Algorithms are common operations performed on elements.
- Algorithms operate through iterators on containers.



Review Questions

- **Question 1: Which of the following perform both read and write operations?**
 - Option 1: Bidirectional
 - Option 2: Forward
 - Option 3: Random access
- **Question 2: Operator overloading feature of the C++ is used by the iterators to represent pointer-like objects**
 - True / False



Review Questions

- Question 3: Algorithm routines are parameterized by ____.



Standard Template Library (STL) Lab Book

Copyright © 2011 IGATE Corporation (a part of Capgemini Group). All rights reserved.

No part of this publication shall be reproduced in any way, including but not limited to photocopy, photographic, magnetic, or other record, without the prior written permission of IGATE Corporation (a part of Capgemini Group).

IGATE Corporation (a part of Capgemini Group) considers information included in this document to be confidential and proprietary.

Document Revision History

Date	Revision No.	Author	Summary of Changes
March 09	1.0	Vaishali Kunchur	Content creation
1-Jun-09		CLS team	Review

Table of Contents

Document Revision History	2
Table of Contents	1
Getting Started	2
Overview	2
Setup Checklist for STL	2
Instructions	2
Learning More (Bibliography if applicable)	2
STL – Sequential Containers	3
1.1: Using vector container	3
Working with Associative Containers	5
2.1: Using set to store unique values in sorted order	5
2.2: Merging values from two sets	6
Working with Containers Adapters	8
3.1: Using stack container	8
Working with iterators and algorithms	9
4.1: Using iterator and algorithms – vector using “find” algorithm	9
4.2: Using sort algorithm for sorting class objects	10
4.3: Using multiset and algorithms for sets	12
Appendices	14
Appendix A: STL on Unix	14

Getting Started

Overview

This lab book is a guided tour for learning Standard Template Library (C++). It comprises solved examples and “To Do” assignments. Follow the steps provided in the solved examples and work out the given “To Do” assignments.

Setup Checklist for STL

Here is what is expected on your machine in order for the lab to work.

Minimum System Requirements

- Intel Pentium 90 or higher (P166 recommended)
- Microsoft Windows 95, 98, or NT 4.0, 2k, XP.
- Memory: 64MB of RAM (128MB or more recommended)
- Microsoft Visual Studio
- Apache Tomcat Version 5.0.
- Telnet Connectivity to Unix Server:
 - Refer to Appendix A to run STL programs on Unix.

Instructions

- For all coding standards refer Appendix A. All lab assignments should refer coding standards.
- Create a directory by your name in drive <drive>. In this directory, create a subdirectory stl_assign. For each lab exercise create a directory as lab <lab number>.
- You may also look up the on-line help provided in the MSDN library.

Learning More (Bibliography if applicable)

- Thinking in C++ Volume Two: Practical Programming by Bruce Eckel and Chuck Allison

STL – Sequential Containers

Goals	<ul style="list-style-type: none">• Understand the use of containers like vector, list, Deque• Learn to apply different functions in container
Time	90 minutes

1.1: Using vector container

Create a vector. Assign values to the vector. Reverse the values in the vector container.

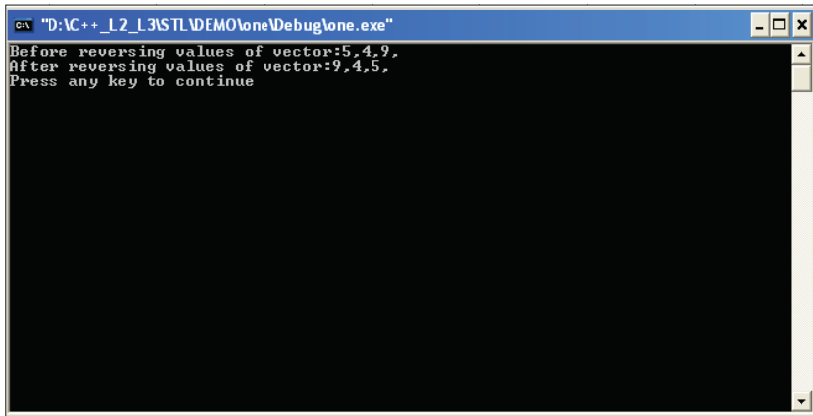
Solution:

