C++ Standard Template Library (STL) Datasheet

<bits/stdc++.h>

INDEX

https://www.linkedin.com/in/durgesh-mahajan-99bab0212/ durgeshmahajan1722@gmail.com @durgeshm01722

https://github.com/durgeshm01722

1. Introduction

What is Standard Template Library in C++ :-

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 mainly has 3 components -

1. Containers
2. Algorithms
3. Iterators

So, let's see them one by one…

Note – The 'bits/stdc++.h>' is the universal header file for every C++ STL element. Include it in your every C++ program to avoid writing the header files for specific or separate STL elements.

2. Containers

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

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

In C++ STL, we have 4 types of Containers –

1. Sequence Containers
   - 1.1 Vector
   - 1.2 Deque
   - 1.3 List
   - 1.4 Arrays
   - 1.5 Forward Lists
2. Container Adaptors
   - 2.1 Stack
   - 2.2 Queue
   - 2.3 Priority Queue
3. Associative Containers
   - 3.1 Set
   - 3.2 Map
   - 3.3 Multiset
   - 3.4 Multimap
4. Unordered Associative Containers
   - 4.1 Unordered Set
   - 4.2 Unordered Map
   - 4.3 Unordered Multiset
   - 4.4 Unordered Multimap

Let’s see the most widely and most commonly used STL containers

2.1 Sequence Containers

2.1.1 Vectors

Vectors are dynamic arrays with the ability to resize themselves automatically when an element is inserted or deleted, with their storage being handled automatically by the container.

Operations on vectors -

1. push_back() - It push the elements into a vector from the back.
2. pop_back() - It is used to pop or remove elements from a vector from the back.
3. size() - Returns the number of elements in the vector.
4. empty() - Returns whether the vector is empty or not.
5. erase() - It is used to remove elements from a container from the specified position or range.
6. clear() - It is used to remove all the elements of the vector container.
7. front() - Returns a reference to the first element in the vector.
8. back() - Returns a reference to the last element in the vector.
9. sort(v.begin(), v.end()) - Sorts the elements in the vector.
10. reverse(v.begin(), v.end()) - Reverses the elements in the vector.

Syntax -

```
vector<data_type> vector_name;
```

Operation	Time Complexity	Space Complexity
push_back()	O(1)	O(1)
pop_back()	O(1)	O(1)
size()	O(1)	O(1)
empty()	O(1)	O(1)
erase()	O(n)	O(1)
clear()	O(n)	O(1)
front()	O(1)	O(1)
back()	O(1)	O(1)
sort()	O(nlog(n))	O(1)
reverse()	O(n)	O(1)

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

int main(){
    vector<int> v;              // Creating a vector
    vector<int> v2(5);          // Creating a vector with 5 elements
    vector<int> v3(5, 0);       // Creating a vector with 5 elements having value 0

    v.push_back(5);             // Adding 5
    v.push_back(11);            // Adding 11
    v.push_back(2);             // Adding 2
    v.push_back(7);             // Adding 7

    cout<<v.size()<<endl;       // Outputs 4

    v.pop_back();               // Remove last element i.e. 7

    cout<<v.empty()<<endl;      // Outputs 0 (v is not empty)

    sort(v.begin(), v.end());   // Sorting the vector

    for(int i=0;i<v.size();i++){
        cout<<v[i]<<" ";            // Outputs 2 5 11
    }
    return 0;
}

```

2.1.2 Deque

Double-ended queues are sequence containers with the feature of expansion and contraction on both ends. Double-ended queues are a special case of queues where insertion and deletion operations are possible at both the ends.

Operations on deque -

1. push_back() - It pushes the elements into deque from the back.
2. push_front() - It pushes the elements into deque from the front.
3. pop_back() - It is used to pop or remove elements from deque from the back.
4. pop_front() - It is used to pop or remove elements from deque from the front.
5. size() - Returns the number of elements in the deque.
6. empty() - Returns whether the deque is empty or not.
7. erase() - It is used to remove elements from a container from the specified position or range.
8. clear() - It is used to remove all the elements of the deque container.
9. front() - Returns a reference to the first element in the deque.
10. back() - Returns a reference to the last element in the deque.

Syntax -

