Unit-5

Unit-5: contents

Generic Programming with Templates: Need for Templates, Definition of class Templates, Function Template, Working of Function Templates, Class Template with more parameters, Function Template with more parameters.

Standard Template Library: Introduction to STL, STL Programming model,
containers, sequence container: vector, list; Associative containers: set, map; Algorithms: sort, search, find; Iterators.

Why do we need template?

To support Generic Programming

```
int maximum(int a,int b){
   return a>b?a:b;
}
float maximum(float a,float b){
   return a>b?a:b;
}
double maximum(double a,double b){
   return a>b?a:b;
}
```

Functionality is same but differ in the type of data

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

```
int main(){
   int i1=10,i2=20;
   float f1=2.3f,f2=4.1f;
   double d1=2.45,d2=1.71;
   cout<<"maximum of 10 and 20 is "<<maximum<int>(i1,i2)<<endl;
   cout<<"maximum of 2.3 and 4.1 is "<<maximum(f1,f2)<<endl;
   cout<<"maximum of 2.45 and 1.71 is "<<maximum(d1,d2)<<endl;
   return 0;
}
```

Templates

- Generic programming is a programming technique which is focused on the design , implementation and use of general algorithms. The term “general” means that a algorithm can be designed to accept a wide variety of types as long as they meet the algorithm’s requirements on its arguments.
- Templates are used to support generic programming.

C++ allows you to define *templates*—parameterized families of related functions or classes:

- a *function template* defines a group of statements for a function using a parameter instead of a concrete type
- a *class template* specifies a class definition using a parameter instead of a concrete type.

Contd..

	Function template	**Class Template**
Syntax	template<class identifier> function definition (or) template<typename identifier> function definition	template<class identifier> class definition (or) template<typename identifier> class definition
Example	template<class T> void exchange(T& a , T& b){ T temp; temp=a; a=b; b=temp; } ***T is known as type parameter***	template<class T> class MyClass{ T data; public: MyClass(T x):data(x){} void display(){ cout<<data<<endl; } };
Instantiation	int a=10,b=20; exchange<int>(10,20); cout<<a<<‘ “<<b<<endl; char ch1=‘x’,ch2=‘y’; exchange(ch1,ch2); cout<<ch1<<“ “<<ch2<<endl; double d1=3.12,d2=5.41; exchange(d1,d2); cout<<d1<<“ “<<d2<<endl;	MyClass<int> obj1(10); Obj1.display(); MyClass<double> obj2(11.5); Obj2.display(); MyClass<char> obj3(‘a’); Obj3.display();

The compiler determines the parameter type of `T` by the function arguments.

Example : function template

```
template<class T>
void exchange(T& a,T& b){
    T temp;
    temp=a;
    a=b;
    b=temp;
}

int main(){
    int a=10,b=20;
    exchange<int>(a,b);
    cout<<a<<" "<<b<<endl;
    char ch1='x',ch2='y';
    exchange(ch1,ch2);
    cout<<ch1<<" "<<ch2<<endl;
    double d1=2.31,d2=1.46;
    exchange(d1,d2);
    cout<<ch1<<" "<<ch2<<endl;
    return 0;
}
```

Example : class template

```
template<class T>
class MyClass{
    T data;
public:
    MyClass(T x): data(x) {}
    void display(){
            cout<<data<<endl;
     }
};

int main(){
    MyClass<int> obj1(10);
    obj1.display();

    MyClass<double> obj2(2.53);
    obj2.display();

    MyClass<char> obj3('a');
    obj3.display();
    return 0;
}
```

Class template with multiple parameters

```
template<class T1,class T2>
class A{
    T1 x;
    T2 y;
    public:
       A(T1 x1,T2 y1):x(x1),y(y1){}
     ~A(){}
       void display(){
          cout<<x<<" "<<y<<endl;
        }
};
int main(){
    A<char,int> obj1('a',97);
    A<string,float> obj2((string)"hindaloc",545.45f);
    A<string,long long> obj3((string)"william",8919756819);
    obj1.display();
    obj2.display();
    obj3.display();
    return 0;
}
```

```
a 97
hindaloc 545.45
william 8919756819
```

Example: Generic Sort

A function template can have non-template parameters

```
template<class T>
void Sort(T* arr,int n){
   for(int i=1;i<=n-1;i++){
      for(int j=0;j<n-i;j++){
         if(arr[j]>arr[j+1]){
            T t=arr[j];
            arr[j]=arr[j+1];
            arr[j+1]=t;
         }
      }
   }
}

template<class T>
void print(T* arr,int n){
   for(int i=0;i<n;i++)
     cout<<arr[i]<<" ";
   cout<<endl;
}
```

```
int main(){
int a[]={2,1,5,4,3};
   double b[]={4.1,2.1,3.2,7.1,5.7};
   char c[]={'x','a','u','c','m','e'};
   Sort<int>(a,5);
   print(a,5);
   Sort(b,5);
   print(b,5);
   Sort(c,6);
   print(c,6);
   return 0;
}
```

```
1 2 3 4 5
2.1 3.2 4.1 5.7 7.1
a c e m u x
```

