C++ STL - Cheat Sheet

This C++ STL cheatsheet covers a wide range of topics from basic STL like **vectors**, **hashmaps**, sets, etc., to advanced concepts like **functors**, **iterators**, and so on. It is designed for programmers who want to quickly read through key concepts along with the syntax and related examples.

Introduction

The Standard Template Library (STL) is a **C++ library** that contains all **class** and **function** templates. It is used for implementing the basic Data Structures (like **Hashset**, **Heap**, **List**, etc.) and functions (like math, algorithms, etc.), and contains all **containers** available in C++ programming language.

Components of STL

Standard Template Library contains of several containers and functions, and can be categorized in the following manner −

- **Containers**
- **Functors**
- **Algorithms**
- **Iterators**

STL Containers

The STL container template class contains data structures like **Dynamic Arrays** (or Vectors, Lists), Hashmaps, Hashsets, **Trees**, **Linked Lists**, etc. These are used for storing and performing operations on the data.

The container template has the following components −

- **Sequential Containers**
- **Container Adapters**
- **Associative Containers**
- **Unordered Containers**

Each container has its respective header file, which can be included at the start of the program. For example, the std::vector is included in the #include<vector> library.

Sequential Containers

The sequential containers implement the data structures with sequential access. These include –

- Vector
- List
- Deque
- Array
- Forward List

Container Adapters

The container adapters implement data structures like queues, stacks, etc by providing different interfaces for sequential containers. These include –

- Stack
- Queue
- Priority Queue

Associative Containers

The associative containers are used to store ordered data that can be quickly searched using their key value. These include –

- Set
- Multiset
- Map
- Multimap

Unordered Containers

The unordered containers are similar to associative containers but the only difference is that they don't store sorted data. Still, these containers provide quick search time using key-value pairs. They are –

- Unordered Set
- Unordered Multiset
- Unordered Map

- Unordered Multimap

Now that we have established a learning graph of all containers, we will briefly explain each container in the STL with an exemplar code −

Vector in STL

The vector is initialized as a dynamic array at runtime, and its size is variable. It can be found in the C++ <vector> header file.

Syntax

```
vector<data_type> vec1; //1D vector
vector<vector<data_type>> vec2; //2D vector
vector<container_type<data_type>> vec3; //2D vector of other container
vector<data_type> vec1(n); //vector of size n
vector<data_type> vec1(n,k); // vector of size n, with each element=k
```

There are different functions in the **vector template**. These are explained briefly in the table below −

S.No.	Function	Function Explanation	TC
1.	**begin()**	Returns an iterator to the first element.	O(1)
2.	**end()**	Returns an iterator to the theoretical element after the last element.	O(1)
3.	**size()**	Returns the number of elements present.	O(1)
4.	**empty()**	Returns true if the vector is empty, false otherwise.	O(1)
5.	**at()**	Return the element at a particular position.	O(1)
6.	**assign()**	Assign a new value to the vector elements.	O(n)
7.	**push_back()**	Adds an element to the back of the vector.	O(1)
8.	**pop_back()**	Removes an element from the end.	O(1)
9.	**insert()**	Insert an element at the specified position.	O(n)
10.	**erase()**	Delete the elements at a specified position or range.	O(n)
11.	**clear()**	Removes all elements.	O(n)

Here, TC indicates time complexity of different member functions of the vector template. For more information on time complexity, visit this article − **Time complexity**.

Example

</> Open Compiler

```
// C++ program to illustrate the vector container
#include <iostream>
#include <vector>
using namespace std;

int main(){
   int n=2;
   vector<int> vec1 = { 1, 2, 3, 4, 5 };
   vector<int> vec2(n,0);

   vector<vector<int>> vec3(n,vector<int>(2*n,1));

   vec1.push_back(6);

   cout << "vec1: ";
   for (int i = 0; i < vec1.size(); i++) {
      cout << vec1[i] << " ";
   }
   cout << endl<<"vec2: ";
   for (int i = 0; i < vec2.size(); i++) {
      cout << vec2[i] << " ";
   }
   cout << endl;

   vec1.erase(vec1.begin() + 4);

   cout << "vec1 after erasing: ";
   for (auto i = vec1.begin(); i != vec1.end(); i++) {
      cout << *i << " ";
   }
   cout << endl;

   cout << "vec3:-" << endl;
   for (auto i : vec3) {
      for (auto j : i) {
         cout << j << " ";
      }
```

```
      cout << endl;
   }
   cout << endl;

   vector<pair<int,int>> vec4;
   vec4.push_back({2,3});
   vec4.push_back({4,3});

   cout<<"vector of pairs, vec4 : "<<endl;
   for(auto i: vec4){
      cout<<i.first<<" "<<i.second<<endl;
   }
   return 0;
}
```

