Search...

C++ STL Cheat Sheet

Last Updated : 02 Apr, 2025

The C++ STL Cheat Sheet provides short and concise notes on Standard Template Library (STL) in C++. Designed for programmers that want to quickly go through key STL concepts, the STL cheatsheet covers the concepts such as vectors and other containers, iterators, functors, etc., with their syntax and example.

What is Standard Template Library(STL)?

The **C++ Standard Template Library (STL)** is a collection of generic class and function templates to provide some commonly used data structures and algorithms. It contains optimized and error-free code for useful containers such as vector, list, stack, queue, etc. It is a part of the standard library of C++ and

Components of STL

C++ STL provides various components to make programming easier and more efficient. These components can be divided into four categories:

1. **Containers**
2. **Iterators**
3. **Algorithms**
4. **Functors**

STL Containers

The STL containers are the template classes to implement useful data structures such as dynamic arrays, hashmaps, linked lists, trees, etc. These containers allow programmers to store and manipulate data.

The STL containers can also be divided into 4 parts which are:

1. Sequential Containers
2. Container Adapters
3. Associative Containers
4. Unordered Containers

1. Sequential Containers

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

- *Vector*
- *List*
- *Deque*
- *Array*
- *Forward List*

2. 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*

3. 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*

4. Unordered Containers

The unordered containers are similar to associative containers except that they don't store sorted data but still provide quick search time using key-value pairs. They are:

- *Unordered Set*
- *Unordered Multiset*
- *Unordered Map*
- *Unordered Multimap*

Each container is defined inside its respective header file with the same name.

1.1 – STL Vector

The vector container provides a dynamic array. It is defined as **std::vector** class template inside **<vector>** header file.

Vector Declaration

```
vector <data_type> vector_name;    // 1D Vector
vector < vector<data_type> > vector_name;    // 2D Vector
```

std::vector Member Functions

S. No.	Function	Description	Time Complexity
1.	begin()	Returns an iterator to the first element.	O(1)

S. No.	Function	Description	Time Complexity
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)

Example:

```
// C++ program to illustrate the vector container
#include <iostream>
```

```cpp
#include <vector>
using namespace std;

int main()
{
    // 1d vector with initialization list
    vector<int> v1 = { 1, 2, 3, 4, 5 };
    // 2d vector with size and element value initialization
    vector<vector<int> > v2(3, vector<int>(3, 5));

    // adding values using push_back()
    v1.push_back(6);
    // printing v1 using size()
    cout << "v1: ";
    for (int i = 0; i < v1.size(); i++) {
        cout << v1[i] << " ";
    }
    cout << endl;

    v1.erase(v1.begin() + 4);
    // printing v1 using iterators
    cout << "v1: ";
    for (auto i = v1.begin(); i != v1.end(); i++) {
        cout << *i << " ";
    }

    // printing v2 using range based for loop
    cout << "v2:-" << endl;
    for (auto i : v2) {
        for (auto j : i) {
            cout << j << " ";
        }
        cout << endl;
    }
    return 0;
}
```

Output

```
v1: 1 2 3 4 5 6
v1: 1 2 3 4 6 v2:-
5 5 5
5 5 5
5 5 5
```

1.2 – STL List

The list container implements the doubly linked list data structure. It is defined as **std::list** inside **<list>** header file.

List Declaration

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

std::List Member Functions

S. No.	Function	Description	Time Complexity
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

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

// driver code
int main()
{
    // creating std::list object using initializer list
    list<int> l1 = { 1, 5, 9, 1, 4, 6 };

    // vector for initialization
    vector<char> v = { 'G', 'e', 'e', 'k', 's' };
    list<int> l2(v.begin(), v.end());

    // printing first element
    cout << "First element if l1: " << l1.front() << endl;

    // adding element
    l1.insert(l1.begin(), 5);

    // deleting element
    l1.erase(l1.begin());