Example: to find the reverse of a string using generic class

```
template<class T>
class stack{
        T data[100];
        int tos;
        public:
          stack():tos(-1){}
          ~stack(){}
          void push(const T& ele){
               data[++tos]=ele;
          }
          void pop(){
               tos--;
          }
          const T& top() {
               return data[tos];
          }
          bool isEmpty(){
               return tos==-1;
          }
};
```

```
int main(){
        char str[]="ABCDE";
        stack<char> s;
        for(unsigned int i=0;i<strlen(str);i++)
             s.push(str[i]);
        cout<<"reveresed string"<<endl;
        while(!s.isEmpty()){
          cout<<s.top();
          s.pop();
        }
   return 0;
}
```

Example: to check string is palindrome or not

```
template<class T>
class stack{
        T data[100];
        int tos;
        public:
          stack():tos(-1){}
          ~stack(){}
          void push(const T& ele){
                data[++tos]=ele;
          }
          void pop(){
                tos--;
          }
          const T& top() {
                return data[tos];
          }
          bool isEmpty(){
                return tos==-1;
          }
};
```

```
int main(){
   char str[20];
   bool flag=false;
   stack<char> s;
   cout<<"enter string:";
   cin>>str;
   for(unsigned int i=0;i<strlen(str);i++)
      s.push(str[i]);
   for(unsigned int i=0;i<strlen(str);i++){
            if(!s.empty()){
               if(s.top()==str[i])
                  s.pop();
               else{
                  flag=true;
                  break;
               }
         }
   }
   if(!flag)
      cout<<"palindrome"<<endl;
   else
      cout<<"not palindrome"<<endl;
   return 0;
}
```

C++ Standard Library

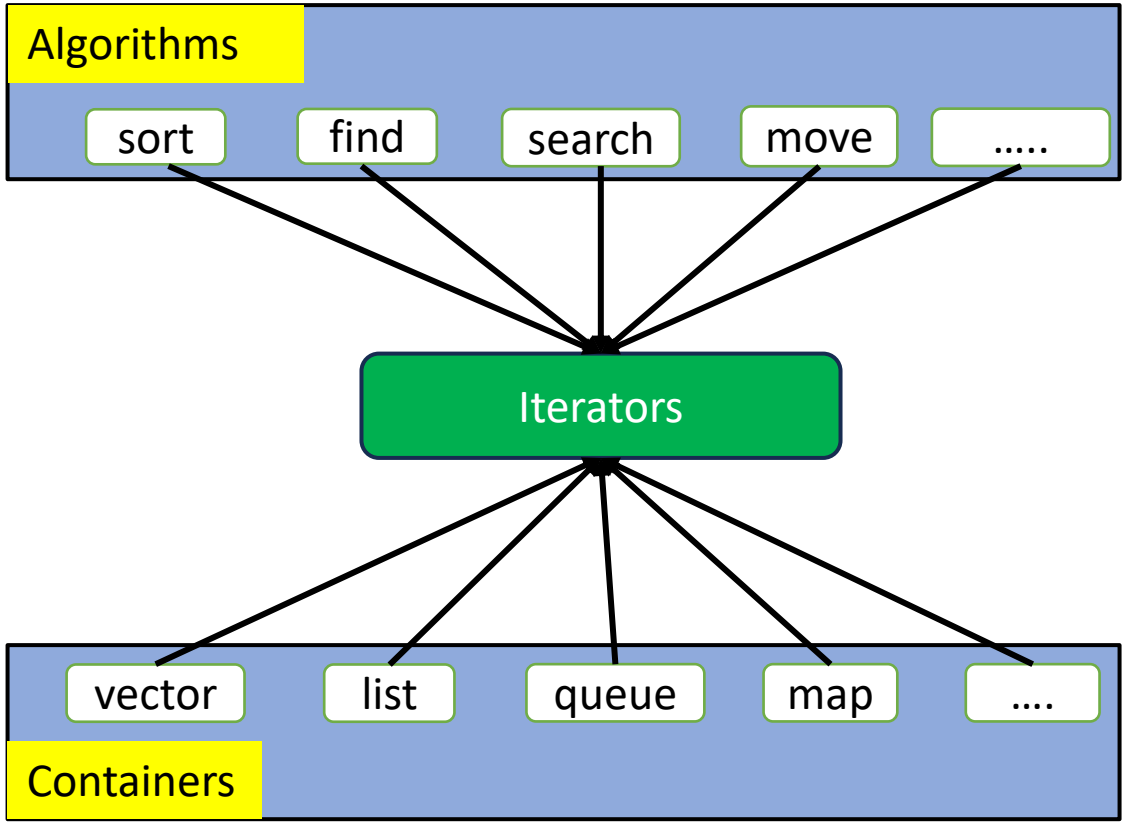
library	classes/functions/macros/manifest constants
iostream fstream	Stream input and output for standard I/O like cout,cin,endl,…etc.
string	Manipulation of string objects Relational operators , IO operators, iterators etc.
STL	
Containers	vector, list, stack, queue, deque, priority_queue, set, multiset, map, multimap **C++11**: array, forward_list, unordered_set/multiset/map/multimap
Iterators	begin & end, rbegin& rend. **C++ 11**: cbegin & cend, crbegin & crend.
Algorithm	sort,binary_search,min,max,for_each,find,find_if,count,search,copy,move,swap, replace,fill, generate, remove, reverse, rotate, merge
numeric	Accumulate,adjacent_difference,inner_product,and partial_sum
Functions	Equal_to,not_equal_to,greater,greater_equal,less,less_equal; plus, minus , multiplies, divides,modulus; logical_and,logical_not,logical_or. **C++11**: bit_and,bit_or,bit_xor

