



Dr. Vishwanath Karad

**MIT WORLD PEACE  
UNIVERSITY** | PUNE

TECHNOLOGY, RESEARCH, SOCIAL INNOVATION & PARTNERSHIPS

**CET1046B**

**Object Oriented Concepts with  
C++ and Java**

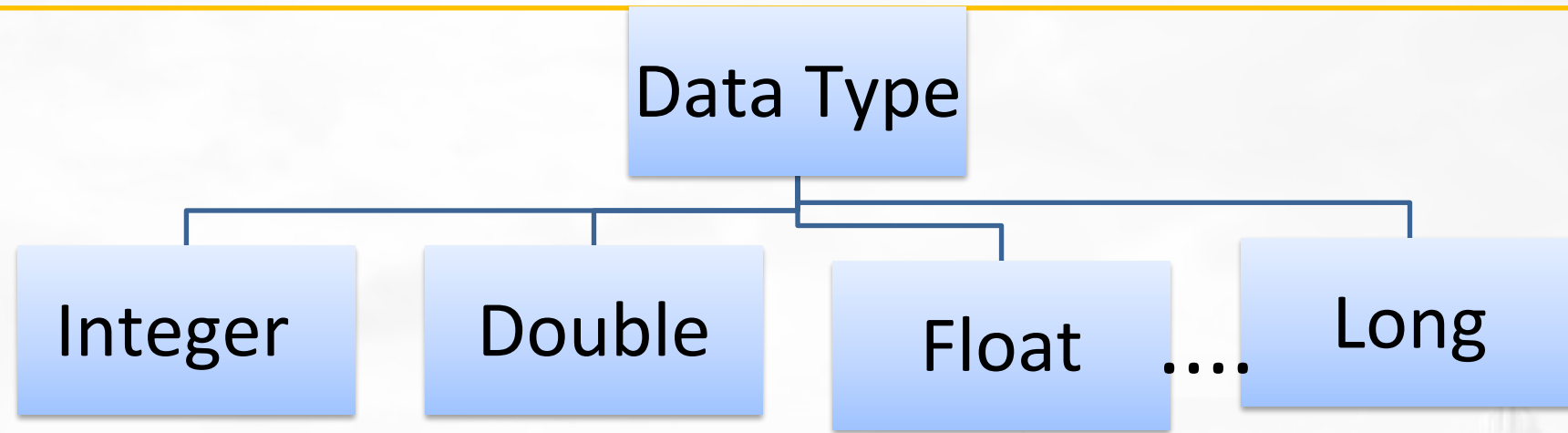


# Templates and Standard Template Library (STL)

Templates: Function Template, Class Template.

Introduction to STL, Containers - Sequence Containers and Associative Containers, Container Adapters, Algorithms and Iterators - input, output, forward, bidirectional and random access.

# Maximum of Two Numbers



<pre>int maximum(int a, int b) {     if (a &gt; b)         return a;     else         return b; }</pre>	<pre>double maximum(double a, double b) {     if (a &gt; b)         return a;     else         return b; }</pre>	<pre>float maximum(float a, float b) {     if (a &gt; b)         return a;     else         return b; }</pre>
---	--	---

# A Template Function for Maximum

- This template function can be used with many data types.

```
template <class T>
T maximum(T a, T b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

# Templates

- ❑ **Templates** are a feature of the C++ programming language that allow functions and classes to operate with generic types. This allows a function or class to work on many different data types without being rewritten for each one.
- ❑ **Generic programming** is about generalizing software components so that they can be easily reused in a wide variety of situations.
- ❑ C++ uses templates to enable generic programming techniques.



# C++ provides two kinds of templates

- Function templates
- Class templates



# Sample Function Template to find absolute value

```
#include iostream
using namespace std;
//function template
template <class T>
T abs1(T n) {
    return (n<0)? -n:n;
}
```

```
abs(5)=5
abs(-6)=6
abs(70000)=70000
abs(-80000)=80000
abs(9.95)=9.95
abs(-10.15)=10.15
```

```
int main(){
int int1 = 5, int2 = -6;
long lon1 = 70000L, lon2 = -80000L;
double dub1 = 9.95, dub2 = -10.15;
cout << "\nabs(" << int1 << ")=" << abs1(int1); //abs(int)
cout << "\nabs(" << int2 << ")=" << abs1(int2); //abs(int)
cout << "\nabs(" << lon1 << ")=" << abs1(lon1); //abs(int)
cout << "\nabs(" << lon2 << ")=" << abs1(lon2); //abs(int)
cout << "\nabs(" << dub1 << ")=" << abs1(dub1); //abs(int)
cout << "\nabs(" << dub2 << ")=" << abs1(dub2); //abs(int)
```

# Function Template to find maximum of 2 numbers

```
#include<iostream>
using namespace std;
template<class T>
void max1(T x, T y) // T represent generic data type
{
    if(x>y)
        cout<<x<<"is greater"<<endl;
    else
        cout<<y<<"is greater"<<endl;
}
```

```
int main()
{
    max1(3,7);
    max1(44.66,55.66);
    return 0;
}
```

**Output:**

**7 is greater**

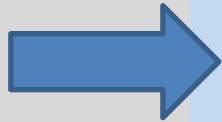
**55.56 is greater**



# Finding the Maximum Item in an Array

A function that can be made more general by changing it to a template function:

```
int array_max(int data[ ], int n)
{
    int i;
    int answer;
    answer = data[0];
    for (i = 1; i < n; i++)
        if (data[i] > answer)
            answer = data[i];
    return answer;
}
```



```
template <class Item>
Item array_max(Item data[ ], int n)
{
    int i;
    Item answer;
    assert(n > 0);
    answer = data[0];
    for (i = 1; i < n; i++)
        if (data[i] > answer)
            answer = data[i];
    return answer;
}
```

```
int main() {
    int array[]={5,4,6,2,1},ians;
    double array1[]={2.3,4.3,1.1,6.7},dans;
    ians=array_max(array,5);
    cout<<"max from int array is"<<ians;
    dans=array_max(array1,4);
    cout<<"max from double array
is"<<dans;
    return 0;
}
```

# Function Overloading Vs Function Template

```
#include <iostream>
using namespace std;
int square (int x)
{
    return x * x;
}
double square (double x)
{
    return x * x;
}
```

```
#include <iostream>
using namespace std;
template <typename T>
T square(T x)
{
    T result;
    result = x * x;
    return result;
}
```

```
int main()
{
    int i, ii;
    double d, dd;
    i = 2;
    d = 2.2;
    ii = square(i);
    cout << "Square of Integer Number " << " : " << ii << endl;
    dd = square(d);
    cout << "Square of double number " << " : " << dd << endl;
    return 0;
}
```

## Ex

- Write a generic function that swaps values in two variables. Function should have two parameters of the same type. Test the function with integer and double values.

# Solution of Ex

```
#include <iostream>
using namespace std;
template <typename X>
void swapargs(X *a, X *b){
    X temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

```
int main(){
    int i=10, j=20; float x=10.1, y=23.3;
    cout << "Original i, j: " << i << ' ' << j << endl;    cout
    << "Original x, y: " << x << ' ' << y << endl;
    swapargs(&i,&j); // swap integers  swapargs(&x,&y);
    // swap floats

    cout << "Swapped i, j: " << i << ' ' << j << endl;    cout
    << "Swapped x, y: " << x << ' ' << y << endl;  return 0;
}
```

# Class Template

- It represent various similar classes operating on data of different types.