    // traversing and printing l1
    cout << "l1: ";
    for (auto i = l1.begin(); i != l1.end(); i++) {
        cout << *i << " ";
    }
    cout << endl;

    // traversing and printing l2
    cout << "l2: ";
    for (auto i : l2) {
        cout << char(i);
    }
    cout << endl;

    return 0;
}
```

Output

```
First element if l1: 1
l1: 1 5 9 1 4 6
l2: Geeks
```

1.3 – STL Deque

The deque implements the double-ended queue which follows the FIFO mode of operation but unlike the queue, the deque can grow and shrink from both ends. It is defined as **std::deque** inside the **<deque>** header file.

Deque Declaration

```
deque <data_type> dequeu_name;
```

std::deque Member Functions

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

Example

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

int main()
{
    // creating a deque
    deque<int> d = { 1, 2, 3, 4, 5 };

    // removing two elements from the back and pushing them
    // at the front
    d.push_front(d.back());
    d.pop_back();
    d.push_front(d.back());
    d.pop_back();

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

    return 0;
}
```

Output

```
4 5 1 2 3
```

1.4 – STL Stack

The stack is a container adaptor that operates one LIFO principle. It is defined as **std::stack** in **<stack>** header file.

Stack Declaration

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

std::stack Member Functions

S. No.	Function	Description	Time Complexity
1.	empty()	Returns true if the stack is empty, false otherwise.	O(1)

S. No.	Function	Description	Time Complexity
2.	size()	Returns the number of elements in the stack.	O(1)
3.	top()	Returns the top element.	O(1)
4.	push(g)	Push one element in the stack.	O(1)
5.	pop()	Removes one element from the stack.	O(1)

Example

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

int main()
{
    stack<int> s;

    for (int i = 1; i <= 5; i++) {
        s.push(i);
    }

    s.push(6);
    // checking top element
    cout << "s.top() = " << s.top() << endl;
    cout << "s: ";
    while (!s.empty()) {
        cout << s.top() << " ";
        s.pop();
    }

    // size after popping all elements
    cout << "Final Size: " << s.size();

    return 0;
}
```


Output

```
s.top() = 6
s: 6 5 4 3 2 1 Final Size: 0
```

1.5 – STL Queue

The queue is a container adapter that uses the FIFO mode of operation where the most recently inserted element can be accessed at last. It is defined as the **std::queue** class template in the **<queue>** header file.

Queue Declaration

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

std::queue Member Functions

S. No.	Function	Description	Time Complexity
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

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

int main()
{

    // creating queue
    queue<int> q;
```

```
    // pushing elements
    for (int i = 1; i <= 5; i++) {
        q.push(i);
    }
    q.push(6);

    cout << "q.front() = " << q.front() << endl;
    cout << "q.back() = " << q.back() << endl;

    // printing queue by popping all elements
    cout << "q: ";
    int size = q.size();
    for (int i = 0; i < size; i++) {
        cout << q.front() << " ";
        q.pop();
    }

    return 0;
}
```

Output

```
q.front() = 1
q.back() = 6
q: 1 2 3 4 5 6
```

1.6 – STL Set

The set is an associative container that stores unique values in sorted order, either ascending or descending. It generally implements a red-black tree as an underlying data structure. It is defined as **std::set** class template inside **<set>** header file.

> ***Note:*** *To store the multiple keys, we can use the* ***multiset*** *container.*

Set Declaration

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

std::set Member Functions

S. No.	Function	Description	Time Complexity
1.	begin()	Returns an iterator to the first element.	O(1)
2.	end()	Return an iterator to the last element.	O(1)
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

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

int main()
{
    // creating vector
    vector<int> v = { 1, 5, 3, 4, 2 };
    // creating set using vector v
    set<int> s(v.begin(), v.end());

    // finding 4
    if (s.find(4) == s.end()) {
        cout << "4 not found" << endl;
    }
    else {
        cout << "4 found" << endl;
    }

    // adding 9
```

```
    s.insert(9);