STL Programming Model

It consists three components

- ❖ Algorithms
- ❖ Containers
- ❖ Iterators

- Algorithms manipulate data but do not know about containers.
- Containers store data , but do not know about Algorithms
- Algorithms and Containers interact through Iterators.
- Each Container has its own iterator types.



Containers

- A container is a holder object that stores a collection of other objects.
- The containers manages the storage space for its elements and provide *member functions* to access them either directly or through *iterators*
- Containers are divided into 3 categories
  - Sequence containers
  - Associative containers
  - Container adapters
- Sequence containers maintain the ordering of inserted elements at the time of insertion.
  Examples : vector, array, deque, list(DLL), forward_list(SLL)
- Associative containers maintain the pre-defined insertion order for the inserted elements. They can be unordered also.
  Examples : map , unordered_map, set and unordered_set. (for single instance)
  multimap,unorderd_multimap,multiset and unordered_multiset (for multiple instances)
- Container adapters are not containers but uses the container class internally .These are used to restrict the interface for simplicity and clarity.
  Examples: stack, queue, priorirty_queue

***These are not used with c++ standard Algorithms because they don't support iterators*.**

Sequence containers

Sequence containers implement data structures which can be accessed sequentially.

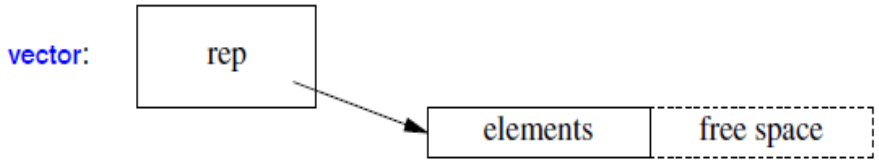
array (C++11)	static contiguous array
vector	dynamic contiguous array
deque	double-ended queue
forward_list (C++11)	singly-linked list
list	doubly-linked list

vector

- vector is a sequence container that encapsulates dynamic size arrays.
- Available in <vector> package

template < class T, class Allocator = std::allocator<T> > class vector;

vector representation



constructing a vector object

- vector<int> v; // creates an empty vector
- vector<int> v(3); // creates a 3-element vector initialized to 0, v={0, 0, 0}
- vector<int> v(3, 2); // creates a 3-element vector with 3 times value 2 , v={2, 2, 2}
- vector<int> v{2}; // creates a 1-element vector with value 2 ,v={2}
- Vector<int> v{5, 2, 4, 6}; // creates a 4-element vector with the given values v={5, 2, 4, 6}

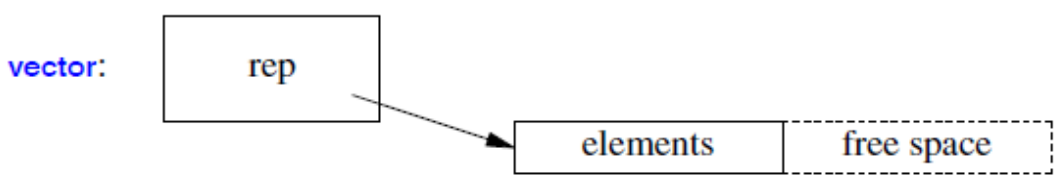
Element access		**v={5, 2, 6, 1, 9, 7}**
at	access specified element with bounds checking	v.at(2) gives 6
operator[]	access specified element	v[2] gives 6
front	access the first element	v.front() gives 5
back	access the last element	v.back() gives 7
data	Direct access to the underlying array	v.data() gives the pointer to the start of the array