```
deque<data_type> deque_name;
```

Operation	Time Complexity	Space Complexity
push_back()	O(1)	O(1)
pop_back()	O(1)	O(1)
push_front()	O(1)	O(1)
pop_front()	O(1)	O(1)
size()	O(1)	O(1)
empty()	O(1)	O(1)
erase()	O(n)	O(1)
clear()	O(n)	O(1)
front()	O(1)	O(1)
back()	O(1)	O(1)

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

int main(){
    deque<int> d;              // Creating a deque
    deque<int> d2(5);          // Creating a deque with 5 elements
    deque<int> d3(5, 0);       // Creating a deque with 5 elements having value 0

    d.push_back(5);             // Adding 5 from back
    d.push_back(11);            // Adding 11 from back
    d.push_back(2);             // Adding 2 from back
    d.push_front(7);             // Adding 7 from front
    d.push_front(14);              // Adding 14 from front

    cout<<d.size()<<endl;       // Outputs 5

    d.pop_back();               // Remove element from back i.e. 2
    d.pop_front();               // Remove element from front i.e. 14

    cout<<d.empty()<<endl;      // Outputs 0 (d is not empty)

    reverse(d.begin(), d.end());   // Reversing the deque

    for(int i=0;i<d.size();i++){
        cout<<d[i]<<" ";            // Outputs 11 5 7
    }
    return 0;
}

```

2.1.3 Lists

Lists are sequence containers that allow non-contiguous memory allocation. As compared to vector, the list has slow traversal, but once a position has been found, insertion and deletion are quick.

This list is implemented using a doubly-linked list. For implementing a singly linked list, we use a forward list.

Operations on list -

1. push_back() - Adds a new element at the end of the list.
2. push_front() – Adds a new element at the beginning of the list.
3. pop_back() – Removes the last element of the list.
4. pop_front() - Removes the first element of the list.
5. size() - Returns the number of elements in the list.
6. empty() - Returns whether the list is empty or not.
7. erase() - It is used to remove elements from a container from the specified position or range.
8. clear() - It is used to remove all the elements of the list container.
9. front() - Returns the value of the first element in the list.
10. back() - Returns the value of the last element in the list.

Syntax -

```
list<data_type> list_name;
```

Operation	Time Complexity	Space Complexity
push_back()	O(1)	O(1)
pop_back()	O(1)	O(1)
push_front()	O(1)	O(1)
pop_front()	O(1)	O(1)
size()	O(1)	O(1)
empty()	O(1)	O(1)
erase()	O(n)	O(1)
clear()	O(n)	O(1)
front()	O(1)	O(1)
back()	O(1)	O(1)

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

int main(){
    list<int> l;              // Creating a list
    list<int> l2(5);          // Creating a list with 5 elements
    list<int> l3(5, 0);       // Creating a list with 5 elements having value 0

    l.push_back(5);             // Adding 5 from back
    l.push_back(11);            // Adding 11 from back
    l.push_back(2);             // Adding 2 from back
    l.push_front(7);             // Adding 7 from front
    l.push_front(14);             // Adding 14 from front

    cout<<l.size()<<endl;       // Outputs 5

    l.pop_back();               // Remove element from back i.e. 2
    l.pop_front();               // Remove element from front i.e. 14

    cout<<l.empty()<<endl;      // Outputs 0 (l is not empty)

    reverse(l.begin(), l.end());   // Reversing the list

    for(auto i: l){
        cout<<i<<" ";            // Outputs 11 5 7
    }
    return 0;
}
```

2.2 Container Adaptors

2.2.1 Stack

Stacks are a type of container adaptors with LIFO(Last In First Out) type of working, where a new element is added at one end (top) and an element is removed from that end only.

Operations on stack -

1. push() - It adds an element to the top of the stack.
2. pop() - It deletes the top most element of the stack.
3. size() - Returns the number of elements in the stack.
4. empty() - Returns whether the stack is empty or not.
5. top() - Returns a reference to the top most element of the stack.

Syntax -