Syntax:

```
template<class T>  
Class class_name  
{  
    Class members  
}
```



# Class template Example 1 - Adding 2 numbers

```
template <class T>
class Addition {    // Template class
    public:
    T Add(T, T);
};

template <class T>
T Addition<T>::Add(T n1, T n2) {
    T rs;
    rs = n1 + n2;
    return rs;
}
```

```
int main()
{
    Addition <int>obj1;
    Addition <long>obj2;
    int A=10,B=20,C;
    long l=11,J=22,K;
    C = obj1.Add(A,B);
    cout<<"\nThe sum of integer values :"<<C;
    K = obj2.Add(l,J);
    cout<<"\nThe sum of long values :"<<K;
}
```

The sum of integer values: 30

The sum of long values: 33



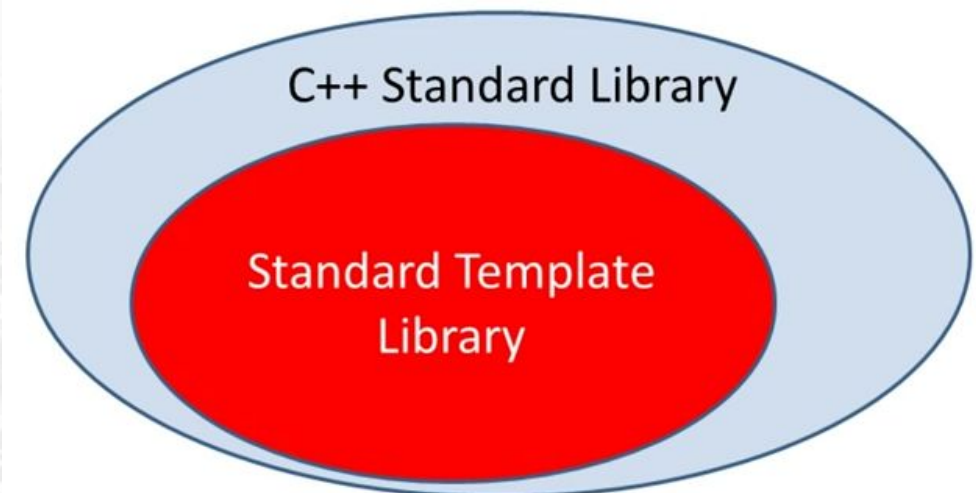
# Class template Example 2-Display arguments

```
template<class T1,class T2>
class sample {
    T1 a;
    T2 b;
public:
    void getdata() {
        cout<<"Enter a and b: "<<endl;
        cin>>a>>b;
    }
    void display() {
        cout<<"Displaying values"<<endl;
        cout<<"a="<<a<<endl;
        cout<<"b="<<b<<endl;
    }
};
```

```
int main() {
    sample<int,int> s1;
    sample<int,char> s2;
    sample<int,float> s3;
    cout <<"Two Integer data"<<endl;
    s1.getdata();
    s1.display();
    cout <<"Integer and Character data"<<endl;
    s2.getdata();
    s2.display();
    cout <<"Integer and Float data"<<endl;
    s3.getdata();
    s3.display();
    return 0;
}
```

# STL-Standard Template Library

- ❑ The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc. It is a library of container classes, algorithms and iterators. It is a generalized library and so, its components are parameterized.
- ❑ Collection of useful classes for common data structure
- ❑ Ability to store objects of any type (template)
- ❑ **STL has four components**
  - Algorithms
  - Containers
  - Functions
  - Iterators



# Why to use STL?

- ❑ It offers an assortment of containers
- ❑ Publicizes the time and storage complexity of its containers
- ❑ Containers grow and shrink in size automatically
- ❑ Provides built-in algorithms for processing containers
- ❑ Provides iterators that make the containers and algorithms flexible and efficient.
- ❑ It is extensible which means that users can add new containers and new algorithms such that:
  - STL algorithms can process STL containers as well as user defined containers
  - User defined algorithms can process STL containers as well user defined containers

# Standard Template Library

- ❑ The STL-Standard Template Library contains:
  - Containers
  - Algorithms
  - Iterators
- ❑ A container is a way that stored data is organized in memory , for example an array of elements.
- ❑ Algorithms in the STL are producers that are applied to containers to process their data for example search for an element in an array or sort an array.
- ❑ Iterators are a generalization of the concept of pointers they point to elements in a container for example you can increment an iterator to point to the next element in an array.



# Component of STL

## □ Containers

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

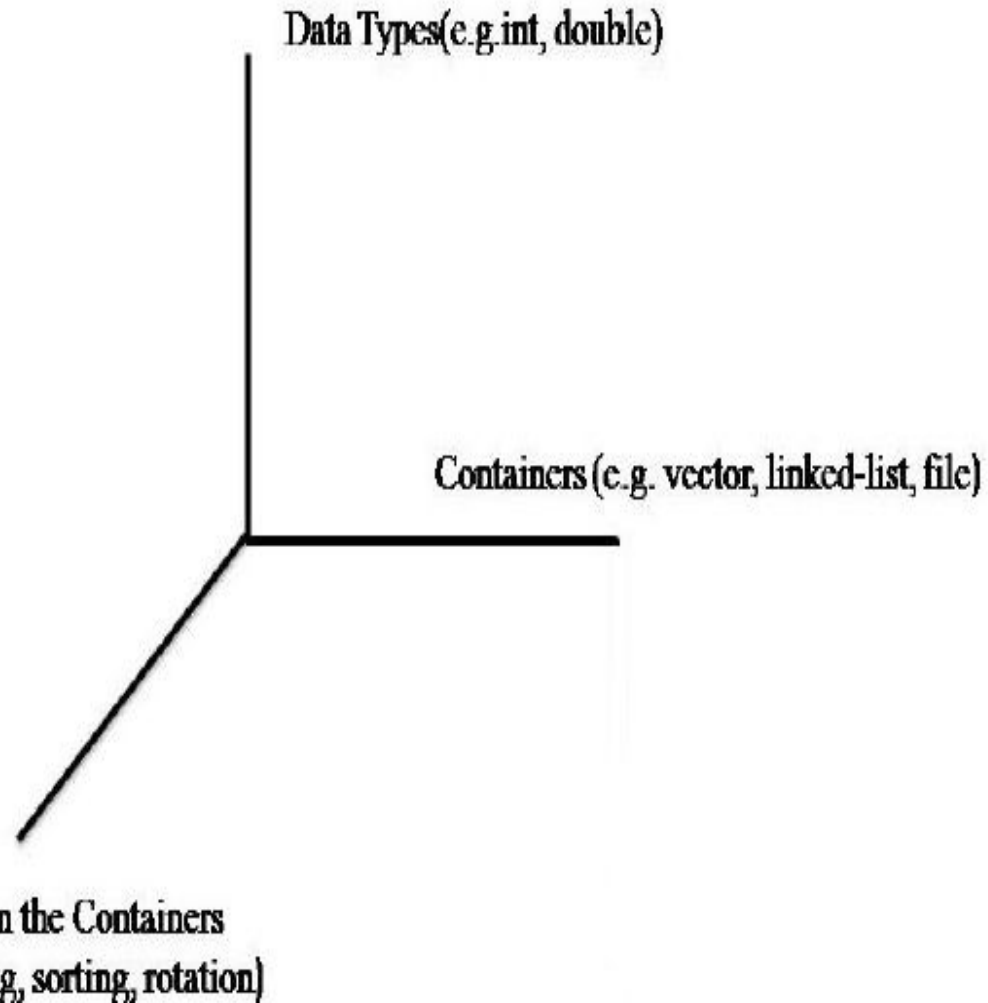
## □ Iterators

They are a generalization of pointers: they are objects that point to other objects. As the name suggests, iterators are often 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.

## □ Algorithms

The header `<algorithm>` defines a collection of functions especially designed to be used on ranges of elements.

# Component of STL



- Containers: template data structures.
  - *Containers* hold data
- Iterators: like pointers, access elements of containers
  - *Iterators* access data
- Algorithms: data manipulation, searching, sorting, etc.
  - *Algorithms, function objects* manipulate data



# Component of STL

## Algorithms

sort, find, search, copy, ...