    // printing set
    cout << "s: ";
    for (set<int>::iterator i = s.begin(); i != s.end();
         i++) {
        cout << *i << " ";
    }
    cout << endl;
    return 0;
}
```

Output

```
4 found
s: 1 2 3 4 5 9
```

1.7 – STL Map

Maps are associative containers used to store the key-value pairs where each key should be unique. It generally implements a red-black tree to store data in sorted order. It is defined as **std::map** inside the **<map>** header file.

> ***Note:*** *To store the multiple keys, we can use the* ***multimap*** *container.*

Map Declaration

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

std::map Member Functions

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 in the map	O(1)

S. No.	Function	Description	Time Complexity
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)
8.	find(key)	Return iterator to the element with the given key if found, otherwise returns iterator to the end.	O(logn)

Example

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

int main()
{

    // creating std::map object
    map<int, string> m;

    // adding elements
    m[1] = "ONE";
    m[2] = "TWO";
    m[3] = "THREE";

    // checking size
    cout << "Size of map m: " << m.size() << endl;

    // inserting using insert pair
    m.insert({ 4, "FOUR" });

    // deleting key 2 with its value
    m.erase(2);

    // printing the map
    cout << "Map:-" << endl;
    for (auto i : m) {
        cout << "Key: " << i.first << '\t';
```

```
        cout << "Value: " << i.second << endl;
    }

    return 0;
}
```

Output

```
Size of map m: 3
Map:-
Key: 1    Value: ONE
Key: 3    Value: THREE
Key: 4    Value: FOUR
```

1.8 – STL Unordered_set

The unordered_set is the version of the set container where the data is not sorted but we can still perform a quick search. It is due to the fact that these unordered_sets are implemented using hash tables. It is defined as **std::unordered_set** inside the **<unordered_set>** header file.

> ***Note:*** *To store the multiple keys, we can use the* ***unordered_multiset*** *container.*

unordered_set Declaration

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

std::unordered_set Member Functions

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)

S. No.	Functions	Description	Time Complexity
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

```cpp
// C++ Program to illustrate the unordered_set container
#include <iostream>
#include <unordered_set>
using namespace std;

int main()
{

    // creating an unordered_set object
    unordered_set<int> us = { 1, 5, 2, 3, 4 };

    // checking size
    cout << "Size of us: " << us.size() << endl;

    // inserting data
    us.insert(7);

    // finding some key
    if (us.find(3) != us.end()) {
        cout << "3 found!" << endl;
    }
    else {
        cout << "3 not found" << endl;
    }

    // traversing unordered_set using iterators
    cout << "us: ";
    for (auto i = us.begin(); i != us.end(); i++) {
        cout << *i << " ";
    }
    cout << endl;
```

```
    return 0;
}
```

Output

```
Size of us: 5
3 found!
us: 7 4 1 5 2 3
```

1.9 – STL Unordered_map

Unordered_maps are stores the data in the form of key-value pairs. They implement the hash table so there is no particular order in which the data is stored. They are defined as **std::unordered_map** class template inside **<unordered_map>** header file.

> ***Note:*** *To store the multiple keys, we can use the* ***unordered_multimap*** *container.*

unordered_map Declaration

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

std::unordered_map Member Functions

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_set is empty, otherwise false.	O(1)

S. No.	Function	Description	Time Complexity
5.	find()	Returns the pointer to the given element if present, otherwise, a pointer to the end.	O(1)
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

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

int main()
{
    // creating unordered_map object
    unordered_map<int, string> umap;

    // inserting key value pairs
    umap[1] = "ONE";
    umap[2] = "TWO";
    umap[3] = "THREE";
    umap.insert({ 4, "FOUR" });

    // finding some key
    if (umap.find(12) != umap.end()) {
        cout << "Key 12 Found!" << endl;
    }
    else {
        cout << "Key 12 Not Found!" << endl;
    }