Output

```
vec1: 1 2 3 4 5 6
vec2: 0 0
vec1 after erasing: 1 2 3 4 6
vec3:-
1 1 1 1
1 1 1 1

vector of pairs, vec4 :
2 3
4 3
```

List in STL

The list container is initialized as a **doubly linked list**, whereas for implementing a **singly linked list**, we use a **forward_list**. It can be found in the C++ <list> header file.

Syntax

```
list<data_type> list1;
```

There are different functions in the **list template**. These are explained briefly in the table below –

S.No.	Function	Function Explanation	TC
1.	**begin()**	Return the iterator to the first element.	O(1)
2.	**end()**	Returns an iterator to the theoretical element after the last element.	O(1)
3.	**size()**	Returns the number of elements in the list.	O(1)
4.	**push_back()**	Adds one element at the end of the list.	O(1)
5.	**pop_back()**	Removes a single element from the end.	O(1)
6.	**push_front()**	Adds a single element to the front of the list.	O(1)
7.	**pop_front()**	Removes a single element from the front.	O(1)
8.	**insert()**	Inserts an element at the specified position.	O(n)
9.	**erase()**	Deletes the element at the given position.	O(n)
10.	**remove()**	Removes all the copies of the given elements from the list.	O(n)

Example

</> Open Compiler

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

int main(){
   list<int> list1 = { 1, 5, 9, 1, 4, 6 };

   cout << "List1 first and last element: " << list1.front() <<" "<<list1.b

   // adding element
   list1.insert(list1.begin(), 5);

   // deleting element
   list1.erase(list1.begin());

   // traversing list1
   cout << "list1: ";
```

```
    for (auto i = list1.begin(); i != list1.end(); i++) {
       cout << *i << " ";
    }
    cout << endl;

    return 0;
}
```

Output

```
List1 first and last element: 1 6
list1: 1 5 9 1 4 6
```

Deque in STL

The deque container is initialized as a **doubly ended queue**, where elements can be pushed and popped from both ends of the queue. It can be found in the C++ <deque> header file.

Syntax

```
deque<data_type> dq1;
```

There are different functions in the **deque template**. These are explained briefly in the table below –

S.No.	Function	Function Explanation	TC
1.	**begin()**	Returns iterator to the first element.	O(1)
2.	**end()**	Returns an iterator to the theoretical element after the last element.	O(1)
3.	**at()**	Access specified element.	O(1)
4.	**[ ]**	Access element at the given index.	O(1)
5.	**front()**	Returns the first element.	O(1)
6.	**back()**	Returns the last element.	O(1)
7.	**size()**	Returns the number of elements.	O(1)
8.	**push_back()**	Add the elements at the end.	O(1)
9.	**pop_back()**	Removes the elements from the end.	O(1)

S.No.	Function	Function Explanation	TC
10.	push_front()	Add the elements at the front.	O(1)
11.	pop_front()	Removes the element from the front.	O(1)

Example

</> Open Compiler

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

int main(){

   deque<int> dq = { 1, 2, 3, 4, 5 ,6, 8 };
   cout<<"Initial Deque: "<<endl;
   for (auto i : dq) {
      cout << i << " ";
   }
   cout<<endl;
   dq.push_front(dq.back());
   dq.pop_back();
   for (auto i : dq) {
      cout << i << " ";
   }
   cout<<endl;

   dq.push_front(dq.back());
   dq.pop_back();
   for (auto i : dq) {
      cout << i << " ";
   }
   cout<<endl;

   dq.pop_back();
   dq.pop_front();
   for (auto i : dq) {
      cout << i << " ";
   }
```

```
    cout<<endl;

    dq.push_front(11);
    dq.push_back(99);

    for (auto i : dq) {
       cout << i << " ";
    }

    return 0;
}
```

Output

```
Initial Deque:
1 2 3 4 5 6 8
8 1 2 3 4 5 6
6 8 1 2 3 4 5
8 1 2 3 4
11 8 1 2 3 4 99
```

Stack in STL

The stack container is initialized as a **LIFO container**, where elements can be pushed to the top, and popped from the top. Hence, the last element to enter is the first one to exit from the container. It can be found in the C++ <stack> header file.