iterators

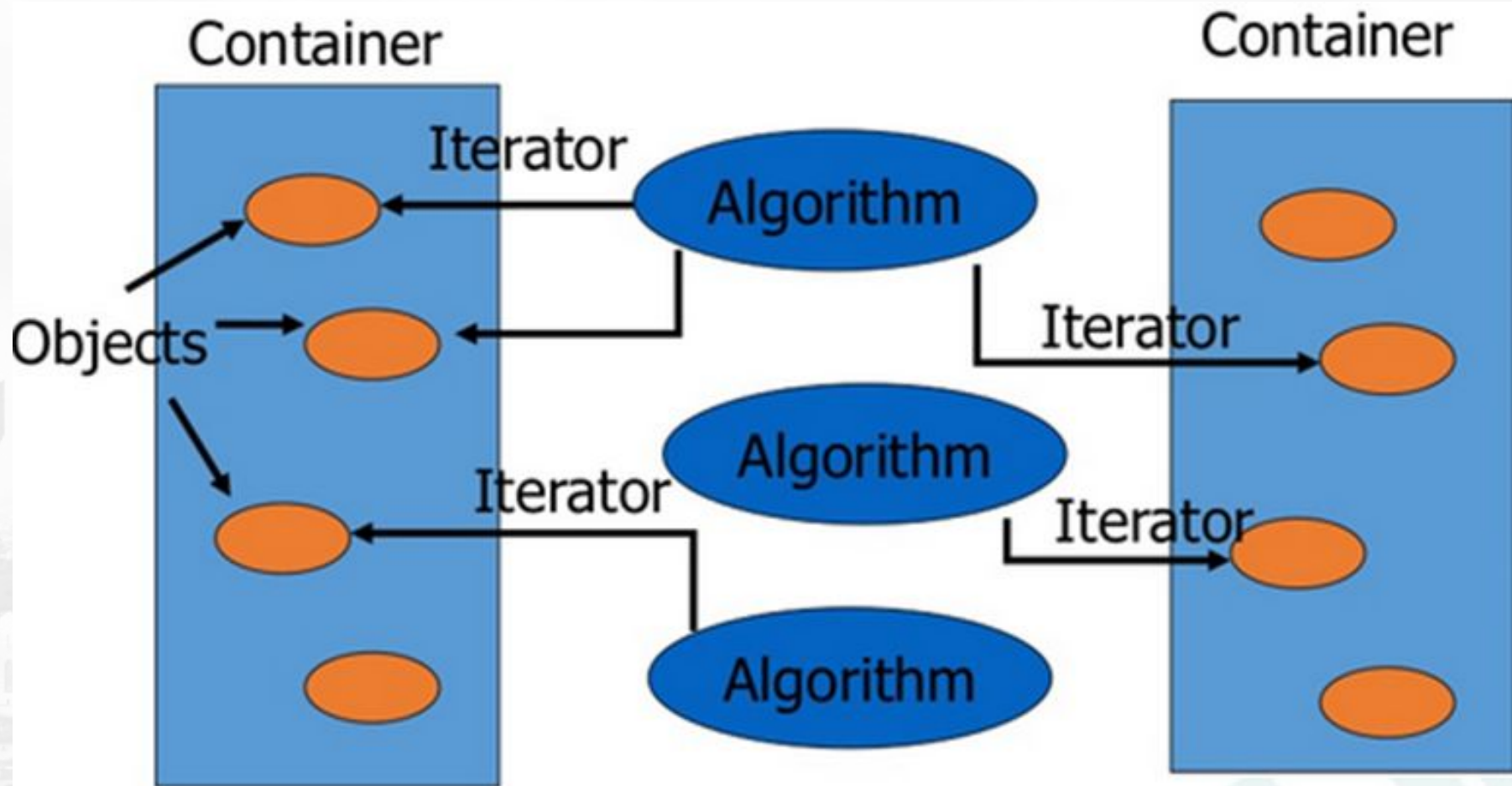
## Containers

vector, list, map, hash\_map, ...

- **Separation of concerns**

- Algorithms manipulate data, but don't know about containers
- Containers store data, but don't know about algorithms
- Algorithms and containers interact through iterators
  - Each container has its own iterator types

# Standard Template Library



# Container

- ❑ A container is a way to store data, either built-in data types like int and float or class objects .
- ❑ Containers or container classes store objects and data. There are in total seven standard “first-class” container classes and three container adaptor classes and only seven header files that provide access to these containers or container adaptors.





# Container

Standard Library container class	Description
<i>Sequence containers</i>	
<b>Vect or</b>	rapid insertions and deletions at back direct access to any element
<b>deque</b>	rapid insertions and deletions at front or back direct access to any element
<b>l i s t</b>	doubly linked list, rapid insertion and deletion anywhere
<i>Associative containers</i>	
<b>set</b>	rapid lookup, no duplicates allowed
<b>mul t i s e t</b>	rapid lookup, duplicates allowed
<b>map</b>	one-to-one mapping, no duplicates allowed, rapid key-based lookup
<b>mul t i map</b>	one-to-many mapping, duplicates allowed, rapid key-based lookup
<i>Container adapters</i>	
<b>stack</b>	last-in, first-out (LIFO)
<b>queue</b>	first-in, first-out (FIFO)
<b>pri o r i t y _queue</b>	highest-priority element is always the first element out

# Vector – Sequence Container

□ Defining a new vector

Header file: `<vector>`

Syntax: `vector<int> vec`

↓      ↓      ↓  
Keyword   Data Type   Object



# Vector - Operations

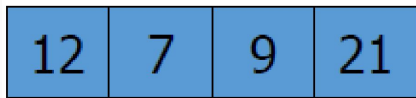
- `size()`
  - provides the number of elements
- `push_front()` & `push_back()`
  - appends an element at the front and end
- `pop_front()` & `pop_back()`
  - Erases the last element
- `begin()`
  - Provides reference to first element
- `end()`
  - Provides reference to end of vector

# Vector Container

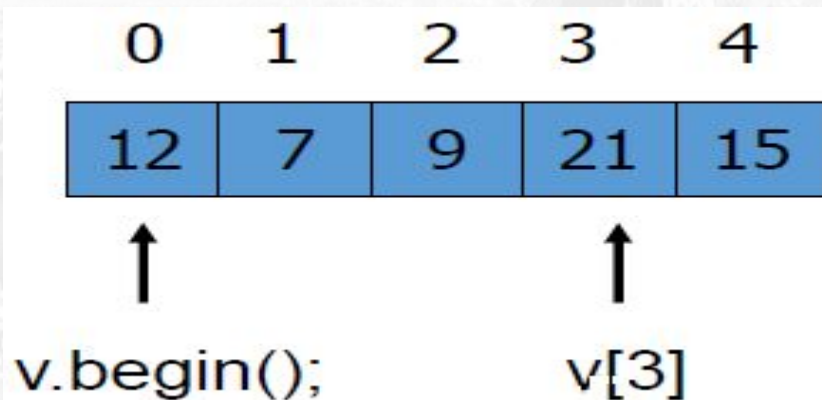
```
int array[5] = {12, 7, 9, 21, 13};  
vector<int> v(array, array+5);
```



```
v.pop_back();
```



```
v.push_back(15);
```



# Vector



```
vector<int> vec;    // vec.size() == 0
vec.push_back(4);
vec.push_back(1);
vec.push_back(8);  // vec: {4, 1, 8};    vec.size() == 3
```

```
// Vector specific operations:
cout << vec[2];      // 8   (no range check)
cout << vec.at(2);   // 8   (throw range_error exception of out of range)
```

# Vector container using stack

```
for (int i; i < vec.size(); i++) {  
    cout << vec[i] << " ";  
}
```

Output: "4 1 8"

```
cout << vec.size();    // 3
```

```
vector<int> vec2(vec);  // Copy constructor, vec2: {4, 1, 8}
```

```
vec.clear();           // Remove all items in vec;    vec.size() == 0
```

```
vec2.swap(vec);        // vec2 becomes empty, and vec has 3 items.
```



# Vector- Member Functions `push_back()`, `size()`, and `operator[]`

```
#include <vector>
using namespace std;
int main()
{
    vector<int> v; //create a vector of ints
    v.push_back(10); //put values at end of array
    v.push_back(11);
    v.push_back(12);
    v.push_back(13);
    v[0] = 20; //replace with new values
    v[3] = 23;
    for(int j=0; j<v.size(); j++) //display vector contents
        cout << v[j] << ' '; //20 11 12 23
    cout << endl;
    return 0;
}
```

# Vector- Member Functions swap(), empty(), back(), and pop\_back()

```
#include <vector>
using namespace std;
int main()
{ //an array of doubles
    double arr[] = { 1.1, 2.2, 3.3, 4.4 };
    vector<double> v1(arr, arr+4); //initialize vector to array
    vector<double> v2(4); //empty vector of size 4
    v1.swap(v2); //swap contents of v1 and v2
    while( !v2.empty() ) //until vector is empty,
    {
        cout << v2.back() << ' '; //display the last element
        v2.pop_back(); //remove the last element
    } //output: 4.4 3.3 2.2 1.1
    cout << endl;
    return 0;
}
```



# Vector- Member Functions insert() and erase()

```
#include <iostream>
#include <vector>
using namespace std;
int main(){
    int arr[] = { 100, 110, 120, 130 }; //an array of ints
    vector<int> v(arr, arr+4); //initialize vector to array
    cout << "\nBefore insertion: ";
    for(int j=0; j<v.size(); j++) //display all elements
        cout << v[j] << ' ';
    v.insert( v.begin()+2, 115); //insert 115 at element 2
    cout << "\nAfter insertion: ";
    for(j=0; j<v.size(); j++) //display all elements
        cout << v[j] << ' ';
    v.erase( v.begin()+2 ); //erase element 2
    cout << "\nAfter erasure: ";
    for(j=0; j<v.size(); j++) //display all elements
        cout << v[j] << ' ';
    return 0;
}
```