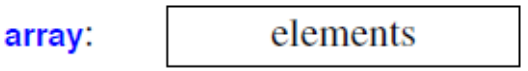
Iterators	
begin **cbegin**	returns an iterator to the beginning
end **cend**	returns an iterator to the end
rbegin **crbegin**	returns a reverse iterator to the beginning
rend **crend**	returns a reverse iterator to the end
Capacity	
empty	checks whether the container is empty
size	returns the number of elements
max_size	returns the maximum possible number of elements
reserve	reserves storage
capacity	returns the number of elements that can be held in currently allocated storage

Modifiers		
insert	*iterator insert(const_iterator pos , const T& value);* inserts value before pos *iterator insert(const_iterator pos, size_type count, const T& value);* Inserts count copies of the value before pos.	vector<int> v = {8, 4, 5, 9} auto it=v.begin(); v.insert(it,20) v = {20,8,4,5,9} v.insert(it,2,15) v = {15,15,20,8,4,5,9} v.insert(v.end(),{6,11}) v= {15,15,20,8,4,5,9,6,11}
emplace (C++11)	constructs element in-place	Similar to insert but it constructs new object and insert
push_back	*void push_back(const T& value)* adds an element to the end	vector<int> v = {8, 4, 5, 9} v.push_back(10); // v = {8, 4, 5, 9,10}
emplace_back (C++11)	constructs an element in-place at the end	Similar to push_back but it constructs new object and append at end
pop_back	*void pop_back();* removes the last element	vector<int> v = {8, 4, 5, 9} v.pop_back(); // v = {8, 4, 5}
erase	*iterator erase(iterator pos);* erases elements *iterator erase(iterator first, iterator last);* Erases elements from first to up to last(excluding last)	vector<int> v = {8, 4, 5, 9, 10, 12, 7, 2} v.erase(v.begin()+2); // v = {8, 4, 9, 10, 12, 7, 2} v.erase(v.begin()+2,v.begin()+5); // v = {8, 4, 9, 10, 12, 7, 2}
clear	clears the contents	v.clear() // makes the vector empty
swap	void swap(vector& other); swaps the contents of one container with another	vector<int> a1{1, 2, 3} vector<int> a2{4, 5}; a1.swap(a2); a1={4,5} and a2={1,2,3}

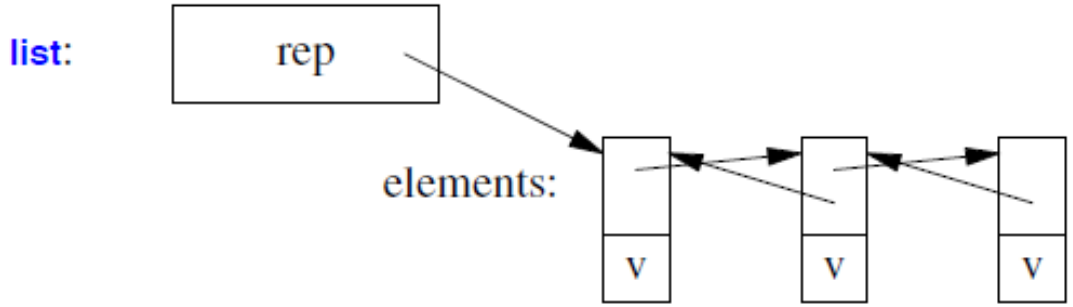
Container Representations

For a **vector**, the element data structure is most likely an array:

