STL: Containers: Sequence and Associative Container - Sequence Container: Vector, List, Deque, Array, Stack - Associative Containers: Map, Multimap - Iterator and Specialized iterator - Functions of iterator - Algorithms: find(), count(), sort() - Algorithms: search(), merge(), for\_each(), transform()

# The C++ Standard Template Library (STL)

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.

The C++ Standard Template Library (STL) is a collection of algorithms, data structures, and other components that can be used to simplify the development of C++ programs. The STL provides a range of containers, such as vectors, lists, and maps, as well as algorithms for searching, sorting and manipulating data.

One of the key benefits of the STL is that it provides a way to write generic, reusable code that can be applied to different data types. This means that you can write an algorithm once, and then use it with different types of data without having to write separate code for each type.

The STL also provides a way to write efficient code. Many of the algorithms and data structures in the STL are implemented using optimized algorithms, which can result in faster execution times compared to custom code.

C++ STL Containers

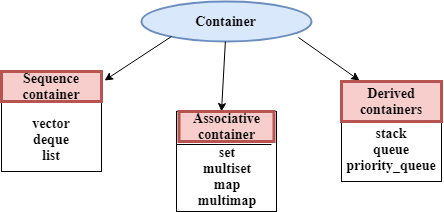
Containers can be described as the objects that hold the data of the same type. Containers are used to implement different data structures for example arrays, list, trees, etc.

Following are the containers that give the details of all the containers as well as the header file and the type of iterator associated with them :

|  |  |  |  |
| --- | --- | --- | --- |
| **Container** | **Description** | **Header file** | **iterator** |
| vector | vector is a class that creates a dynamic array allowing insertions and deletions at the back. | <vector> | Random access |
| list | list is the sequence containers that allow the insertions and deletions from anywhere. | <list> | Bidirectional |
| deque | deque is the double ended queue that allows the insertion and deletion from both the ends. | <deque> | Random access |
| set | set is an associate container for storing unique sets. | <set> | Bidirectional |
| multiset | Multiset is an associate container for storing non- unique sets. | <set> | Bidirectional |
| map | Map is an associate container for storing unique key-value pairs, i.e. each key is associated with only one value(one to one mapping). | <map> | Bidirectional |
| multimap | multimap is an associate container for storing key- value pair, and each key can be associated with more than one value. | <map> | Bidirectional |
| stack | It follows last in first out(LIFO). | <stack> | No iterator |
| queue | It follows first in first out(FIFO). | <queue> | No iterator |
| Priority-queue | First element out is always the highest priority element. | <queue> | No iterator |

**Classification of containers :**

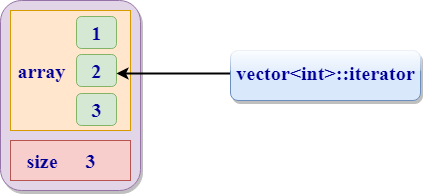
* Sequence containers
* Associative containers
* Derived containers



Note : Each container class contains a set of functions that can be used to manipulate the contents.

ITERATOR

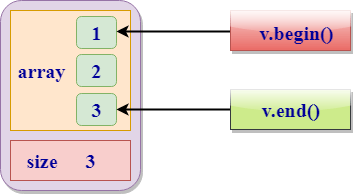
* Iterators are pointer-like entities used to access the individual elements in a container.
* Iterators are moved sequentially from one element to another element. This process is known as iterating through a container.



* Iterator contains mainly two functions:

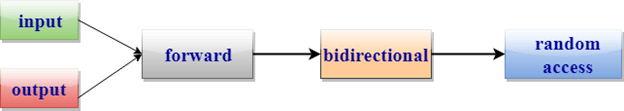
**begin()**: The member function begin() returns an iterator to the first element of the vector.

**end()**: The member function end() returns an iterator to the past-the-last element of a container.



Iterator Categories

Iterators are mainly divided into five categories:



1. **Input iterator:**
   * An Input iterator is an iterator that allows the program to read the values from the container.
   * Dereferencing the input iterator allows us to read a value from the container, but it does not alter the value.
   * An Input iterator is a one way iterator.
   * An Input iterator can be incremented, but it cannot be decremented.
2. **Output iterator:**
   * An output iterator is similar to the input iterator, except that it allows the program to modify a value of the container, but it does not allow to read it.
   * It is a one-way iterator.
   * It is a write only iterator.
3. **Forward iterator:**
   * Forward iterator uses the ++ operator to navigate through the container.
   * Forward iterator goes through each element of a container and one element at a time.
4. **Bidirectional iterator:**
   * A Bidirectional iterator is similar to the forward iterator, except that it also moves in the backward direction.
   * It is a two way iterator.
   * It can be incremented as well as decremented.
5. **Random Access Iterator:**
   * Random access iterator can be used to access the random element of a container.
   * Random access iterator has all the features of a bidirectional iterator, and it also has one more additional feature, i.e., pointer addition. By using the pointer addition operation, we can access the random element of a container.