Step 1: Create a Console Based Project in Visual Studio 6.0. Create a new C++ source file. Write the following code in the source file.

```
#include <vector> // needed for vector
#include <algorithm> // needed for reverse
#include <iostream>
using namespace std;
void main() {
    vector<int> v(3); // Declare a vector of 3 ints
    v[0] = 5;
    v[1] = 4;
    v[2] = v[0] + v[1];
    cout<<"Before reversing values of vector:";
    for(int i=0;i<v.size();i++)
    {
        cout<<v[i]<<" ";
    }
    cout<<endl;
    reverse(v.begin(), v.end());
    cout<<"After reversing values of vector:";
    for(int j=0;j<v.size();j++)
    {
        cout<<v[j]<<" ";
    }
}
```

Example 1: Sample Code

Step 2: Build(F7) the file and execute.



```
"D:\C++_12_13\STL\DEMO\one\Debug\one.exe"
Before reversing values of vector:5,4,9,
After reversing values of vector:9,4,5,
Press any key to continue
```

Figure 1: Output

<TO DO>

1. Create an integer vector and put the following values into it (34, 65, 22, 43, 12, 51). Sort these elements, and then display the sorted elements.
2. For the above vector, find the sum of numbers stored in vector, and validate if the sum=227.
3. Write a program to create a list of different container names using list container. Use `for_each` to display all the values in the list.

Working with Associative Containers

Goals	<ul style="list-style-type: none">At the end of this lab session, you will be able to understand:<ul style="list-style-type: none">Use of associative containers like set and map
Time	120 minutes

2.1: Using set to store unique values in sorted order

Create a set and store values into the set. Display the values to show that they are stored in an ascending order.

Step 1: Write the following code in the C++ source file of the console based project in Visual Studio.

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

int main()
{
    /* type of the collection:
     * - no duplicates
     * - elements are integral values
     * - ascending order
     */
    typedef set<int,greater<int> > IntSet;

    IntSet coll1;    // empty set container

    // insert elements in random order
    coll1.insert(4);
    coll1.insert(3);
    coll1.insert(5);
    coll1.insert(1);
    coll1.insert(6);
    coll1.insert(2);
    coll1.insert(5);

    copy(coll1.begin(), coll1.end(), ostream_iterator<const int>(cout, " "));
    cout << endl;
}
```

Example 2: Sample Code

Step 2: Build the file and execute the program.

2.2: Merging values from two sets

Create two set with different values assigned to it. Create a method to compare the values in these two sets. Combine the two sets A and B to create a single set C with all the values in sorted order.

Step 1: Write the following code in C++ source file of console based project in Visual Studio.

```
#include <set>
#include <algorithm>
#include <iostream>
using namespace std;

// define how the items are to be tested for equality
struct ltstr
{
    bool operator()(const char* s1, const char* s2) const
    {
        return strcmp(s1, s2) < 0;
    }
};

int main()
{
    const int N = 7;
    const char* a[N] = {"sun", "mon", "tue",
                       "wed", "thur", "fri", "sat"};
    const char* b[N] = {"one", "two", "three",
                       "four", "five", "six", "seven"};

    set<const char*, ltstr> A(a, a + N);
    set<const char*, ltstr> B(b, b + N);
    set<const char*, ltstr> C;

    cout << "Set A: ";
    copy(A.begin(), A.end(), ostream_iterator<const char*>(cout, " "));
    cout << endl;
    cout << "Set B: ";
    copy(B.begin(), B.end(), ostream_iterator<const char*>(cout, " "));
    cout << endl;