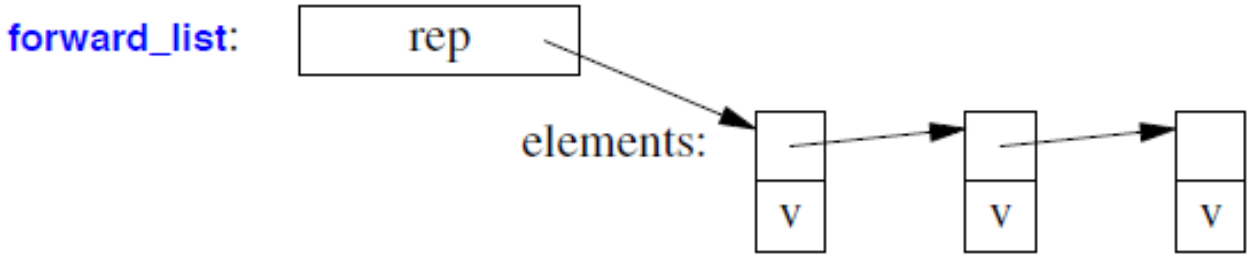




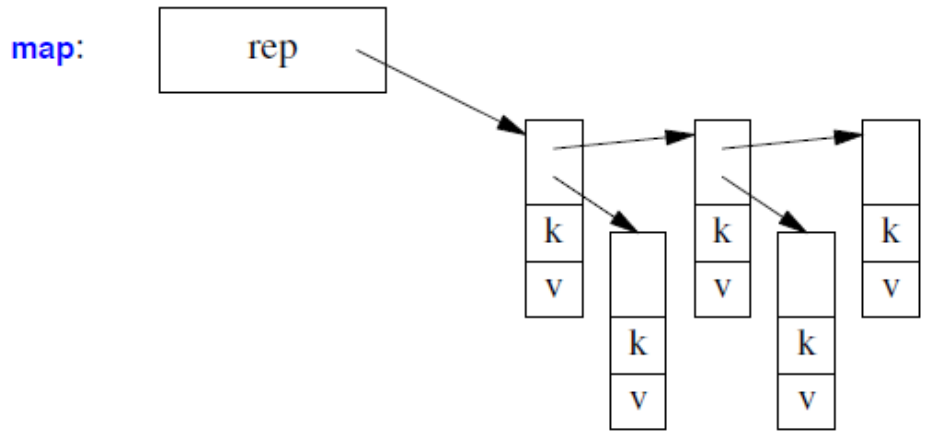
A **list** is most likely represented by a sequence of links pointing to the elements and the number of elements:



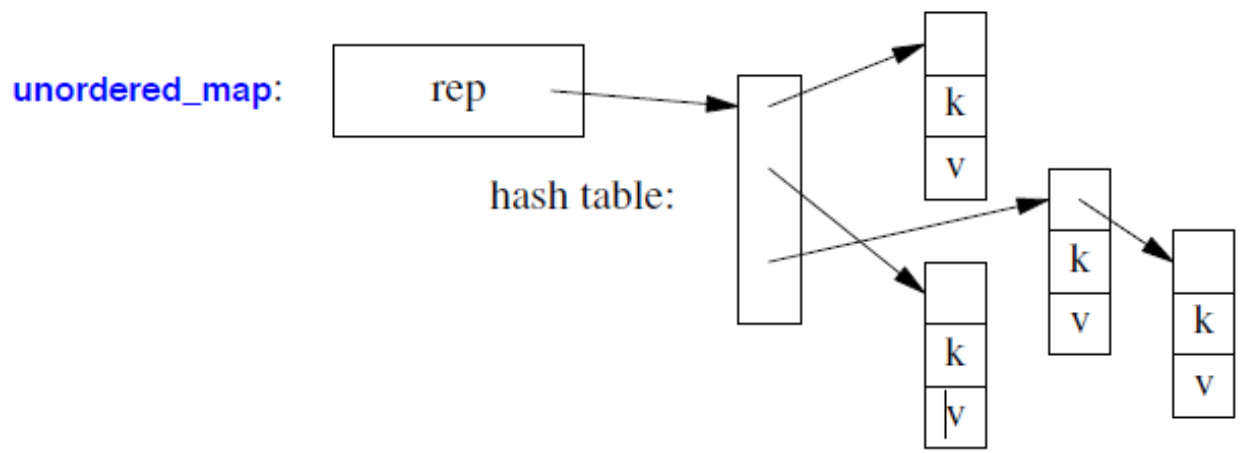
A **forward_list** is most likely represented by a sequence of links pointing to the elements:



A map is most likely implemented as a (balanced) tree of nodes pointing to (key,value) pairs:



An unordered_map is most likely implemented as a hash table:



- For sequences of small elements, a **vector** can be an excellent representation for a data structure requiring list operations.
- When inserting and erasing elements of a **vector**, elements may be moved. In contrast, elements of a list or an associative container do not move when new elements are inserted or other elements are erased.