```
stack<data_type> stack_name;
```

Operation	Time Complexity	Space Complexity
push()	O(1)	O(1)
pop()	O(1)	O(1)
size()	O(1)	O(1)
empty()	O(1)	O(1)
top()	O(1)	O(1)

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

int main(){
    stack<int> s;              // Creating a stack

    s.push(5);             // Adding 5
    s.push(11);            // Adding 11
    s.push(2);             // Adding 2
    s.push(7);             // Adding 7

    cout<<s.size()<<endl;       // Outputs 4
    cout<<s.top()<<endl;       // Outputs 7

    s.pop();               // Remove topmost element i.e. 7

    cout<<s.empty()<<endl;      // Outputs 0 (s is not empty)

    while(!s.empty()){
        cout<<s.top()<<" ";            // Outputs 2 11 5
        s.pop();               // Pops/Removes every top most element
    }
    return 0;
}

```

2.2.2 Queue

Queues are a type of container adaptors that operate in a first in first out (FIFO) type of arrangement. Elements are inserted at the back (end) and are deleted from the front.

Operations on stack -

1. push() - It adds an element to the end of the queue.
2. pop() - It deletes the first element of the queue.
3. size() - Returns the number of elements in the queue.
4. empty() - Returns whether the queue is empty or not.
5. top() - Returns a reference to the top most element of the stack.
6. front() - Returns a reference to the first element in the queue.
7. back() - Returns a reference to the last element in the queue.

Syntax -

```
queue<data_type> queue_name;
```

Operation	Time Complexity	Space Complexity
push()	O(1)	O(1)
pop()	O(1)	O(1)
size()	O(1)	O(1)
empty()	O(1)	O(1)
front()	O(1)	O(1)
back()	O(1)	O(1)

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

int main(){
    queue<int> q;              // Creating a queue

    q.push(5);             // Adding 5
    q.push(11);            // Adding 11
    q.push(2);             // Adding 2
    q.push(7);             // Adding 7

    cout<<q.size()<<endl;       // Outputs 4
    cout<<q.front()<<endl;       // Outputs 5
    cout<<q.back()<<endl;       // Outputs 7

    q.pop();               // Remove the front element i.e. 5

    cout<<q.empty()<<endl;      // Outputs 0 (q is not empty)

    while(!q.empty()){
        cout<<q.front()<<" ";            // Outputs 11 2 7
        q.pop();               // Pops/Removes every front element
    }
    return 0;
}

```

2.2.3 Priority Queue

Priority queues are a type of container adapters, specifically designed such that the first element of the queue is either the greatest or the smallest of all elements in the queue. Priority queues are built on the top to the max heap by default. Hence, the elements are stored in decreasing order by default.

Operations on priority queue -

1. push() - It adds an element to the end of the queue.
2. pop() - It deletes the first element of the queue.
3. size() - Returns the number of elements in the queue.
4. empty() - Returns whether the queue is empty or not.
5. top() - Returns a reference to the top most element of the queue.

Syntax -

For max-heap implementation(default) i.e. elements are stored in decreasing order -

```
priority_queue<data_type> queue_name;
```

For min-heap implementation i.e. elements are stored in increasing order -

```
priority_queue<data_type, vector<data_type>, greater<data_type>> queue_name;
```

Operation	Time Complexity	Space Complexity
push()	O(log(n))	O(1)
pop()	O(log(n))	O(1)
size()	O(1)	O(1)
empty()	O(1)	O(1)
top()	O(1)	O(1)

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

int main(){
    priority_queue<int> pq;             // Creating a priority_queue for max-heap (default)
    priority_queue<int, vector<int>, greater<int>> pq2;    // Creating a priority_queue for min-heap

    pq.push(5);             // Adding 5
    pq2.push(5);
    pq.push(11);            // Adding 11
    pq2.push(11);
    pq.push(2);             // Adding 2
    pq2.push(2);
    pq.push(7);             // Adding 7
    pq2.push(7);

    cout<<pq.size()<<endl;       // Outputs 4
    cout<<pq.top()<<endl;       // Outputs 11

    pq.pop();               // Remove topmost element i.e. 11

    cout<<pq.empty()<<endl;      // Outputs 0 (pq is not empty)

    while(!pq.empty()){
        cout<<pq.top()<<" ";            // Outputs 7 5 2
        pq.pop();                // Pops/Removes every top/first element of pq
    }
    cout<<endl;
    while(!pq2.empty()){
        cout<<pq2.top()<<" ";            // Outputs 2 5 7 11
        pq2.pop();               // Pops/Removes every top/first element of pq2
    }
    return 0;
}
```