    cout << "Union: ";
    set_union(A.begin(), A.end(), B.begin(), B.end(),
             ostream_iterator<const char*>(cout, " "),
             ltstr());
    cout << endl;
    exit(0);
}
```

Example 3: Sample Code

Step 2: Build the file and execute the program.

<TO DO>

4. Write a menu driven program to add a list of unique names to the container. Continue adding elements till the user wants to add. Create functions to add and to delete the elements in the container.
5. Create a class called "Student" with two attributes "Stud_ID" and "age". Create map to store the string type of "student name" and "student class" object. Store the data for four students. Retrieve the "stud_ID" and "age" of a student using his name.

Working with Containers Adapters

Goals	<ul style="list-style-type: none">At the end of this lab session, you will be able to understand:<ul style="list-style-type: none">Use of container adaptors like stacks and queues
Time	60 minutes

3.1: Using stack container

Create a simple stack of integer values. Study different methods like push(), pop(), and size().

Step 1: Create a Console Based Project in Visual Studio 6.0. Create a new C++ source file. Write the following code in the source file.

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

int main() {
    stack<int> mynums;
    mynums.push(55);
    mynums.push(66);
    mynums.push(77);
    mynums.pop();
    cout << "Top element is " << mynums.top() << endl;

    mynums.pop();
    mynums.pop();
    cout << "Number of elements is now " << mynums.size() << endl;
}
```

Example 4: Sample Code

Step 2: Build the file and execute the program.

<TO DO>

6. Create a stack of complex numbers that have integer values. Push three different values in the stack. Retrieve each value and check if the stack is empty after all the values from the stack are removed. Also check the size of the stack before and after the values are removed.
7. Given is a list of values: {23,42,12,45,22,56,64,55}. Create a queue and push the first three elements from the above list. Display the current size. Pop these elements. Now add next five elements and pop them to out. Display the size of the queue.

Working with iterators and algorithms

Goals	<ul style="list-style-type: none"> At the end of this lab session, you will be able to understand: <ul style="list-style-type: none"> Use of iterators and algorithms like find and sort
Time	60 minutes

4.1: Using iterator and algorithms – vector using “find” algorithm

1. Use iterator with vector, and demonstrate the use of “find” algorithm.

Step 1: Create a Console Based Project in Visual Studio 6.0. Create a new C++ source file. Write the following code in the source file.

```
#include <iostream>
#include <cassert>
#include <vector>
#include <algorithm> // For find
using namespace std;
template <typename Container>
Container make(const char s[])
{
    return Container(&s[0], &s[strlen(s)]);
}

int main()
{
    cout << "Demo for find algorithm with a vector" << endl;

    vector<char> vector1 =
        make< vector<char> >("C++ is a better C");

    // Search for the first occurrence of the letter e:
    vector<char>::iterator
        where = find(vector1.begin(), vector1.end(), 'e');

    assert (*where == 'e' && *(where + 1) == 't');
    cout << " --- Ok." << endl;
    return 0;
}
```

Example 5: Sample Code

Step 2: Build the file and execute the program.

4.2: Using sort algorithm for sorting class objects

The following example is for a doubly linked list. Since we are using a class and we are not using defined built-in C++ types, we have included the following:

- We are creating a copy constructor that has been included although the compiler will generate a member-wise one automatically, if required. This has the same functionality as the assignment operator (=).
- The assignment (=) operator must be specified so that sort routines can assign a new order to the members of the list.
- The “less than” (<) operator must be specified so that the sort routines can determine if one class instance is “less than” another.
- The “equals to” (==) operator must be specified so that the sort routines can determine if one class instance is “equal to” another.

Step 1: Create a Console Based Project in Visual Studio 6.0. Create a new C++ source file. Write the following code in the source file.

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

// The List STL template requires overloading operators =, == and <.

class AAA
{
    friend ostream &operator<<(ostream &, const AAA &);
public:
    int x;
    int y;
    float z;
    AAA();
    AAA(const AAA &);
    ~AAA(){};
    AAA &operator=(const AAA &rhs);
    int operator==(const AAA &rhs) const;
    int operator<(const AAA &rhs) const;
};

AAA::AAA() // Constructor
{
    x = 0;
    y = 0;
    z = 0;
}

AAA::AAA(const AAA &copyin)
// Copy constructor to handle pass by value.
{
    x = copyin.x;
    y = copyin.y;
    z = copyin.z;
```