Container - list

- list is a container that supports constant time insertion and removal of elements from anywhere in the container.
- It is usually implemented as a doubly-linked list, so, random access is not supported.
- It supports bidirectional iteration.
- available in <list> package

```
template< class T, class Allocator = std::allocator<T> >
class list;
```

Element access	
front	access the first element
back	access the last element

Modifiers
contains modifiers of vector and push_front() emplace_front() pop_front() emplace_back()

Iterators	
begin **cbegin**	returns an iterator to the beginning
end **cend**	returns an iterator to the end
rbegin **crbegin**	returns a reverse iterator to the beginning
rend **crend**	returns a reverse iterator to the end
Capacity	
empty	checks whether the container is empty
size	returns the number of elements
max_size	returns the maximum possible number of elements

Operations		Example
merge	merges two sorted lists	list<int> l1 = {1,2,3} list<int> l2 = {4,5,6} l1.merge(l2); l1={1,2,3,4,5,6}
splice	moves elements from another list	list<int> l1 = {1,2,3} list<int> l2 = {4,5,6} auto it1=l1.begin(); advance(it1,2) l1.splice(it1,l2) // l1={1,2,4,5,6,3}, l2={}
remove **remove_if**	removes elements satisfying specific criteria	list<int> l1 = {1,2,2,3,1,2} l1.remove(1); // l1={2,2,3,2} l1.remove_if(l1.begin(),l1.end(),[](int n){ return n%2==0}); // l1={1,3}
reverse	reverses the order of the elements	list<int> l1 = {1,2,3} l1.reverse(); // l1={3,2,1}
unique	removes consecutive duplicate elements	list<int> l1 = {1,2,2,3,1,2} l1.unique(); // l1 = {1,2,3,1,2}
sort	sorts the elements	list<int> l1={4,1,2,5,3} l1.sort(); // l1= {1, 2, 3, 4, 5}

Set

- set is an associative container that contains a sorted set of unique objects of type Key.
- sorting is done using the key comparison function.
- search, removal, and insertion operations have logarithmic complexity.
- sets are implemented as Red–black trees.

```
template<
    class Key,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<Key>
> class set;
```

```
set<int> s;
```

```
set<int> s={1,5,2,3,4};
  for(auto i : s)
     cout<<i<<" ";
  cout<<endl;
```

Prints 1 2 3 4 5

Set Operations	**Output**	**Description**
set<int> s={1,5,2,3,4, 1}; for(auto i:s) cout<<i<<" "; cout<<endl;	Prints 1 2 3 4 5	Creates set s with ascending order
set<int,greater<int>> s={1,5,2,3,4, 1}; for(auto i:s) cout<<i<<" "; cout<<endl;	Prints 5 4 3 2 1	Creates set s with descending order
s.insert(7) for(auto i:s) cout<<i<<" "; cout<<endl;	Prints 7 5 4 3 2 1	Inserts element 7 into s
s.erase(3) for(auto i:s) cout<<i<<" "; cout<<endl;	Prints 7 5 4 2 1	Removes element 3 from s

Map

- map is a sorted associative container that contains key-value pairs with unique keys.
- Keys are sorted by using the comparison function Compare.
- Search, removal, and insertion operations have logarithmic complexity.
- Maps are implemented as Red–black trees

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

<table>
<tr>
<td>

```
map<char,int> m{{'a',3},{'c',1},{'b',2}};
   for(auto [k,v]: m){
      cout<<"( "<<k<<" , "<<v<<" ) "<<endl;
   }
```

</td>
<td>Prints
(a , 3)
(b , 2)
(c , 1)</td>
<td rowspan="2">Creates a map m with ascending order based on key</td>
</tr>
<tr>
<td>

```
map<char,int> m{{'a',97},{'c',99},{'b',98}};
   for(auto it=m.begin();it!=m.end();++it)
      cout<<"( "<<it->first<<" , "<<it->second<<" ) "<<endl;
   }
```

</td>
<td>Prints
(a , 3)
(b , 2)
(c , 1)</td>
</tr>
</table>

m.insert(pair<char,int>(‘d’,4)) for(auto [k,v]: m){ cout<<"("<<k<<" , "<<v<<") "<<endl; }	Prints (d, 4) (c , 1) (b , 2) (a , 3)	Insert an element into map

Algorithm package <algorithm>

```
template< class ForwardIt, class T >
bool binary_search( ForwardIt first, ForwardIt last, const T& value );
```

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
int main()
{
   vector<int> haystack{1, 3, 4, 5, 9};
   vector<int> needles{1, 2, 3};

   for (const auto needle : needles)
   {
      cout << "Searching for " << needle << '\n';
      if (binary_search(haystack.begin(), haystack.end(), needle))
         cout << "Found " << needle << '\n';
      else
         cout << "No dice!\n";
   }
   return 0;
}
```

Sort()

default

```
template <class RandomAccessIterator>
void sort (RandomAccessIterator first, RandomAccessIterator last);
```

custom

```
template <class RandomAccessIterator, class Compare>
void sort (RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

Sorts the elements in the range [first,last) into ascending order.