Syntax

```
stack<data_type> s1;
```

There are different functions in the **stack template**. These are explained briefly in the table below −

S.No.	Function	Function Explanation	TC
1.	**empty()**	Returns true if the stack is empty, false otherwise.	O(1)
2.	**size()**	Returns the number of elements in the stack.	O(1)
3.	**top()**	Returns the top element.	O(1)

S.No.	Function	Function Explanation	TC
4.	push(x)	Push one element in the stack.	O(1)
5.	pop()	Removes one element from the stack.	O(1)

Example

</> Open Compiler

```
// C++ Program to illustrate the stack
#include <bits/stdc++.h>
using namespace std;

int main(){
   stack<int> s;

   s.push(2);
   s.push(9);
   s.push(3);
   s.push(1);
   s.push(6);

   cout << "Top is: " << s.top() << endl;

   while (!s.empty()) {
      cout<<"size is: "<<s.size()<<" ";
      cout << "element is: "<<s.top() << endl;
      s.pop();
   }
   return 0;
}
```

Output

```
Top is: 6
size is: 5 element is: 6
size is: 4 element is: 1
size is: 3 element is: 3
```

```
size is: 2 element is: 9
size is: 1 element is: 2
```

Queue in STL

The queue container is initialized as a **FIFO container**, where elements can be pushed to the front, and popped from the front. Hence, the first element to enter is the first one to exit from the container. It can be found in the C++ <queue> header file.

Syntax

```
queue<data_type> q1;
```

There are different functions in the **queue template**. These are explained briefly in the table below –

S.No.	Function	Function Explanation	TC
1.	**empty()**	Returns true if the queue is empty, otherwise false.	O(1)
2.	**size()**	Returns the number of items in the queue.	O(1)
3.	**front()**	Returns the front element.	O(1)
4.	**back()**	Returns the element at the end.	O(1)
5.	**push()**	Add an item to the queue.	O(1)
6.	**pop()**	Removes an item from the queue.	O(1)

Example

</> Open Compiler

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

int main(){
   queue<int> q;
   q.push(1);
   q.push(1);
   q.push(6);
```

```
    q.push(1);

    cout << "Front element: " << q.front() << endl;
    cout << "Back element: " << q.back() << endl;

    cout << "q: ";
    int size = q.size();

    for (int i = 0; i < size; i++) {
       cout << q.front() << " ";
       q.pop();
    }

    return 0;
}
```

Output

```
Front element: 1
Back element: 1
q: 1 1 6 1
```

Hash-Set/Set in STL

The set container is initialized as unique element storage data structure, which also implements a sorted order, both ascending and descending. It generally implements a **red-black tree** as an underlying data structure. It can be found in the C++ <set> header file.

Syntax

```
set<data_type> set;
set<data_type, greater<data_type>> set2; //this is a set in descending orde
set<data_type, comparator/lambda_function> set3; //this is a set in custom
```

There are different functions in the **set template**. These are explained briefly in the table below –

S.No.	Function	Function Explanation	TC
1.	**begin()**	Returns an iterator to the first element.	O(1)
2.	**end()**	Return an iterator to the last element.	O(1)

S.No.	Function	Function Explanation	TC
3.	size()	Returns the number of elements.	O(1)
4.	empty()	Checks if the container is empty.	O(1)
5.	insert()	Inserts a single element.	O(logn)
6.	erase()	Removes the given element.	O(logn)
7.	clear()	Removes all elements.	O(n)
8.	find()	Returns the pointer to the given element if present, otherwise, a pointer to the end.	O(logn)

Example

</> Open Compiler

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

int main(){
   set<int> set;
   set.insert(9);
   set.insert(11);
   set.insert(9);
   set.insert(11);

   bool flag=set.find(9)!=set.end();

   cout<<"Is there a 9 in the set? "<<flag<<endl;

   set.insert(21);

   for (auto i : set) {
      cout << i << " ";
   }
   cout << endl;
```

```
    return 0;
}
```

Output

```
Is there a 9 in the set? 1
9 11 21
```

Hash-Map/Map in STL

The map is a container which stores data in form of key-value pairs in a sorted order of keys, where each key is unique. It is implemented using the red-black tree data structure. It is included in the <map> header file.