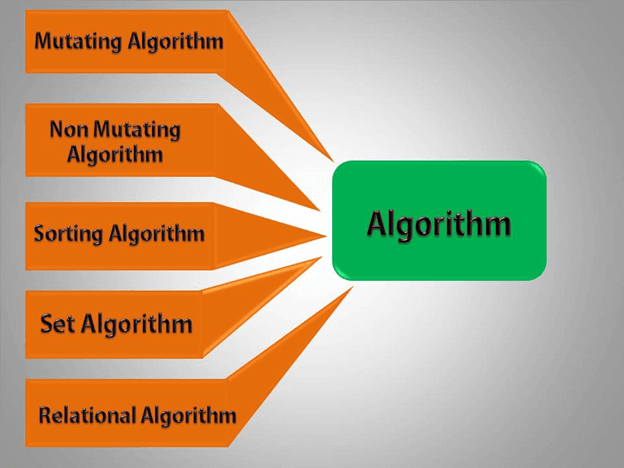
Algorithms

Algorithms are the functions used across a variety of containers for processing its contents.

**Points to Remember:**

* Algorithms provide approx 60 algorithm functions to perform the complex operations.
* Standard algorithms allow us to work with two different types of the container at the same time.
* Algorithms are not the member functions of a container, but they are the standalone template functions.
* Algorithms save a lot of time and effort.
* If we want to access the STL algorithms, we must include the <algorithm> header file in our program.

**STL algorithms can be categorized as:**



* **Nonmutating algorithms**: Nonmutating algorithms are the algorithms that do not alter any value of a container object nor do they change the order of the elements in which they appear. These algorithms can be used for all the container objects, and they make use of the forward iterators.
* **Mutating algorithms**: Mutating algorithms are the algorithms that can be used to alter the value of a container. They can also be used to change the order of the elements in which they appear.
* **Sorting algorithms**: Sorting algorithms are the modifying algorithms used to sort the elements in a container.
* **Set algorithms**: Set algorithms are also known as sorted range algorithm. This algorithm is used to perform some function on a container that greatly improves the efficiency of a program.
* **Relational algorithms**: Relational algorithms are the algorithms used to work on the numerical data. They are mainly designed to perform the mathematical operations to all the elements in a container.

**Sequence Container**

In C++, sequence containers are a group of template classes used to store data elements. Because they are template classes, they can be used to store any data elements, including custom classes. The data structure they implement enables sequential access.

## The array Sequence Container in C++

The array is a collection of homogeneous objects and this array container is defined for constant size arrays or (static size). This container wraps around fixed-size arrays and the information of its size are not lost when declared to a pointer.   
In order to utilize arrays, we need to include the array header:   
  #include <array>

SYNTAX of array container:

array<object\_type, array\_size> array\_name;

### A Quick Example of an Array Sequence in C++

#include <iostream>

#include <iomanip>

#include <array>

#include <iterator>

using namespace std;

int **main**()

{

   array<int, 7> arr = { 0, 1, 1, 2, 3, 5, 8 };

   array<int, 7>::iterator iter;

   for(iter = arr.begin(); iter < arr.end(); iter++)

      cout<<\*iter<<" ";

   cout<<endl;

   iter = arr.begin();

   advance(*iter*,4);

   cout<<\*iter<<endl;

   auto pa = prev(iter,2);

   cout<<\*pa<<endl;

   auto pb = next(iter,1);

   cout<<\*pb<<endl;

   cout<<"Size : "<<arr.size()<<endl;

   return 0;

}

**OUTPUT**

0 1 1 2 3 5 8

3

1

5

Size : 7

**Vectors:**

Vectors are also the data structure which stores the value in the same way as an array does, but they have the capability to resize themselves. Due to increasing the size of itself automatically, the vectors are also known as dynamic arrays.

In the arrays, at the declaration time, we have to tell the number of elements we want to put into the array, but in the case of vectors, we do not need to declare the number of elements because it can increase its size when it is filled completely.

Vectors are defined in the STL (Standard Template Library) of C++, so to use the vectors, we have to import the STL library into the program file.