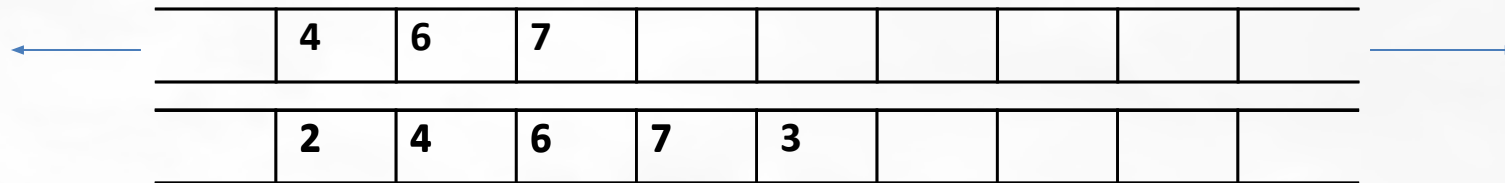
**O/p**

**Before insertion: 100 110 120 130**

**After insertion: 100 110 115 120 130**

**After erasure: 100 110 120 130**

# Deque



```
/*  
 * Deque  
 */  
deque<int> deq = { 4, 6, 7 };  
  
deq.push_front(2); // deq: {2, 4, 6, 7}  
deq.push_back(3);  // deq: {2, 4, 6, 7, 3}  
  
// Deque has similar interface with vector  
cout << deq[1]; // 4
```

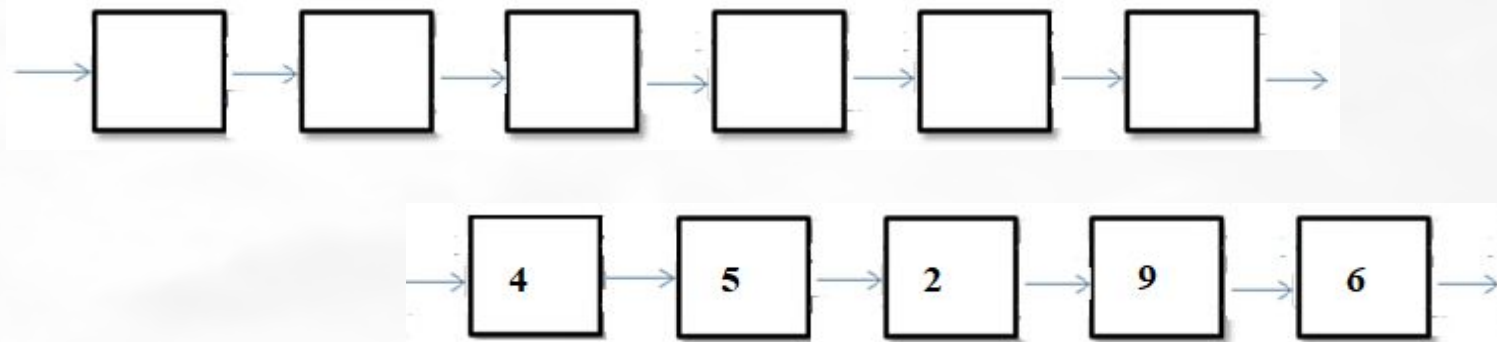
# Deque

```
#include <iostream>
#include <deque>
using namespace std;
int main()
{
    deque<int> deq;
    deq.push_back(30); //push items on back
    deq.push_back(40);
    deq.push_back(50);
    deq.push_front(20); //push items on front
    deq.push_front(10);
    deq[2] = 33; //change middle item
    for(int j=0; j<deq.size(); j++)
        cout << deq[j] << ' '; //display items
    cout << endl;
    return 0;
}
```

**Output:**

**10 20 33 40 50**

# List



```
list<int> mylist = {5, 2, 9 };  
mylist.push_back(6); // mylist: { 5, 2, 9, 6}  
mylist.push_front(4); // mylist: { 4, 5, 2, 9, 6}
```



# Member Functions `push_front()`, `front()`, and `pop_front`

```
#include <iostream>
#include <list>
using namespace std;
int main()
{
    list<int> ilist;
    ilist.push_back(30); //push items on back
    ilist.push_back(40);
    ilist.push_front(20); //push items on front
    ilist.push_front(10);
    int size = ilist.size(); //number of items
    for(int j=0; j<size; j++)
    {
        cout << ilist.front() << ' '; //read item from front
        ilist.pop_front(); //pop item off front
    } cout << endl; return 0;
}
```

**Output:**  
**10 20 30 40**



# Member Functions reverse(), merge(), and unique()

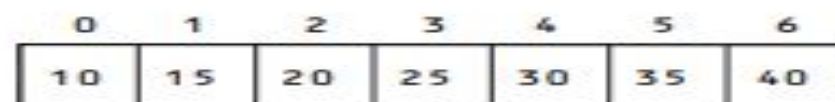
```
#include <iostream>
#include <list>
using namespace std;
int main(){
    int j;
    list<int> list1, list2;
    int arr1[] = { 40, 30, 20, 10 };
    int arr2[] = { 15, 20, 25, 30, 35 }
    for(j=0; j<4; j++)
        list1.push_back( arr1[j] ); //list1: 40, 30, 20, 10
    for(j=0; j<5; j++)
        list2.push_back( arr2[j] ); //list2: 15, 20, 25, 30, 35
```

```
list1.reverse(); //reverse list1: 10 20 30 40
list1.merge(list2); //merge list2 into list1
list1.unique(); //remove duplicate 20 and 30
int size = list1.size();
while( !list1.empty() )
{
    cout << list1.front() << ' '; //read item from front
    list1.pop_front(); //pop item off front
}
cout << endl;
return 0;
}
```

**Output:**

```
10, 15, 20, 20, 25, 30, 30, 35, 40
15, 20, 25, 30, 35
```

### VECTOR

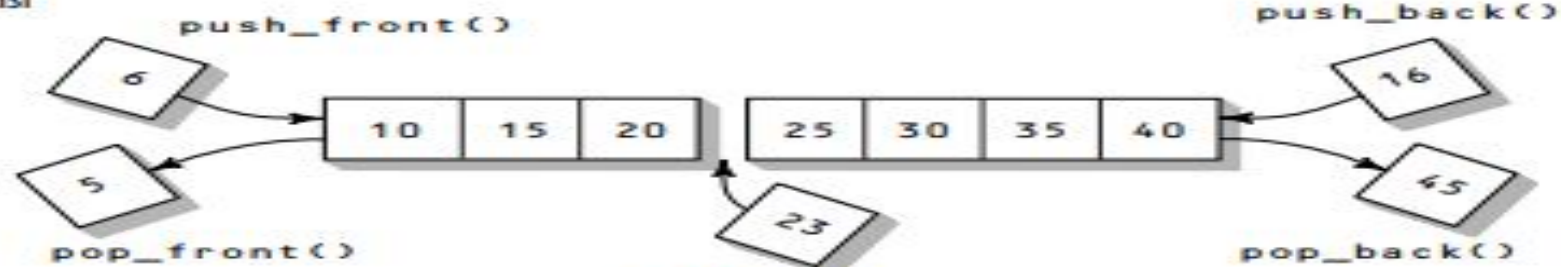


`intVect [3] == 25`

`push_back()`

`pop_back()`

### LIST



`push_front()`

`pop_front()`

`insert()`

`push_back()`

`pop_back()`

### DEQUE



`intDeque [3] == 25`

`push_front()`

`pop_back`

`push_back()`

`pop_back()`

# Stack container

```
#include<stack>
#include<iostream>
using namespace std
int main()
{
    stack <int> s
    int ch,elem;
    cout<<"push"<<"pop"
    cin>>ch;
    swicth(ch)
    {
        case 1:cout<<"enter element:"
                cin>>elem;
                s.push(elem);
                break;
```

```
        case 2: if(!stack.empty())
                  {
                    cout<<stack.top();
                    s.pop();
                  }
                else
                    cout<<"\n stack empty:"
            }
    }
```

# Vector container using stack

```
#include<vector>
class Stack {
    public :
        int element;
        vector <int> s;
        vector <int> :: iterator itr;
        void push();
        void displaystack();
        void pop();
}
```

```
void push()
{
    cout<<"\n Enter a number : ";
    cin>>a;
    s.push_back(a);
}
```

# Vector container using stack

```
void displaystack()
{
    cout<<"\n The elements in the stack are : "<<"\n";
    for(itr=s.begin() ; itr!=s.end() ; itr++)
    {
        cout<<*itr<<"\t";
    }
}
```

```
void pop()
{
    itr=s.end();
    itr--;
    s.pop_back();
    cout<<"\n The element popped
out of the stack is "<<*itr;
}
```