    // traversing whole map at once using iterators
    cout << "umap:--" << endl;
    for (auto i = umap.begin(); i != umap.end(); i++) {
        cout << "Key:" << i->first
             << "\tValue: " << i->second << endl;
    }

    return 0;
}
```

Output

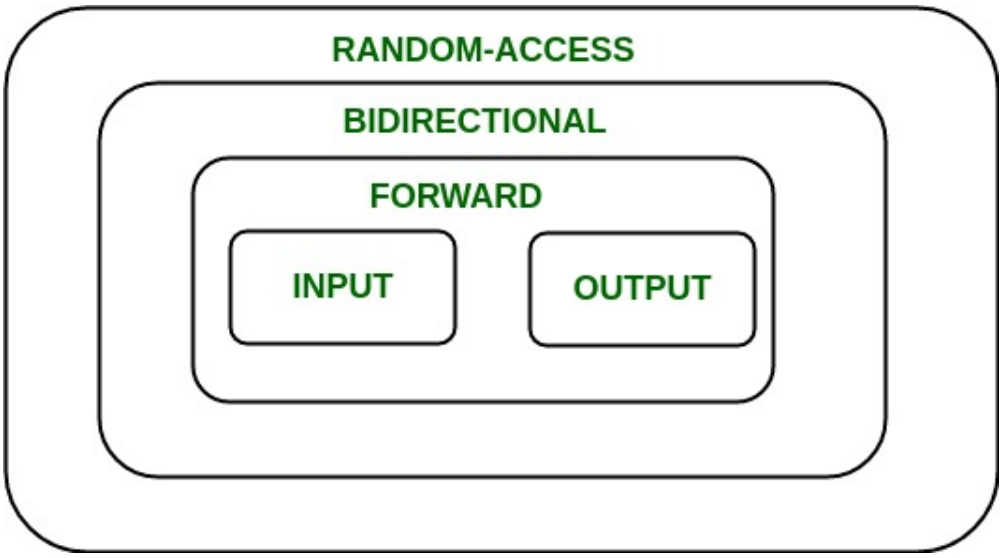
```
Key 12 Not Found!
umap:--
Key:4    Value: FOUR
Key:3    Value: THREE
Key:2    Value: TWO
Key:1    Value: ONE
```

STL Iterators

Iterators are the objects used to iterate through the STL containers. They can be seen as pointers that are used to traverse and manipulate the data inside containers.

- Iterators are defined inside the **<iterator> header file.**
- Each container has its own iterators.

Iterators can be classified into 5 types which are:



STL Iterators

1. Input Iterator

Input Iterators are used for single-pass input operations.

- *They can only be used for accessing (read operations) not assigning.*
- *They cannot be decremented.*
- *An element can only be accessed once.*

- *They have limited capability and come lowest in the iterator hierarchy.*

istream_iterator is an example of the input iterator.

2. Output Iterator

Output Iterators are used for single-pass output operations.

- *They can only be used for assigning purposes (write operations).*
- *An element can only be accessed once.*
- *They cannot be decremented.*
- *They come lowest in the hierarchy along with the Input Iterators.*

ostream_iterator is an example of the output iterator.

3. Forward Iterator

Forward iterators contain features of both input and output iterators along with:

- *It can be used for both read and write operations.*
- *It cannot be decremented as it can move only in a single direction.*
- *It can only move sequentially i.e., one step at a time.*
- *It is in the upper hierarchy compared to both input and output iterators.*

forward_list::iterator are examples of the forward iterators.

4. Bi-Directional Iterator

The bi-directional iterators have all the features of forward iterators along with:

- *They can move in both forward and backward directions.*
- *They can be used for both read and write operations.*

map::iterator, set::iterator, multiset::iterator, and multimap::iterators are some examples of input iterator.

5. Random Access Iterator

Random access iterators are the most powerful iterators.

- *They contain features of all the other iterators.*
- *They can move in both forward and backward directions.*
- *Read and write operations can be performed.*
- *Can move to any point in the container i.e. random access.*

vector::iterator and array::iterator are some examples.

CONTAINER	TYPES OF ITERATOR SUPPORTED
Vector	Random-Access
List	Bidirectional
Deque	Random-Access
Map	Bidirectional
Multimap	Bidirectional
Set	Bidirectional
Multiset	Bidirectional
Stack	No iterator Supported
Queue	No iterator Supported
Priority-Queue	No iterator Supported

STL Algorithms

Algorithms are set of generic and optimal implementations of some useful algorithms to make programming easier and more efficient.

- These algorithms work with STL containers and iterators.
- Algorithms are defined inside the <algorithm> header file.
- C++ STL contains around 114 Algorithms which are listed in the article – Algorithm Library in C++ STL

Some of the commonly used algorithms are:

1. Sort

The std::sort algorithm is used to sort data in any given order.

Syntax of std::sort