Example : for using sort function

```
#include <iostream> // std::cout
#include <algorithm> // std::sort
#include <vector> // std::vector
using namespace std;
bool myfunction (int i,int j) { return (i<j); }
class myclass { public:bool operator() (int i,int j) { return (i<j);} } myobject;
int main ()
{
    int myints[] = {32,71,12,45,26,80,53,33};
    std::vector<int> myvector (myints, myints+8); // 32 71 12 45 26 80 53 33
    //using default comparison (operator <):
    std::sort (myvector.begin(), myvector.begin()+4); //(12 32 45 71)26 80 53 33
    // using function as comp
    std::sort (myvector.begin()+4, myvector.end(), myfunction); // 12 32 45 71(26 33 53 80)
    // using object as comp
    std::sort (myvector.begin(), myvector.end(), myobject); //(12 26 32 33 45 53 71 80)
    // print out content:
    cout << "myvector contains:";
    for (vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
        cout << ' ' << *it;
    cout << endl;
    return 0;
}
```

find()

returns an iterator to the first element in the range[first,last) that compares equal to *val*. If no such element is found, the function returns *last*.

Syntax

template <class InputIterator, class T>
InputIterator find (InputIterator first, InputIterator last, const T& val);

```
// find example
#include <iostream> // for cout
#include <algorithm> // for find
#include <vector> //  for vector
using namespace std;
int main () {
	// using std::find with array and pointer:
	int myints[] = { 10, 20, 30, 40 };
	int * p;
	p = find (myints, myints+4, 30);
	if (p != myints+4)
		cout << "Element found in myints: " << *p << endl;
	else
		cout << "Element not found in myints\n";
	// using std::find with vector and iterator:
	vector<int> myvector (myints,myints+4);
	vector<int>::iterator it;
	it = find (myvector.begin(), myvector.end(), 30);
	if (it != myvector.end())
		cout << "Element found in myvector: " << *it << endl;
	else
		cout << "Element not found in myvector“<<endl;
	return 0;
}
```

Search()

Searches the range [first1,last1) for the first occurrence of the sequence defined by[first2,last2) and returns an iterator to its first element , or *last1* if no occurrences are found.

default

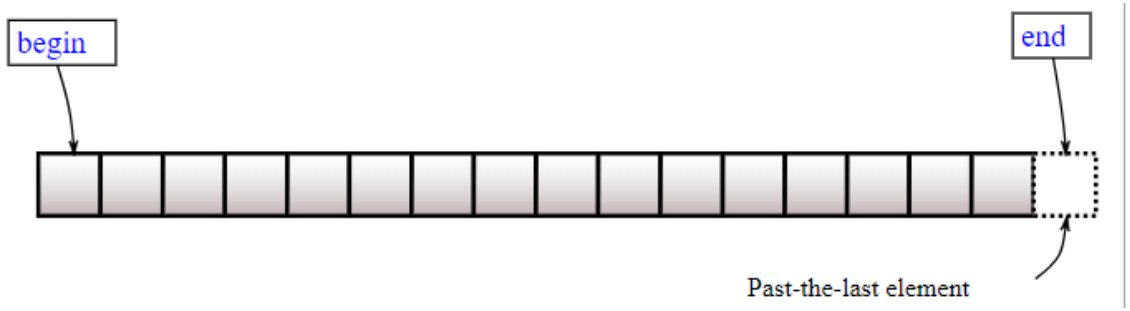
```
template <class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 search (ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2);
```

custom

```
template <class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
ForwardIterator1 search (ForwardIterator1 first1,
                         ForwardIterator1 last1,
                         ForwardIterator2 first2,
                         ForwardIterator2 last2,
                         BinaryPredicate);
```

```
#include <iostream> //  for cout
#include <algorithm> // for search
#include <vector> // for vector
using namespace std;
bool mypredicate (int i, int j) { return (i==j); }
int main () {
        vector<int> haystack;      // set some values: haystack: 10 20 30 40 50 60 70 80 90
        for (int i=1; i<10; i++)   haystack.push_back(i*10);
        // using default comparison:
        int needle1[] = {40,50,60,70};
        vector<int>::iterator it;
        it = search (haystack.begin(), haystack.end(), needle1, needle1 + 4);
        if (it!=haystack.end())
                cout << "needle1 found at position " << (it-haystack.begin()) << '\n’;
        else
                cout << "needle1 not found\n";
        // using predicate comparison:
        int needle2[] = {20,30,50};
        it = std::search (haystack.begin(), haystack.end(), needle2, needle2 + 3, mypredicate);
        if (it!=haystack.end())
                cout << "needle2 found at position " << (it-haystack.begin()) << '\n’;
        else
                cout << "needle2 not found\n";
        return 0;
}
```