Syntax

```
map<key_type,value_type> map;
```

There are different functions in the **map template**. These are explained briefly in the table below –

S.No.	Function	Function Explanation	TC
1.	**begin()**	Returns an iterator to the first element.	O(1)
2.	**end()**	Returns an iterator to the theoretical element that follows the last element	O(1)
3.	**size()**	Returns the number of elements in the map	O(1)
4.	**insert()**	Adds a new element to the map.	O(logn)
5.	**erase(*iterator*)**	Removes the element at the position pointed by the iterator.	O(logn)
6.	**erase(*key*)**	Removes the key and its value from the map.	O(logn)
7.	**clear()**	Removes all the elements from the map.	O(n)

Example

</> Open Compiler

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

int main(){
   map<char,int> map;
   string s="abbacdbbac";
   for(char c: s){
      map[c]++;
   }

   for(auto it:map){
      cout<<it.first<<" : "<<it.second<<endl;
   }
   cout<<"after erasing element 'c' : "<<endl;
   map.erase('c');
   for(auto it:map){
      cout<<it.first<<" : "<<it.second<<endl;
   }
   return 0;
}
```

Output

```
a : 3
b : 4
c : 2
d : 1
after erasing element 'c' :
a : 3
b : 4
d : 1
```

Unordered Set in STL

The unordered_set container stores unique data values in it, but the only difference from set is that the data is not stored in any order. It is implemented using the <unordered_set> header file.

Syntax

```
unordered_set<data_type> set;
```

There are different functions in the **unordered_set template**. These are explained briefly in the table below –

S. No.	Functions	Description	Time Complexity
1.	**begin()**	Returns an iterator to the first element.	O(1)
2.	**end()**	Returns an iterator to the theoretical element that follows the last element	O(1)
3.	**size()**	Returns the number of elements.	O(1)
4.	**empty()**	Returns true if the unordered_set is empty, otherwise false.	O(1)
5.	**insert()**	Insert an item in the container.	O(1)
6.	**erase()**	Removes an element from the container.	O(1)
7.	**find()**	Returns the pointer to the given element if present, otherwise, a pointer to the end.	O(1)

Example

</> Open Compiler

```
#include <iostream>
#include <unordered_set>
#include <vector>
using namespace std;

int main(){
   unordered_set<int> set;
   set.insert(19);
   set.insert(10);
   set.insert(13);
   set.insert(21);

   bool flag=set.find(9)!=set.end();

   cout<<"Is there a 9 in the set? "<<flag<<endl;
```

```
    set.insert(21);

    for (auto i : set) {
       cout << i << " ";
    }
    cout << endl;
    return 0;
}
```

Output

```
Is there a 9 in the set? 0
21 13 10 19
```

Unordered Map in STL

In the unordered_map container, the data is stored similar to the map container, but the order of the data stored is random, hence, ascending or descending order is not followed. It is included in the <unordered_map> header file.

Syntax

```
unordered_map<key_type, value_type> map;
```

There are different functions in the **unordered_map template**. These are explained briefly in the table below –

S. No.	Function	Description	Time Complexity
1.	**begin()**	Returns an iterator to the first element.	O(1)
2.	**end()**	Returns an iterator to the theoretical element that follows the last element	O(1)
3.	**size()**	Returns the number of elements.	O(1)
4.	**empty()**	Returns true if the unordered_map is empty, otherwise false.	O(1)
5.	**find()**	Returns the pointer to the given element if present, otherwise, a pointer to the end.	O(1)

S. No.	Function	Description	Time Complexity
6.	bucket()	Returns the bucket number where the data is stored.	O(1)
7.	insert()	Insert an item in the container.	O(1)
8.	erase()	Removes an element from the container.	O(1)

Example

</> Open Compiler

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

int main(){
   unordered_map<char,int> map;
   string s="abbacdbbac";
   for(char c: s){
      map[c]++;
   }

   for(auto it:map){
      cout<<it.first<<" : "<<it.second<<endl;
   }
   cout<<"after erasing element 'c' : "<<endl;
   map.erase('c');
   for(auto it:map){
      cout<<it.first<<" : "<<it.second<<endl;
   }
   return 0;
}
```