```
sort (beginIterator, endIterator);
sort (beginIterator, endIterator, comparator);    // for custom comparator
```

*Note: Iterators must be **RandomAccessIterators**.*

Example

```cpp
// C++ program to demonstrate default behaviour of
// sort() in STL.
#include <bits/stdc++.h>
using namespace std;

int main()
{
    int arr[] = { 1, 5, 8, 9, 6, 7, 3, 4, 2, 0 };
    int n = sizeof(arr) / sizeof(arr[0]);

    /*Here we take two parameters, the beginning of the
    array and the length n upto which we want the array to
    be sorted*/
    sort(arr, arr + n);

    cout << "\nArray after sorting using "
            "default sort is : \n";
    for (int i = 0; i < n; ++i)
        cout << arr[i] << " ";

    return 0;
}
```

Output

```
Array after sorting using default sort is :
0 1 2 3 4 5 6 7 8 9
```

2. Copy

The std::copy method efficiently copies a range of elements to another container using its iterators.

Syntax of std::copy

```
copy (beginIterator, endIterator, destIterator);
```

*Note: Iterators can be of **InputIterator, OutputIterator or ForwardIterator.***

Example

```
// C++ Program to print vector using copy function and input
// and output stream iterators
#include <algorithm>
#include <iostream>
#include <iterator>
#include <sstream>
#include <vector>
using namespace std;

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

    // copying data to ostream
    copy(v.begin(), v.end(),
         ostream_iterator<int>(cout, " "));
    return 0;
}
```

Output

```
1 2 3 4 5
```

3. Max Element

The std::max_element implements an efficient algorithm to find the maximum element in the container. To find minimum element, use std::min_element.

Syntax of std::max_element

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

*Note: The iterators can be of type **ForwardIterators.***

Example

```
// C++ program to demonstrate the use of std::max_element
```

```cpp
// C++ program to demonstrate the use of std::max_element
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>
using namespace std;
int main()
{
    // creating vector
    vector<int> v = { 10, 88, 2, 9, 45, 82, 546, 42, 221 };

    // Finding the maximum value between the first and the
    // fourth element
    auto max = max_element(begin(v), end(v));

    cout << "Maximum Element: " << *max << "\n";
    return 0;
}
```

Output

```
Maximum Element: 546
```

4. Find

The std::find function is used to find the element in the given range.

Syntax of std::find

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

*Note: The iterators can be of the type **InputIterator, ForwardIterator.***

Example

```cpp
// C++ program to illustrate the find()
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

int main()
{

    // creating vector
    vector<int> v
        = { 1, 8, 97, 3, 654, 132, 65, 4, 321, 5, 45 };

    // finding 5
    auto itr = find(v.begin(), v.end(), 5);
    if (itr != v.end()) {
        cout << *itr << " is found!" << endl;
    }
```

```
    else {
        cout << "5 is not found!" << endl;
    }

    return 0;
}
```

Output

```
5 is found!
```

5. For Each

The std::for_each algorithm applies the specified instruction to each of the elements in the given range.

Syntax of std::for_each

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

Note: *The iterators can be of the type* ***ForwardIterator, InputIterator.***

Example