2.3 Associative Containers

2.3.1 Set

Sets are a type of associative containers in which each element has to be unique because the value of the element identifies it. Here, the values are stored in a specific or sorted order. It doesn't allow duplicate values i.e. values are unique. The values are immutable i.e. 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.

It follows binary search tree implementation. Also, the values stored in it are unindexed.

Operations on set -

1. insert() - It adds a new element to the set.
2. erase() - It removes the specified value from the set.
3. clear() - It removes all the elements from the set.
4. size() - Returns the number of elements in the set.
5. empty() - Returns whether the set is empty or not.
6. find() - Returns an iterator to the specified element in the set if found, else returns the iterator to end.

Syntax -

For storing values in ascending order -

```
set<data_type> set_name;
```

For storing values in descending order -

```
set<data_type, greater<data_type>> set_name;
```

Operation	Time Complexity	Space Complexity
insert()	O(log(n))	O(1)
erase()	O(log(n))	O(1)
clear()	O(log(N))	O(1)
size()	O(1)	O(1)
empty()	O(1)	O(1)
find()	O(log(n))	O(1)

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

int main(){
    set<int> s;                     // Creating a set in which values are stored in ascending order
    set<int, greater<int>> s2;      // Creating a set in which values are stored in descending order

    s.insert(5);           // Adding 5
    s.insert(11);          // Adding 11
    s.insert(2);           // Adding 2
    s.insert(7);           // Adding 7
    s2.insert(5);
    s2.insert(11);
    s2.insert(2);
    s2.insert(7);

    s.erase(5);             // Removes 5 from s
    cout<<s.size()<<endl;       // Outputs 3

    cout<<s.empty()<<endl;      // Outputs 0

    for(auto i: s){
        cout<<i<<" ";            // Outputs 2 7 11
    }
    cout<<endl;
    for(auto i: s2){
        cout<<i<<" ";            // Outputs 11 7 5 2
    }
    return 0;
}
```

2.3.2 Map

Maps are 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.

Operations on map -

6. insert(pair<dt, dt> (key, value)) – It inserts elements with a particular key in the map container.
7. erase() - It removes the specified key-value from the map.
8. clear() - It removes all the elements from the map.
9. size() - Returns the number of elements in the map.
10. empty() - Returns whether the map is empty or not.
11. find() - Returns an iterator to the specified element in the map if found, else returns the iterator to end.
12. count() - Returns the number of matches to element with the specified key-value in the map.

Syntax -

```
map<data_type, data_type> map_name;
```

Operation	Time Complexity	Space Complexity
insert()	O(log(n))	O(1)
erase()	O(log(n))	O(1)
clear()	O(log(N))	O(1)
size()	O(1)	O(1)
empty()	O(1)	O(1)
find()	O(log(n))	O(1)
count()	O(log(n))	O(1)

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

int main(){
    map<string, int> m;                     // Creating a map

    m.insert(pair<string, int> ("English", 45));     // Adding "English" as a key and "45" as a value in map
    m.insert(pair<string, int> ("Maths", 48));       // Adding "Maths" as a key and "48" as a value in map
    m["Science"] = 50;                              // Another way of adding elements into the map
    m["History"] = 41;

    m.erase("Science");         // Removes "Science" key and it's value from map
    cout<<m.size()<<endl;       // Outputs 3

    cout<<m.empty()<<endl;      // Outputs 0

    for(auto i: m){
        cout<<i.first<<" = "<<i.second<<endl;
    }

    //      Outputs
    //      English = 45
    //      History = 41
    //      Maths = 48
    return 0;
}

```

2.4 Unordered Associative Containers

2.4.1 Unordered Set

An unordered set is implemented using a hash table where keys are hashed into indices of a hash table so that the insertion is always randomized. All operations on the unordered_set takes constant time O(1).