```

    }
    ostream &operator<<(ostream &output, const AAA &aaa)
    {
        output << aaa.x << ' ' << aaa.y << ' ' << aaa.z << endl;
        return output;
    }
    AAA& AAA::operator=(const AAA &rhs)
    {
        this->x = rhs.x;
        this->y = rhs.y;
        this->z = rhs.z;
        return *this;
    }
    int AAA::operator==(const AAA &rhs) const
    {
        if( this->x != rhs.x) return 0;
        if( this->y != rhs.y) return 0;
        if( this->z != rhs.z) return 0;
        return 1;
    }
    // This function is required for built-in STL list functions like sort
    int AAA::operator<(const AAA &rhs) const
    {
        if( this->x == rhs.x && this->y == rhs.y && this->z < rhs.z) return 1;
        if( this->x == rhs.x && this->y < rhs.y) return 1;
        if( this->x < rhs.x ) return 1;
        return 0;
    }

    main()
    {
        list<AAA> L;
        AAA Ablob ;
        Ablob.x=7;
        Ablob.y=2;
        Ablob.z=4.2355;
        L.push_back(Ablob); // Insert a new element at the end
        Ablob.x=5;
        L.push_back(Ablob);
    // Object passed by value. Uses default member-wise
    // copy constructor
        Ablob.z=3.2355;
        L.push_back(Ablob);
        Ablob.x=3;
        Ablob.y=7;
        Ablob.z=7.2355;
        L.push_back(Ablob);
        list<AAA>::iterator i;
        for(i=L.begin(); i != L.end(); ++i) cout << (*i).x << " "; // print member
        cout << endl;
        for(i=L.begin(); i != L.end(); ++i) cout << *i << " ";
    // print all

```

```

        cout << endl;
        cout << "Sorted: " << endl;
        L.sort();
        for(i=L.begin(); i != L.end(); ++i) cout << *i << " ";
// print all
        cout << endl;
        return 0;
    }

```

Example 6: Sample Code

Step 2: Build the file and execute the program.

4.3: Using multiset and algorithms for sets

```

#include <algorithm> // for set_intersection, set_union
#include <iostream>  // for cout, endl
#include <set>       // for set

typedef std::multiset<int, std::less<int>,
                    std::allocator<int> > set_type;

inline std::ostream& operator<< (std::ostream& out,
                                const set_type& s)
{
    typedef std::ostream_iterator<set_type::value_type, char,
    std::char_traits<char> > os_iter;

    std::copy (s.begin (), s.end (), os_iter (std::cout, " "));

    return out;
}

int main ()
{
    // Create a multiset of integers.
    set_type si;

    for (set_type::value_type j = 0; j < 2; j++) {
        for (set_type::value_type i = 0; i < 10; ++i)
            // Insert values with a hint.
            si.insert (si.begin (), i);
    }

    // Print out the multiset.
    std::cout << si << std::endl;
}

```

```

// Make another multiset and an empty multiset.
set_type s12, result;
for (set_type::value_type i = 0; i < 10; i++)
    s12.insert (i + 5);

std::cout << s12 << std::endl;

// Try a couple of set algorithms.
std::set_union (s1.begin (), s1.end (),
               s12.begin (), s12.end (),
               std::inserter (result, result.begin ()));

std::cout << "Union:\n" << result << std::endl;

result.erase (result.begin (), result.end ());

std::set_intersection (s1.begin (), s1.end (),
                      s12.begin (), s12.end (),
                      std::inserter(result, result.begin()));

std::cout << "Intersection:\n" << result << std::endl;

return 0;
}

```

Example 7: Sample Code

<TO DO>

1. The test scores for students are as follows:
 {67, 56, 24, 78, 99, 87, 56}
 Display the following:
 - a. Lowest score
 - b. Highest score
 - c. Number of students passed
 - d. Numbers of students failed
 - e. Average score



Hint: Use the following functions to find the number of students passed and failed

(marks<60):

```
bool pass(int n)
```

```
{
    return (n>=60)
}
```

Use algorithms: min_element, max_element, count_if, accumulate

Appendices

Appendix A: STL on Unix

To run STL programs in Unix we can use the following steps:

Step 1: Connect to Unix server using start -> run -> telnet unix_server

Step 2: vi vectordemo.cpp

Step 3: Type the code given in Lab 1.1 in vi editor.

Step 4: Save the file and exit from vi editor

Step 5: To compile this file, use the following command:

```
$ g++ vectordemo.cpp
```

This will compile the file and generate an executable file a.out.

Step 6: To see the output, use the following command:

```
$ ./a.out
```