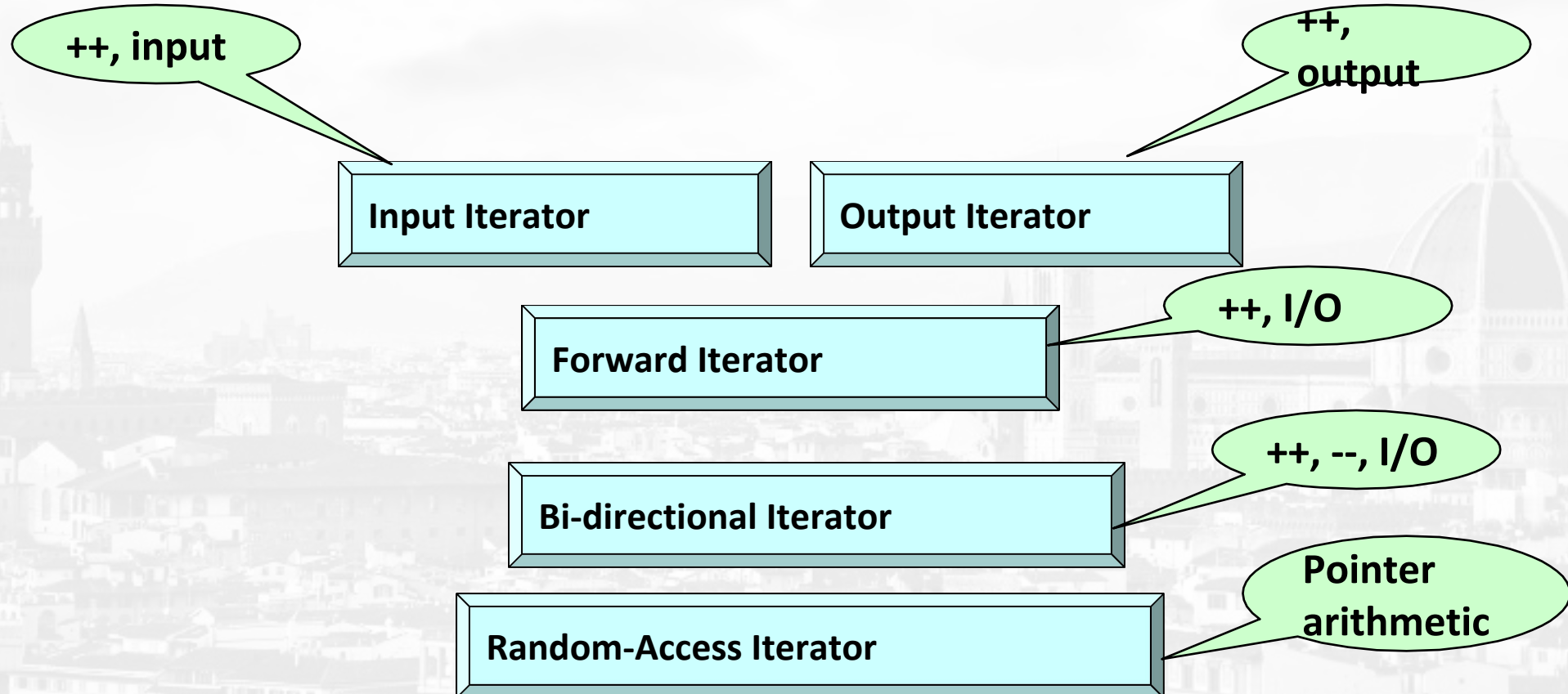
```
int main(){
    Stack p;
    p.push();
    p.displaystack();
    p.pop();
}
```



# Iterators

- Iterators are pointer-like entities that are used to access individual data items (which are usually called elements), in a container.
- Often they are used to move sequentially from element to element, a process called iterating through the container.
- can increment iterators with the `++` operator so they point to the next element, and dereference them with the `*` operator to
- obtain the value of the element they point to.
- In the STL an iterator is represented by an object of an iterator class.
- Different classes of iterators must be used with different types of container.
- There are three major classes of iterators: forward, bidirectional, and random access.

# Iterators



# Types of iterators

- A **forward iterator** can only move forward through the container, one item at a time. Its ++ operator accomplishes this. It can't move backward and it can't be set to an arbitrary location in the middle of the container.
- A **bidirectional iterator** can move backward as well as forward, so both its ++ and -- operators are defined.
- A **random access iterator**, in addition to moving backward and forward, can jump to an arbitrary location. You can tell it to access location 27, for example.
- There are also two specialized kinds of iterators.
- An **input iterator** can “point to” an input device (cin or a file) to read sequential data items into a container, and
- an **output iterator** can “point to” an output device (cout or a file) and write elements from a container to the device.
- While the values of forward, bi-directional, and random access iterators can be stored (so they can be used later), the values of input and output iterators cannot be.
- first three iterators point to memory locations, while input and output iterators point to I/O devices for which stored “pointer” values have no meaning.

# Iterator Characteristics

<i>Iterator Type</i>	<i>Read/Write</i>	<i>Iterator Can Be Saved</i>	<i>Direction</i>	<i>Access</i>
Random access	Read and write	Yes	Forward and back	Random
Bidirectional	Read and write	Yes	Forward and back	Linear
Forward	Read and write	Yes	Forward only	Linear
Output	Write only	No	Forward only	Linear
Input	Read only	No	Forward only	Linear



# Iterators

Container	Type of iterator supported
<i>Sequence containers</i>	
<b>vector</b>	random access
<b>deque</b>	random access
<b>list</b>	bidirectional
<i>Associative containers</i>	
<b>set</b>	bidirectional
<b>multiset</b>	bidirectional
<b>map</b>	bidirectional
<b>multimap</b>	bidirectional
<i>Container adapters</i>	
<b>stack</b>	no iterators supported
<b>queue</b>	no iterators supported
<b>priority_queue</b>	no iterators supported

# Vector – Sequence Container



```
vector<int> vec;    // vec.size() == 0
vec.push_back(4);
vec.push_back(1);
vec.push_back(8);  // vec: {4, 1, 8};    vec.size() == 3
```

```
// Vector specific operations:
cout << vec[2];      // 8   (no range check)
cout << vec.at(2);   // 8   (throw range_error exception of out of range)
```

# Vector container using iterator

```
for (int i; i < vec.size(); i++) {  
    cout << vec[i] << " ";  
}
```

Output: "4 1 8"

```
vector<int> vec;  
vector<int>::iterator it;
```

```
cout << "Vector contains:";  
for (it=vec.begin(); it!=vec.end(); ++it)  
    cout << ' ' << *it;
```

Output: "4 1 8"

# Algorithms

- The Standard Template Library provides a number of useful, generic algorithms to perform the most commonly used operations on groups/sequences of elements. These operations include traversals, searching, sorting and insertion/removal of elements.
- The way that these algorithms are implemented is closely related to the idea of the containers and iterators, although, as we will see, they can be used with standard arrays and pointers.



# Algorithms

STL algorithms are not member functions or friends of containers. They are standalone template functions. To have access to the STL algorithms, we must include **<algorithm>** in our program.

<i>Algorithm</i>	<i>Purpose</i>
<code>find</code>	Returns first element equivalent to a specified value
<code>count</code>	Counts the number of elements that have a specified value
<code>equal</code>	Compares the contents of two containers and returns true if all corresponding elements are equal
<code>search</code>	Looks for a sequence of values in one container that corresponds with the same sequence in another container
<code>copy</code>	Copies a sequence of values from one container to another (or to a different location in the same container)
<code>swap</code>	Exchanges a value in one location with a value in another
<code>iter_swap</code>	Exchanges a sequence of values in one location with a sequence of values in another location
<code>fill</code>	Copies a value into a sequence of locations
<code>sort</code>	Sorts the values in a container according to a specified ordering
<code>merge</code>	Combines two sorted ranges of elements to make a larger sorted range
<code>accumulate</code>	Returns the sum of the elements in a given range
<code>for_each</code>	Executes a specified function for each element in the container

# Algorithms

```
/*
 * Algorithms
 * - mostly loops
 */
vector<int> vec = { 4, 2, 5, 1, 3, 9 };
vector<int>::iterator itr = min_element(vec.begin(), vec.end()); // itr -> 1

// Note 1: Algorithm always process ranges in a half-open way: [begin, end)
sort(vec.begin(), itr); // vec: { 2, 4, 5, 1, 3, 9 }

reverse(itr, vec.end()); // vec: { 2, 4, 5, 9, 3, 1 }   itr => 9

// Note 2:
vector<int> vec2(3);
copy(itr, vec.end(), // Source
      vec2.begin()); // Destination
//vec2 needs to have at least space for 3 elements.
```