The unordered_set allows only unique keys, for duplicate keys unordered_multiset should be used.

Operations on unordered_set -

1. insert() - It adds a new element to the unordered_set.
2. erase() - It removes the specified value from the unordered_set.
3. clear() - It removes all the elements from the unordered_set.
4. size() - Returns the number of elements in the unordered_set.
5. empty() - Returns whether the unordered_set is empty or not.
6. find() - Returns an iterator to the specified element in the unordered_set if found, else returns the iterator to end.

Syntax -

```
unordered_set<data_type> set_name;
```

Operation	Time Complexity	Space Complexity
insert()	O(1) or O(n) in worst case	O(1)
erase()	O(1)	O(1)
clear()	O(1)	O(1)
size()	O(1)	O(1)
empty()	O(1)	O(1)
find()	O(1) or O(n) in worst case	O(1)

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

int main(){
    unordered_set<int> us;           // Creating an unordered_set

    us.insert(5);           // Adding 5
    us.insert(11);          // Adding 11
    us.insert(2);           // Adding 2
    us.insert(7);           // Adding 7

    us.erase(5);             // Removes 5 from s
    cout<<us.size()<<endl;       // Outputs 3

    cout<<us.empty()<<endl;      // Outputs 0

    for(auto i: us){
        cout<<i<<" ";            // Outputs 7 2 11
    }
    return 0;
}

```

2.4.2 Unordered Map

The unordered_map is an associated container that stores elements formed by the combination of key-value and a mapped value. The key value is used to uniquely identify the element and the mapped value is the content associated with the key.

Internally unordered_map is implemented using Hash Table. All the operations on the unordered_map takes constant time O(1).

Operations on unordered_map -

1. insert(pair<dt, dt> (key, value)) – It inserts elements with a particular key in the unordered_map container.
2. erase() - It removes the specified key-value from the unordered_map.
3. clear() - It removes all the elements from the unordered_map.
4. size() - Returns the number of elements in the unordered_map.
5. empty() - Returns whether the unordered_map is empty or not.
6. find() - Returns an iterator to the specified element in the unordered_map if found, else returns the iterator to end.
7. count() - Returns the number of matches to element with the specified key-value in the unordered_map.

Syntax -

```
unordered_map<data_type, data_type> map_name;
```

Operation	Time Complexity	Space Complexity
insert()	O(1) or O(n) in worst case	O(1)
erase()	O(1)	O(1)
clear()	O(1)	O(1)
size()	O(1)	O(1)
empty()	O(1)	O(1)
find()	O(1) or O(n) in worst case	O(1)
count()	O(1)	O(1)

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

int main(){
    unordered_map<string, int> um;                    // Creating an unordered_map

    um.insert(pair<string, int> ("English", 45));     // Adding "English" as a key and "45" as a value in map
    um.insert(pair<string, int> ("Maths", 48));       // Adding "Maths" as a key and "48" as a value in map
    um["Science"] = 50;                               // Another way of adding elements into the map
    um["History"] = 41;

    um.erase("Science");         // Removes "Science" key and it's value from map
    cout<<um.size()<<endl;       // Outputs 3

    cout<<um.empty()<<endl;      // Outputs 0

    for(auto i: um){
        cout<<i.first<<" = "<<i.second<<endl;
    }

    //      Outputs
    //      History = 41
    //      Maths = 48
    //      English = 45
    return 0;
}
```

3. Algorithms

3.1 Binary Search

Syntax -

```
binary_search (start_address, end_address, value_to_find);
```

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

int main(){
    vector<int> v = {4, 13, 27, 55, 71, 98};

    cout<<binary_search(v.begin(), v.end(), 27)<<endl;    // Outputs 1 because 27 is present in v
    cout<<binary_search(v.begin(), v.end(), 44)<<endl;    // Outputs 0 because 44 is not present in v
    return 0;
}

```

It’s Binary Search, so make sure your array/vector/container has sorted elements.

Time Complexity - O(nlog(n))
Space Complexity - O(1)

3.2 Sort

Syntax -

For sorting in ascending(default) order -