Output

```
d : 1
c : 2
b : 4
```

```
a : 3
after erasing element 'c' :
d : 1
b : 4
a : 3
```

Functors in C++ STL

Function objects or Functors, are objects in C++ STL that behave like methods/functions. These are included in the <functional> header file. These require the overloading of the **() parenthesis operator**.

The main functors in C++ STL are as follows –

- **equal_to**
- **not_equal_to**
- **greater**
- **less**
- **plus**
- **minus**

Member functions

Sr.No.	Member functions	Definition
1	**(constructor)**	It is used to construct a new std::function instance
2	**(destructor)**	It is used to destroy a std::function instance
3	**operator=**	It is used to assign a new target
4	**swap**	It is used to swap the contents
5	**assign**	It is used to assign a new target
6	**operator bool**	It is used to check if a valid target is contained
7	**operator()**	It is used to invoke the target

Non-member functions

Sr.No.	Non-member functions	Definition

1	std::swap	It specializes the std::swap algorithm
2	operator== operator!=	It compares an std::function with nullptr

Operator classes

Sr.No.	Operator classes	Definition
1	bit_and	It is a bitwise AND function object class
2	bit_or	It is a bitwise OR function object class
3	bit_xor	It is a bitwise XOR function object class
3	divides	It is a division function object class
4	equal_to	It is a function object class for equality comparison
5	greater	It is a function object class for greater-than inequality comparison
6	greater_equal	It is a function object class for greater-than-or-equal-to comparison
7	less	It is a function object class for less-than inequality comparison
8	less_equal	It is a function object class for less-than-or-equal-to comparison
9	logical_and	It is a logical AND function object class
10	logical_not	It is a logical NOT function object class
11	logical_or	It is a logical OR function object class
12	minus	It is a subtraction function object class
13	modulus	It is a modulus function object class
14	multiplies	It is a multiplication function object class
15	negate	It is a negative function object class
16	not_equal_to	It is a function object class for non-equality comparison
17	plus	It is an addition function object class

Example

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

int main(){
   equal_to<int> obj1;
   not_equal_to<int> obj2;
   greater<int> obj3;
   less<int> obj4;
   plus<int> obj5;
   minus<int> obj6;

   cout << "Functors and their usage: \n";
   cout << "Are these equal? " << obj1(11, 22) << endl;
   cout << "Are these different? " << obj2(11, 22) << endl;
   cout << "Is first index greater than second? " << obj3(10, 20) << endl;
   cout << "Is first index smaller than second? " << obj4(10, 2) << endl;
   cout << "After adding: " << obj5(10, 20) << endl;
   cout << "After subtracting: " << obj6(10, 8) << endl;

   return 0;
}
```

Output

```
Functors and their usage:
Are these equal? 0
Are these different? 1
Is first index greater than second? 0
Is first index smaller than second? 0
After adding: 30
After subtracting: 2
```

Algorithms in C++ STL

In C++ STL, algorithms template provides a variety of operations for the user, which are crucial for implementation of key algorithms like **searching**, **sorting**, returning **maximum/minimum value** from a container, and so on. These algorithms can be accessed using the <algorithm> header.

The <algorithm> library in C++ offers many useful functions. Some of the most commonly used functions are given below −

- Sort
- Copy
- Find
- Max Element and Min Element
- For Each
- Swap

sort() in C++

sort() algorithm is used to sort the given data in ascending, descending or custom order (using **comparator** or **lambda function**).

Syntax

```
sort(start_iterator,end_iterator)
sort(start,end, comparator_function) //for custom sorting
```

Example

</> Open Compiler

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

int main(){
   vector<int> v = {999, 1252, 3117, 122222 , 10, 88, 2, 9, 45, 82, 546, 42
   cout<<"Ascending order: ";
   sort(v.begin(),v.end());
   for(int i:v) cout<<i<<" ";
      cout<<endl;

   cout<<"Descending order: ";
   sort(v.begin(),v.end(),greater<int>());
   for(int i:v) cout<<i<<" ";
      cout<<endl;
```

```
   return 0;
}
```

Output

```
Ascending order: -1 2 9 10 42 45 82 88 221 546 999 1252 3117 122222
Descending order: 122222 3117 1252 999 546 221 88 82 45 42 10 9 2 -1
```

copy() in C++

copy() algorithm is used to copy the elements from one container to another.