Iterator

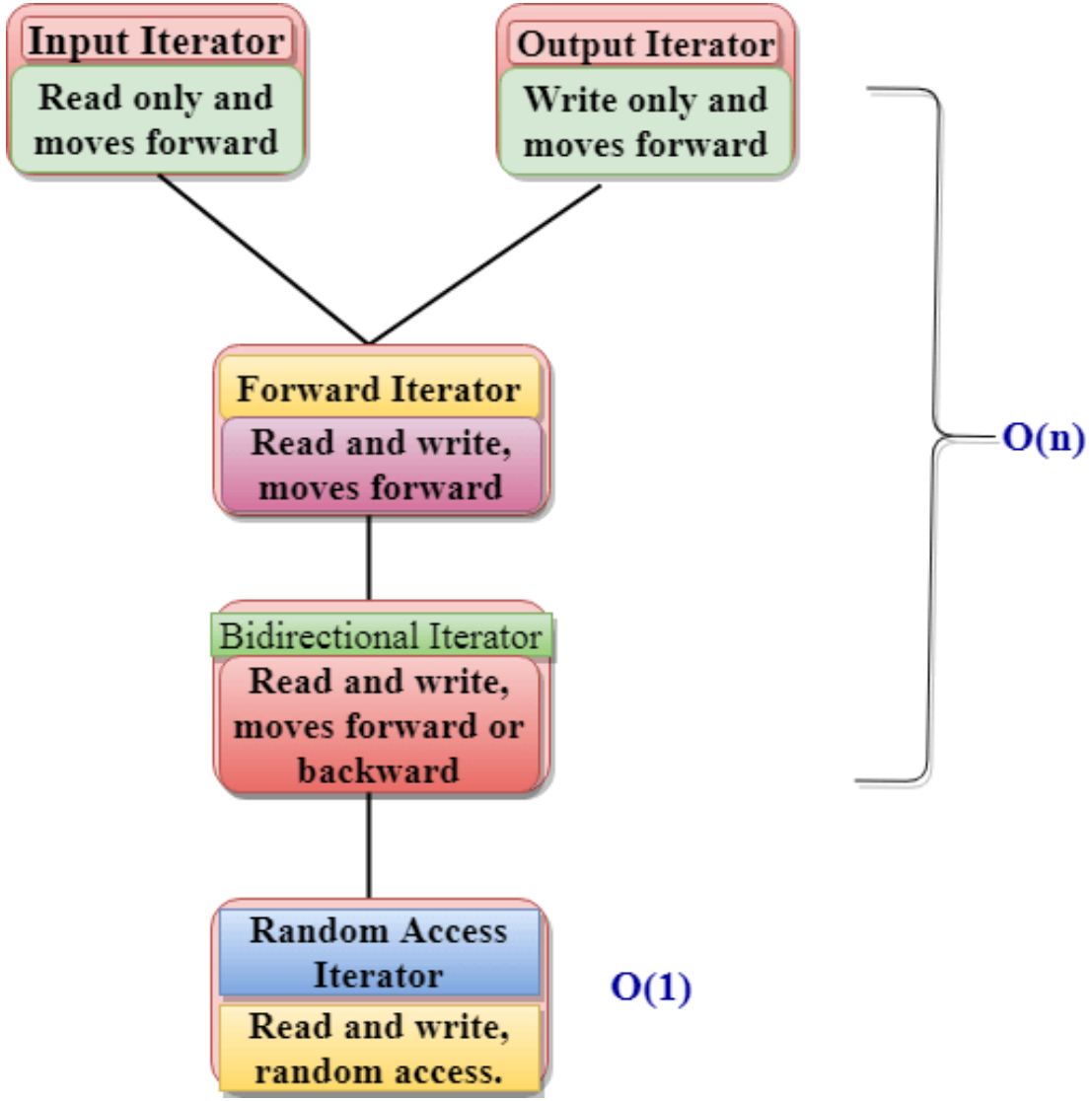


- An *iterator* is any object that, pointing to some element in a range of elements (such as an array or a container), has the ability to iterate through the elements of that range using a set of operators (like ++ and *)

- The C++ Standard Library containers all provide iterators so that algorithms can access their elements in a standard way without having to be concerned with the type of container the elements are stored in

Method	Description
begin	Iterator to begin
End	Iterator to end
prev	Get iterator to previous element
next	Get iterator to next element
advance	Advance(increment) iterator
distance	Returns the distance between(no of elements) iterators

Types of Iterators



STL CONTAINER	ITERATOR SUPPORTED
Vector	Random-Access
List	Bidirectional
Dequeue	Random-Access
Map	Bidirectional
Multimap	Bidirectional
Set	Bidirectional
Multiset	Bidirectional
Stack	Does not support any iterator
Queue	Does not support any iterator
Priority-Queue	Does not support any iterator