```
sort (start_address, end_address);
```

For sorting in descending order -

```
sort (start_address, end_address, greater<data_type>());
```

For sorting in your own way/order -

```
sort (start_address, end_address, your_comparision_function);
```

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

bool compare(string s1, string s2){
    return s1.length() < s2.length();
}

int main(){
    vector<int> v = {4, 55, 98, 27, 13, 71};
    vector<int> v2 = {4, 55, 98, 27, 13, 71};
    vector<string> v3 = {"Apple", "Banana", "Pineapple", "Kiwi"};

    sort(v.begin(), v.end());               // Sorting in ascending order
    sort(v2.begin(), v2.end(), greater<int>());     // Sorting in descending order
    sort(v3.begin(), v3.end(), compare);     // Sorting by the lengths of the strings using our own compare function
    for(int i: v){
        cout<<i<<" ";           // Outputs 4 13 27 55 71 98
    }
    cout<<endl;
    for(int j: v2){
        cout<<j<<" ";           // Outputs 98 71 55 27 13 4
    }
    cout<<endl;
    for(string k: v3){
        cout<<k<<" ";           // Outputs Kiwi Apple Banana Pineapple
    }
    return 0;
}

```

Time Complexity - O(nlog(n))
Space Complexity - O(log(n))

3.3 Swap

Syntax -

```
swap (value1, value2);
```

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

int main(){
    int a = 15;
    int b = 27;

    cout<<a<<" "<<b<<endl;      // Outputs 15 27
    swap(a, b);
    cout<<a<<" "<<b<<endl;      // Outputs 27 15
    return 0;
}

```

Time Complexity - O(1)
Space Complexity - O(1)

3.4 Reverse

Syntax -

```
reverse (start_address, end_address);
```

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

int main(){
    vector<int> v = {4, 55, 98, 27, 13, 71};

    reverse(v.begin(), v.end());
    for(int i: v){
        cout<<i<<" ";           // Outputs 71 13 27 98 55 4
    }
    return 0;
}

```

Time Complexity - O(n)
Space Complexity - O(1)

3.5 Min and Max

Syntax -

For default comparison -

```
min (value1. value2);
max (value1. value2);
```

For comparison in your own way -

```
min (value1. value2, your_comparision_function);
max (value1. value2, your_comparision_function);
```

For comparison on a whole container or on a range of values -

```
*min_element (start_address, end_address);
*max_element (start_address, end_address);
*min_element (start_address, end_address, your_comparision_function);
*max_element (start_address, end_address, your_comparision_function);
```

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

bool compare(string s1, string s2){
    return s1.length() < s2.length();
}

int main(){
    int a = 15, b = 27;
    cout<<min(a, b)<<endl;      // Outputs 15
    cout<<max(a, b)<<endl;      // Outputs 27

    string s1 = "Orange", s2 = "Mango";
    cout<<min(s1, s2, compare)<<endl;       // Outputs Mango
    cout<<max(s1, s2, compare)<<endl;       // Outputs Orange

    vector<int> v = {4, 55, 98, 27, 13, 71};
    cout<<*min_element(v.begin(), v.end())<<endl;       // Outputs 4
    cout<<*max_element(v.begin(), v.end())<<endl;       // Outputs 98

    return 0;
}
```

Time Complexity - O(1), O(n) for range of values

Space Complexity - O(1)

4. Iterators

Iterators are used to point at the memory addresses of STL containers. They are primarily used in sequences of numbers, characters etc. They reduce the complexity and execution time of the program.

Syntax -

```
container_type<data_type>:: iterator iterator_name;
```

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

int main(){
    vector<int> v = {4, 55, 98, 27, 13, 71};

    vector<int>:: iterator ir;      // Declaring an iterator to a vector

    for(ir = v.begin();ir!=v.end();ir++){
        cout<<*ir<<" ";             // Outputs 4 55 98 27 13 71
    }
    return 0;
}

```

Thanks for Reading!!!

Do share your valuable feedback in the comments if you found this content helpful.

You can also email your feedback or suggestions on my email - durgeshmahajan1722@gmail.com