



Associative Containers in STL

Presented by -

- 1) Shruti Lapalkar
- 2) Aryan Kadu



What is STL?

STL - Standard Template Library is a library in C++ which contains various class and function templates which allows us to implement numerous data structures and algorithms like lists, stacks, arrays, searching and sorting.

Features provided by STL can be classified into 4 types:

- 1) Containers
- 2) Algorithms
- 3) Iterators
- 4) Functors



Containers

Containers are the data structures used to store data according to the requirement. A container is implemented as a template class that also contains methods to perform operations on it.

Every STL container is defined inside its own header file.

For example, `#include<vector>`, `#include<set>`, etc.



Types of Containers

- 1) Sequence containers - Containers used for storing the data in linear manner.

Types of sequence containers - arrays, vectors, Deque, Forward Lists, Lists.

- 2) Container adaptors - They are STL containers which are adapted from pre-existing containers and modified according to specific needs.

Types of container adaptors - Stack, Queue, Priority Queue

- 3) Associative Containers - Associative containers are the type of containers that store the elements in a sorted order based on keys rather than their insertion order.

Types of associative containers - Sets, maps, multisets, multimaps

- 4) Unordered associative containers - They store the data in no fixed manner.

Types of unordered associative containers - unordered map, unordered set, unordered multiset, unordered multimap

Sets

Sets are a type of associative container in which each element has to be unique and thus no repetition of elements is allowed. If the set is not unordered the elements of a set are always in a sorted manner. Either ascending or descending. If given multiple same values only one of them is stored in the set.

Example

```
cpp > g++ setdemo.cpp > main()
1  #include <iostream>
2  #include<set>
3  using namespace std;
4  int main()
5  {
6      set<int> set_ = {3, 1, 5, 90, 11};
7      for(auto it = set_.begin(); it!=set_.end(); it++){
8          cout<<*it<<" ";
9      }
10     return 0;
11 }
```

Output

```
C:\Users\Shruti Lapalkar\cpp>cd "c:\Users\Shruti Lapalkar\cpp\cpp\" && g++ setdemo.cpp -o
setdemo && "c:\Users\Shruti Lapalkar\cpp\cpp\"setdemo
1 3 5 11 90
c:\Users\Shruti Lapalkar\cpp\cpp>
```



Sets

auto keyword - it is used to deduce the type of the variable at compile time based on the initialization of the variable.

begin() - it is a function which returns a pointer that points to the first element of the set.

end() - it is a function that returns a pointer that points to the last + 1 element.

By default the set is sorted in ascending order, if we want it to be sorted in descending order then declare the set as `set<int, greater<int>> set_;`



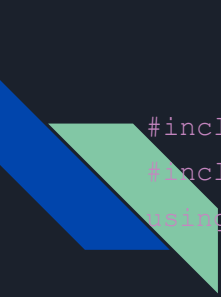
Properties of a set

- 1) Storing order – The set stores the elements in sorted order.
- 2) Values Characteristics – All the elements in a set have unique values.
- 3) Values Nature – The value of the element cannot be modified once it is added to the set, though it is possible to remove and then add the modified value of that element. Thus, the values are immutable.
- 4) Arranging order – The values in a set are unindexed.



Some set methods

<code>insert(val)</code>	Adds an element <code>val</code> to the set. If it already exists, the insertion is ignored.
<code>erase(val)</code>	Removes the element <code>val</code> from the set.
<code>find(val)</code>	Returns an iterator to the element <code>val</code> if found; otherwise, returns <code>end()</code> .
<code>size()</code>	Returns the number of elements in the set.
<code>empty()</code>	Checks if the set is empty and returns <code>true</code> if it is.
<code>clear()</code>	Removes all elements from the set
<code>swap(set& other)</code>	Exchanges the contents of the set with another set <code>other</code> .



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

```
int main() {
    set<int> mySet;
```


```
    mySet.insert(10);
    mySet.insert(20);
    mySet.insert(30);
    mySet.insert(20);
```

```
    cout << "Set after insertions: ";
    for (auto elem : mySet) {
        cout << elem << " ";
    }
    cout << endl;
```

```
    cout << "Size of set: " << mySet.size() << endl;
    cout << "Is set empty? " << mySet.empty() << endl;
```

```
    int searchVal = 20;
    auto it = mySet.find(searchVal);
    if (it != mySet.end()) {
        cout << searchVal << " is found in the set." << endl;
    } else {
        cout << searchVal << " is not found in the set."
        << endl;
    }
```

```
    cout << "Count of 20 in set: " << mySet.count(20) << endl;
```



```
mySet.erase(10);
    cout << "Set after erasing 10:
";
    for (auto elem : mySet) {
        cout << elem << " ";
    }
    cout << endl;

    mySet.clear();
    cout << "Set after clearing:
Size = " << mySet.size() << endl;

    mySet.insert(40);
    mySet.insert(50);
    cout << "Set after reinsertion:
";
    for (auto elem : mySet) {
        cout << elem << " ";
    }
    cout << endl;
```

```
mySet.insert(45);
    mySet.insert(35);

    cout << "Set elements after additional insertions: ";
    for (auto elem : mySet) {
        cout << elem << " ";
    }
    cout << endl;

    return 0;
}
```



Maps

Maps are the associative containers that store elements in a mapped fashion. Each element has a key value and a mapped value. No two mapped values can have the same key values.

`std::map` is the class template for map containers and it is defined inside the `<map>` header file.



Some Basic Functions Associated with map are :


<code>begin()</code>	Return an iterator to the first element in the map.
<code>end()</code>	Return an iterator to the last element in the map.
<code>size()</code>	Return the number of elements in the map.
<code>max_size()</code>	Return the max number of elements map can hold.
<code>empty()</code>	Return whether the map is empty.
<code>insert(key , value)</code>	Add new element to the map.



For example :

```
C++ map.cpp > main()
1  #include<iostream>
2  #include<map>
3  using namespace std;
4  int main(){
5      map<int,int>mpp;
6      mpp[1] = 2;
7      mpp.insert({3,4});
8      for(auto it:mpp){
9          cout<<"key: "<<it.first<<" value: "<<it.second<<" ";
10     }
11     return 0;
12 }
```

key: 1 value: 2,key: 3 value: 4,
PS D:\Aryan.py\CPP Binary Tree>



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

int main(){
    map<int,char>mpp;
    mpp.insert({1,'a'});
    mpp.insert({2,'b'});
    mpp.insert({3,'c'});
    mpp.insert({4,'d'});

    cout<<"Displaying elements : ";
    for(auto it : mpp){
        cout<<it.first<<" "<<it.second<<"
";
    }cout<<endl;

    cout<<"Elements : "<<mpp[1]<<"
"<<mpp[3]<<endl; // Can access mapped
values by using keys;
```

```
auto it = mpp.find(5); // If element is
not present it points to end;

    mpp.erase(3); // Erases Element from
map;

    cout<<"After erasing : ";
    for(auto it : mpp){
        cout<<it.first<<" "<<it.second<<" ";
    }cout<<endl;

    cout<<"Size of map is
:"<<mpp.size()<<endl; // Displays size;

    mpp.erase(1);
    mpp.erase(2);
    mpp.erase(4);

    cout<<mpp.empty();

    return 0;
}
```



THANK YOU !