# The find() Algorithm

```
#include <iostream>
#include <algorithm> //for find()
using namespace std;
int arr[] = { 11, 22, 33, 44, 55, 66, 77, 88 };
int main()
{
    int* ptr;
    ptr = find(arr, arr+8, 33); //find first 33
    cout << "First object with value 33 found at offset "<< (ptr-arr) << endl;
    return 0;
}
```



# The count() Algorithm

```
#include <iostream>
#include <algorithm> //for count()
using namespace std;
int arr[] = { 33, 22, 33, 44, 33, 55, 66, 77 };
int main()
{
    int n = count(arr, arr+8, 33); //count number of 33's
    cout << "There are " << n << " 33's in arr." << endl;
    return 0;
}
```



# The sort() Algorithm

```
#include <iostream>
#include <algorithm>
using namespace std;
//array of numbers
int arr[] = {45, 2, 22, -17, 0, -30, 25, 55};
int main()
{
    sort(arr, arr+8); //sort the numbers
    for(int j=0; j<8; j++) //display sorted array
        cout << arr[j] << ' ';
    cout << endl;
    return 0;
}
```

# The search() Algorithm

```
#include <iostream>
#include <algorithm>
using namespace std;
int source[] = { 11, 44, 33, 11, 22, 33, 11, 22, 44 };
int pattern[] = { 11, 22, 33 };
int main()
{
    int* ptr;
    ptr = search(source, source+9, pattern, pattern+3)
    if(ptr == source+9) //if past-the-end
        cout << "No match found\n";
    else
        cout << "Match at " << (ptr - source) << endl;
    return 0;
}
```

# The merge() Algorithm

```
#include <iostream>
#include <algorithm> //for merge()
using namespace std;
int src1[] = { 2, 3, 4, 6, 8 };
int src2[] = { 1, 3, 5 };
int dest[8];
int main()
{ //merge src1 and src2 into dest
    merge(src1, src1+5, src2, src2+3, dest);
    for(int j=0; j<8; j++) //display dest
        cout << dest[j] << ' ';
    cout << endl;
    return 0;
}
```

# define an iterator for accessing the contents of a LIST container

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
int main(){
    int arr[] = { 2, 4, 6, 8 };
    list<int> theList;
    for(int k=0; k<4; k++) //fill list with array elements
        theList.push_back( arr[k] );
    list<int>::iterator iter; //iterator to list-of-ints
    for(iter = theList.begin(); iter != theList.end(); iter++)
        cout << *iter << ' '; //display the list
    return 0;
}
```

**Output:**  
2 4 6 8



# Using an iterator for inserting data in the LIST container

```
#include <iostream>
#include <list>
using namespace std;
int main(){
list<int> iList(5); //empty list holds 5 ints
list<int>::iterator it; //iterator
int data = 0;
//fill list with data
for(it = iList.begin(); it != iList.end(); it++)
    *it = data += 2;
//display list
for(it = iList.begin(); it != iList.end(); it++)
    cout << *it << ' ';
return 0;
}
```

**Output:**  
2, 4, 6, 8, 10

# Algorithms and Iterators

```
//look for number 8
```

```
iter = find(theList.begin(), theList.end(), 8);
```

```
if( iter != theList.end() )
```

```
    cout << "\nFound 8.\n";
```

```
else
```

```
    cout << "\nDid not find 8.\n";
```

```
return 0;
```

```
}
```

- An algorithm, `find()` takes three arguments. The first two are iterator values specifying the range to be searched, and the third is the value to be found.
- To find the position of element 8.
- this is not a legal operation on the iterators used for lists.
- A list iterator is only a bidirectional iterator, so you can't perform arithmetic with it.
- Arithmetic can be done with with random access iterators, such as those used with vectors and queues as below.

```
iter = find(v.begin(), v.end(), 8);
```

```
if( iter != v.end() )
```

```
    cout << "\nFound 8 at location " << (iter-v.begin() );
```

```
else
```

```
    cout << "\nDid not find 8.";
```

# list

```
#include <list>
int main() {
    list<int> la, lb;
    la.push_back(0), la.push_back(1), la.push_back(3);
    lb.push_back(4), lb.push_front(2);
    la.sort(); lb.sort();
    la.merge(lb);
    print(la);
}
/* Output:
4 3 2 1 0
*/
```

# list

// Standard Template Library example

```
#include <iostream>
```

```
#include <list>
```

```
using namespace std;
```

```
// Simple example uses type 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;
```

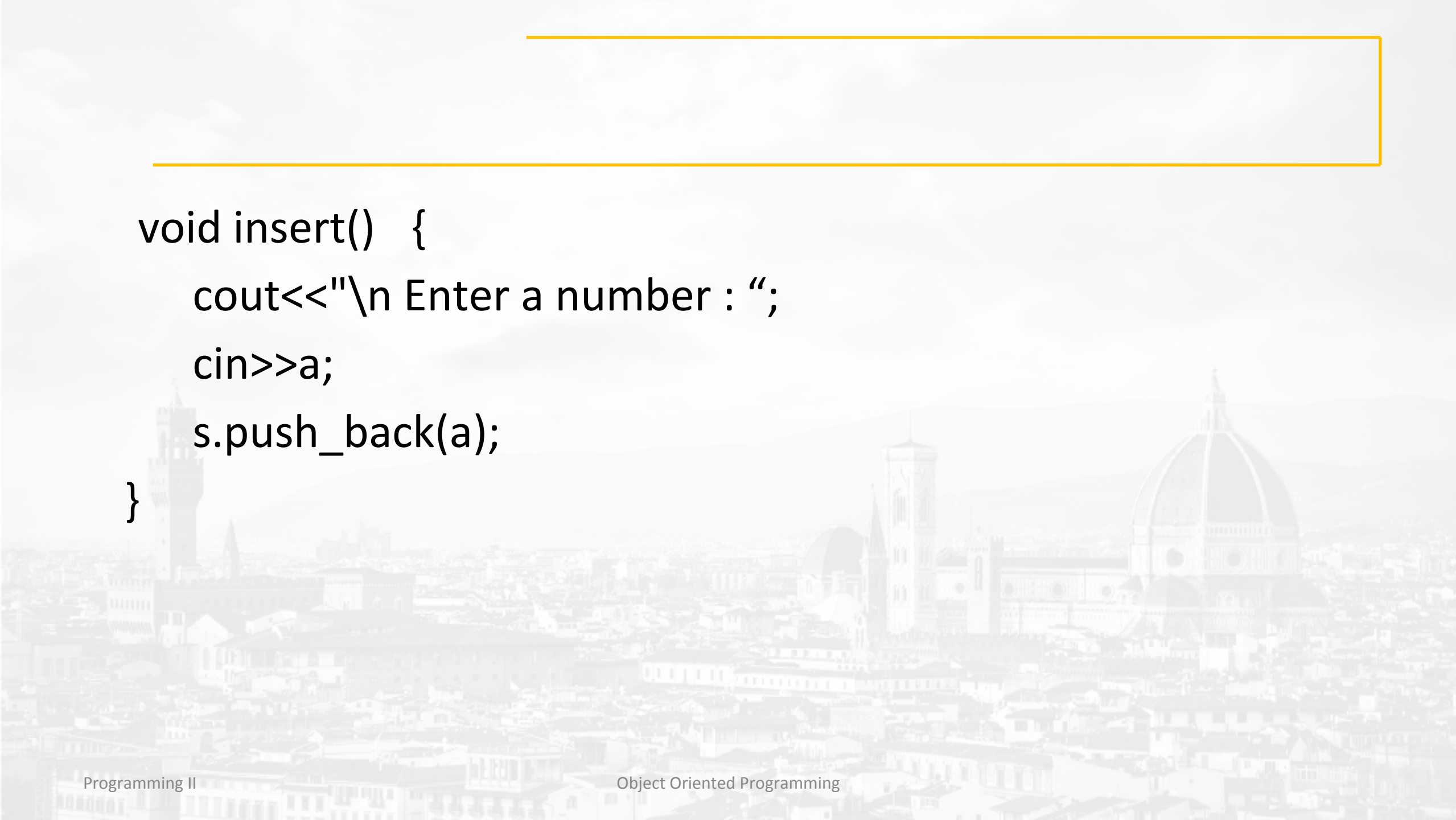
```
    return 0;
```