Syntax

```
copy(start_iterator,end_iterator, destination_operator)
```

Example

</> Open Compiler

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

int main(){
   vector<int> v = { 1, 2, 3, 4, 5 , 4 ,3 , 2, 1};
   vector<int> newvec(9);
   copy(v.begin(), v.end(), newvec.begin());

   for(int &i:newvec) cout<<i<<" ";
   cout<<endl;
   return 0;
}
```

Output

```
1 2 3 4 5 4 3 2 1
```

find() in C++

find() algorithm is used to find a key element in a given range of elements.

Syntax

```
find (firstIterator, lastIterator, value);
```

Example

</> Open Compiler

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

int main(){
   vector<int> v= { 1,3,5,2,3,1,55,41};

   // finding 5
   auto itr = find(v.begin(), v.end(), 55);
   if (itr != v.end()) {
      cout << *itr << " Element found !!!" << endl;
   }else {
      cout << "Element not present !!!" << endl;
   }

   return 0;
}
```

Output

```
55 Element found !!!
```

max_element() and min_element() in C++

max_element() and min_element() are used to find the maximum and minimum value in a given range of elements.

Syntax

```
max_element (firstIterator, lastIterator);
min_element (firstIterator, lastIterator);
```

Example

</> Open Compiler

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

int main(){
   vector<int> v = {999, 1252, 3117, 122222 , 10, 88, 2, 9, 45, 82, 546, 42

   cout << "Maximum Element: " << *max_element(v.begin(),v.end())<<endl;
   cout<<"Minimum Element: "<<*min_element(v.begin(),v.end()) <<"\n";
   return 0;
}
```

Output

```
Maximum Element: 122222
Minimum Element: -1
```

for_each in C++

for_each() algorithm is used to apply a given operation or instruction to a range of elements in a container.

Syntax

```
for_each (firstIterator, lastIterator, unaryFunction);
```

Example

</> Open Compiler

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

int main(){
   vector<int> vec1 = { 1, 2, 3, 4, 5 };

   for_each(vec1.begin(), vec1.end(), [](int& i){
      i++;
   });

   for(int i:vec1) cout<<i<<" ";
      cout<<endl;

   return 0;
}
```

Output

```
2 3 4 5 6
```

swap() in C++

swap() algorithm is used to replace one element with another in place, hence the elements swap places.

Syntax

```
swap(container1,container2)
```

Example

</> Open Compiler

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

int main() {

   vector<int> vec1 = {1, 2, 3};
   vector<int> vec2 = {4, 5, 6};

   swap(vec1,vec2);
   cout<<"vec 1: "<<endl;
   for(int i:vec1) cout<<i<<" ";
      cout<<endl;

   cout<<"vec 2: "<<endl;
   for(int i:vec2) cout<<i<<" ";
      cout<<endl;
   return 0;
}
```

Output

```
vec 1:
4 5 6
vec 2: 1 2 3
```

Iterators in C++ STL

Iterators can be considered as pointers that are used to iterate over containers sequentially. These iterators are included using the <iterator> header file in C++.

Each container has its own iterator. For avoiding confusion, we can use **'auto'** keyword to define an iterator while traversing a container.

Iterators are of five types –

- Input Iterator
- Output Iterator
- Forward Iterator
- Bi-Directional Iterator
- Random Iterator

Example

</> Open Compiler

```
#include <bits/stdc++.h>
using namespace std;

int main(){
   vector<int> myvec={1,5,2,4,3,6,6,9,8};
   set<int>set(myvec.begin(),myvec.end());

   for(auto itr:myvec) cout<<itr<<" ";
      cout<<endl;
   for(auto itr:set) cout<<itr<<" ";
      cout<<endl;

   return 0;
}
```

Output

```
1 5 2 4 3 6 6 9 8
1 2 3 4 5 6 8 9
```