```
// C++ program to print vector using for
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>
using namespace std;

int main()
{

    // creating vector
    vector<int> v = { 1, 2, 3, 4, 5 };

    // adding 1 to each element
    for_each(v.begin(), v.end(), [](int& i){
      i = i + 1;
    });

    // printing vector
    copy(v.begin(), v.end(), ostream_iterator<int>(cout, " "));

    return 0;
}
```

Output

```
2 3 4 5 6
```

STL Function Objects (Functors)

The Function Objects, also known as Functors, are the objects that behave like a function. It is due to the overloading of the **( ) parenthesis operator.** The functors are defined inside the **<functional>** header file.

STL provides some predefined functors such as:

1. equal_to
2. not_equal_to
3. greater
4. less
5. plus
6. minus

Example

```cpp
// C++ program to illustrate some builtin functors
#include <functional>
#include <iostream>
using namespace std;

int main()
{
    // creating objects
    equal_to<int> eq;
    not_equal_to<int> neq;
    greater<int> gt;
    less<int> ls;
    plus<int> p;
    minus<int> m;

    // printing return values
    cout << "Functors and their return value\n";
    cout << boolalpha;
    cout << "equal_to, (10,20): " << eq(10, 20) << endl;
    cout << "greater, (10,20): " << gt(10, 20) << endl;
    cout << "less, (10,20): " << ls(10, 20) << endl;
    cout << "plus, (10,20): " << p(10, 20) << endl;
    cout << "minus(10,20): " << m(10, 20) << endl;

    return 0;
}
```

Output

```
Functors and their return value
equal_to, (10,20): false
```

```
greater, (10,20): false
less, (10,20): true
plus, (10,20): 30
minus(10,20): -10
```


Next Article

Docker Cheat Sheet : Complete Guide (2024)

Similar Reads

Geeksforgeeks Cheatsheets - All Coding Cheat Sheets Collections

Cheatsheets are short documents that contain all the most essential information about a specific technology in short, such as its syntax, commands, functions, or its features. Sheets are designed to help...

4 min read

Subnet Mask Cheat Sheet

A Subnet Mask is a numerical value that describes a computer or device's how to divide an IP address into two parts: the network portion and the host portion. The network element identifies the network to whic...

9 min read

Git Cheat Sheet

Git Cheat Sheet is a comprehensive quick guide for learning Git concepts, from very basic to advanced levels. By this Git Cheat Sheet, our aim is to provide a handy reference tool for both beginners and...

10 min read

NumPy Cheat Sheet: Beginner to Advanced (PDF)

NumPy stands for Numerical Python. It is one of the most important foundational packages for numerical computing & data analysis in Python. Most computational packages providing scientific functionality use...

15+ min read

Linux Commands Cheat Sheet

Linux, often associated with being a complex operating system primarily used by developers, may not necessarily fit that description entirely. While it can initially appear challenging for beginners, once you...

13 min read

Pandas Cheat Sheet for Data Science in Python

Pandas is a powerful and versatile library that allows you to work with data in Python. It offers a range of features and functions that make data analysis fast, easy, and efficient. Whether you are a data scientist,...

15+ min read

Java Cheat Sheet

Java is a programming language and platform that has been widely used since its development by James Gosling in 1991. It follows the Object-oriented Programming concept and can run programs written on a...

15+ min read

C++ STL Cheat Sheet

The C++ STL Cheat Sheet provides short and concise notes on Standard Template Library (STL) in C++. Designed for programmers that want to quickly go through key STL concepts, the STL cheatsheet covers...

15+ min read

Docker Cheat Sheet : Complete Guide (2024)

Docker is a very popular tool introduced to make it easier for developers to create, deploy, and run applications using containers. A container is a utility provided by Docker to package and run an applicatio...

11 min read

C++ Cheatsheet

This is a C++ programming cheat sheet. It is useful for beginners and intermediates looking to learn or revise the concepts of C++ programming. While learning a new language, it feels annoying to switch...

15+ min read