```
}
```

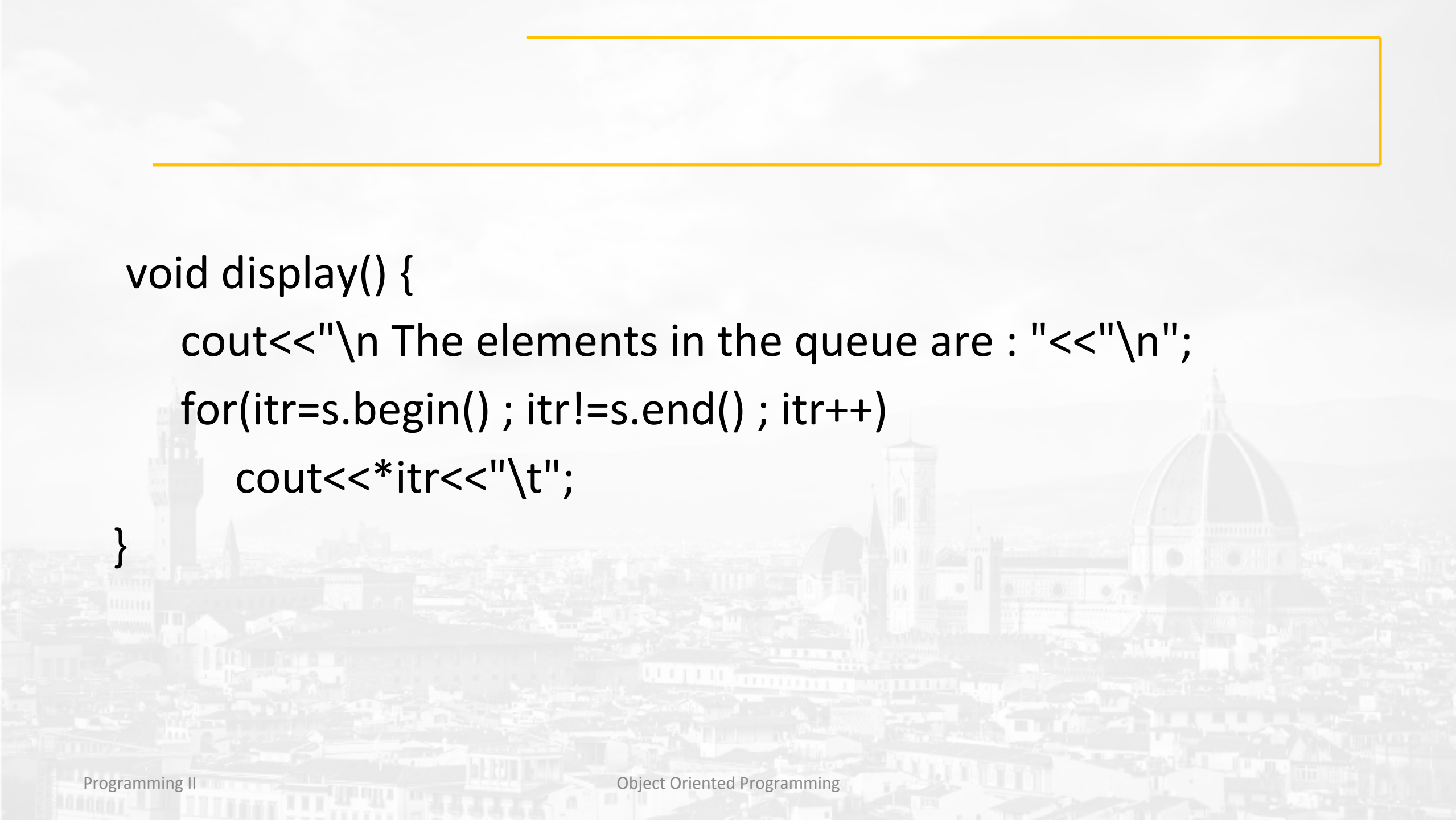


# Implement a Queue using STL List container

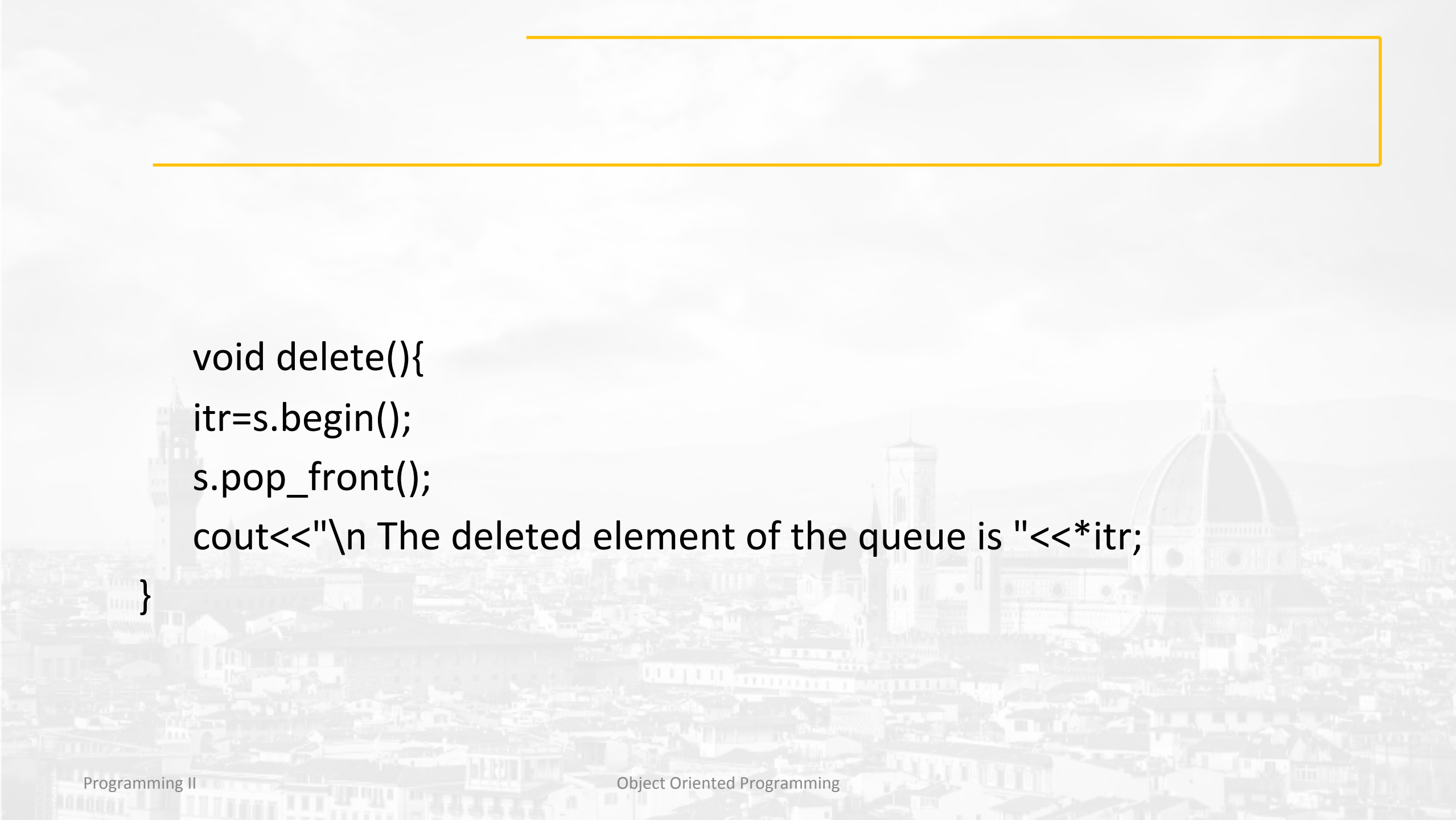
```
class Queue{  
    public :  
        int a;  list <int> s;  
        list <int> :: iterator itr;  
        void insert();  
        void display();  
        void delete();  
};
```



```
void insert() {  
    cout<<"\n Enter a number : “;  
    cin>>a;  
    s.push_back(a);  
}
```




```
void display() {  
    cout<<"\n The elements in the queue are : "<<"\n";  
    for(itr=s.begin() ; itr!=s.end() ; itr++)  
        cout<<*itr<<"\t";  
}
```



```
void delete(){  
    itr=s.begin();  
    s.pop_front();  
    cout<<"\n The deleted element of the queue is "<<*itr;  
}
```





```
int main(){  
    Queue q;  
    q.insert();  
    q.display();  
    q.delete();  
}
```

# deque

```
#include <deque>
int main() {
    deque<int> dq;
    dq.push_back(3);
    dq.push_front(1);
    dq.insert(dq.begin() + 1, 2);
    dq[2] = 0;
}
/* Output:
1 2 0
*/
```

# Containers Adaptors

- There are a few classes acting as wrappers around other containers , adapting them to a specific interface.
  - Stack-ordinary LIFO
  - Queue- single ended FIFO
  - Priority\_queue – the sorting criterion can be specified

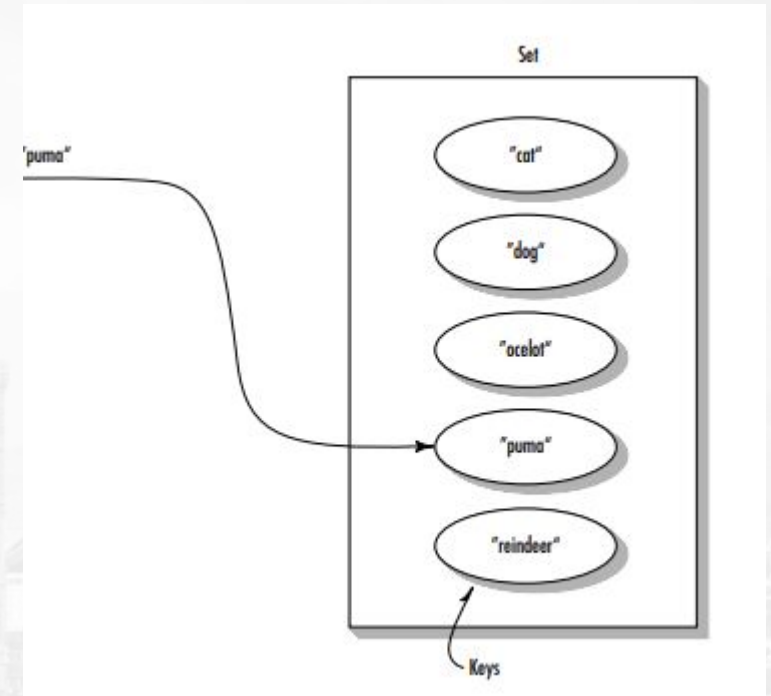
# Associative Containers

- In an associative container the items are not arranged in sequence. Instead they are arranged in a more complex way that makes it much faster to find a given item.
- This arrangement is typically a tree structure, although different approaches (such as hash tables) are possible.
- The speed of searching is the main advantage of associative containers.
- Searching is done using a key, which is usually a single value like a number or string. This value is an attribute of the objects in the container, or it may be the entire object.
- Eg set, multiset, map and multimap



# Set

- Sets are often used to hold objects of user-defined classes such as employees in a database.
- Sets can also hold simpler elements such as strings. The objects are arranged in order, and the entire object is the key.



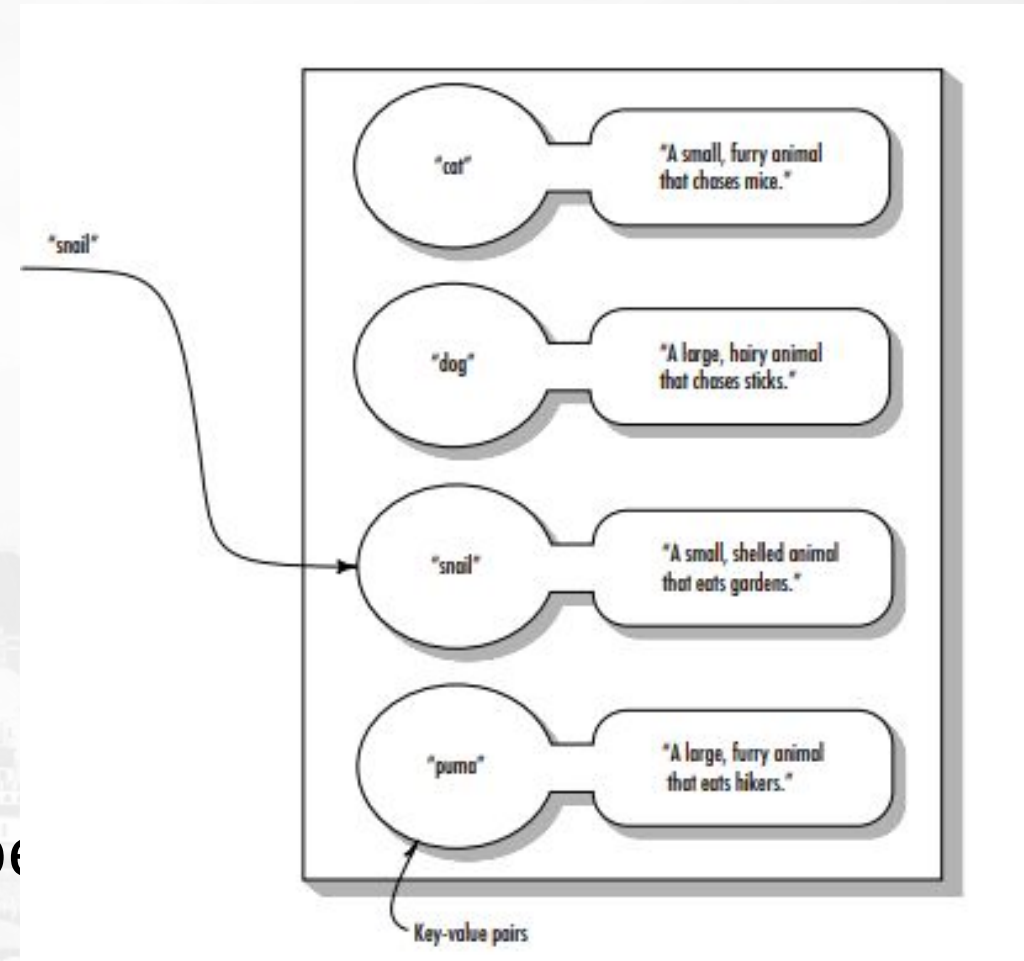
# Set – An example A set that stores objects of class string

```
#include <iostream>
#include <set>
#include <string>
using namespace std;
int main()
{ //array of string objects
string names[] = {"Juanita", "Robert", "Mary",
"Amanda", "Marie"};
//initialize set to array
set<string,less<string>>nameSet(names,names+5);
//iterator to set
set<string, less<string> >::iterator iter;
nameSet.insert("Yvette"); //insert more names
nameSet.insert("Larry");
nameSet.insert("Robert"); //no effect; already in set
nameSet.insert("Barry");
nameSet.erase("Mary"); //erase a name
//display size of set
cout << "\nSize=" << nameSet.size() << endl;
iter = nameSet.begin(); //display members of set
```

```
while( iter != nameSet.end() )
    cout << *iter++ << '\n';
string searchName; //get name from user
cout << "\nEnter name to search for: ";
cin >> searchName;
//find matching name in set
iter = nameSet.find(searchName);
if( iter == nameSet.end() )
    cout << "The name "<< searchName <<" is NOT in the
set.";
else
    cout << "The name " << *iter << " IS in the set.";
cout << endl;
return 0;
}
```

# Map

- A map stores pairs. A pair consists of a key object and a value object.
- The key object contains a key that will be searched for. The value object contains additional data.
- As in a set, the key objects can be strings, numbers, or objects of more complex classes. The values are often strings or numbers, but they can also be objects or even containers.



# Map- An ex where keys will be the names of states, and the values will be the populations of the states

```
#include <iostream>
#include <string>
#include <map>
using namespace std;
int main()
{
    string name;
    int pop;
    string states[] = { "Wyoming", "Colorado", "Nevada",
        "Montana", "Arizona", "Idaho" };
    int pops[] = { 470, 2890, 800, 787, 2718, 944 };
    map<string, int, less<string> > mapStates; //map
    map<string, int, less<string> >::iterator iter; //iterator
```

```
    for(int j=0; j<6; j++)
    {
        name = states[j]; //get data from arrays
        pop = pops[j];
        mapStates[name] = pop; //put it in map
    }
    cout << "Enter state: "; //get state from user
    cin >> name;
    pop = mapStates[name]; //find population
    cout << "Population: " << pop << ",000\n";
    cout << endl; //display entire map
    for(iter = mapStates.begin(); iter != mapStates.end(); iter++)
        cout << (*iter).first << ' ' << (*iter).second << ",000\n";
    return 0;
}
```