Since vectors are defined in the STL (Standard Template Library), it has a lot of built-in functions for insertion, deletion or modification at any index, whether it is at starting, ending or at any position.

For example: size(),capacity(),push\_back() etc.

**There are the following inbuilt functions used in vectors:**

1. **size()**: This function returns the number of elements present in the vector.
2. **capacity()**: This function returns the number of elements or the capacity of the vector.
3. **empty()**: It returns whether the current vector is empty or not empty.
4. **push\_back()**: With the help of this function, we can insert an element from the backside of the vector.
5. **pop\_back()**: With the help of this function, we can remove or delete the element from the backside of the vector.
6. **insert()**: With this function, we can insert the element in the vector at any specified position.
7. **erase()**: With the help of this function, we can remove the elements in the vector from the specific position.

**Syntax:**

vector<data\_type> vector\_name;

Example

1. #include<iostream>
2. #include<vector>
3. **using** **namespace** std;
4. **int** main()
5. {
6. vector<string> v1;
7. v1.push\_back("javaTpoint ");
8. v1.push\_back("tutorial");
9. **for**(vector<string>::iterator itr=v1.begin();itr!=v1.end();++itr)
10. cout<<\*itr;
11. **return** 0;
12. }

**Output:**

javaTpoint tutorial

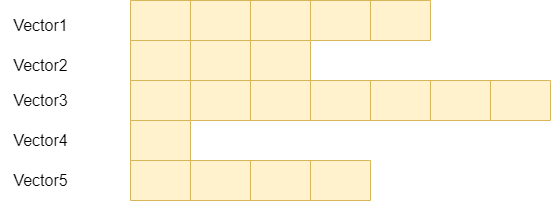
|  |  |
| --- | --- |
| **Function** | **Description** |
| [at()](https://www.javatpoint.com/cpp-vector-at-function) | It provides a reference to an element. |
| [back()](https://www.javatpoint.com/cpp-vector-back-function) | It gives a reference to the last element. |
| [front()](https://www.javatpoint.com/cpp-vector-front-function) | It gives a reference to the first element. |
| [swap()](https://www.javatpoint.com/cpp-vector-swap-function) | It exchanges the elements between two vectors. |
| [push\_back()](https://www.javatpoint.com/cpp-vector-push-back-function) | It adds a new element at the end. |
| [pop\_back()](https://www.javatpoint.com/cpp-vector-pop-back-function) | It removes a last element from the vector. |
| [empty()](https://www.javatpoint.com/cpp-vector-empty-function) | It determines whether the vector is empty or not. |
| [insert()](https://www.javatpoint.com/cpp-vector-insert-function) | It inserts new element at the specified position. |
| [erase()](https://www.javatpoint.com/cpp-vector-erase-function) | It deletes the specified element. |
| [resize()](https://www.javatpoint.com/cpp-vector-resize-function) | It modifies the size of the vector. |
| [clear()](https://www.javatpoint.com/cpp-vector-clear-function) | It removes all the elements from the vector. |
| [size()](https://www.javatpoint.com/cpp-vector-size-function) | It determines a number of elements in the vector. |
| [capacity()](https://www.javatpoint.com/cpp-vector-capacity-function) | It determines the current capacity of the vector. |
| [assign()](https://www.javatpoint.com/cpp-vector-assign-function) | It assigns new values to the vector. |

Arrays of Vectors

Arrays of vectors are basically two-dimensional matrices or arrays where the number of columns can be anything, but the number of rows is fixed.

Since each row represents one vector and one vector can have any number of elements, each row can have any number of columns.

It can look like this:



**Syntax:**

vector<data\_type> array\_name[array\_size];

The syntax is similar to array declaration, but the data type of the array is a vector.

**C++ Example:**

#include <iostream>

#include<vector>

**using** **namespace** std;

**int** main() {

**int** n =5;// number of rows is 5

    vector<**int**> vc[n];

    //inserting in the first row

    vc[0].push\_back(1);

    vc[0].push\_back(2);

    vc[0].push\_back(3);

    vc[0].push\_back(4);

    vc[0].push\_back(5);

    //inserting in the second row

    vc[1].push\_back(6);

    vc[1].push\_back(7);

    vc[1].push\_back(8);

    //inserting in the third row

    vc[2].push\_back(9);

    vc[2].push\_back(10);

    vc[2].push\_back(11);

    vc[2].push\_back(12);

    //inserting in the fourth row

    vc[3].push\_back(13);

    vc[3].push\_back(14);

    //inserting in the fifth row

    vc[4].push\_back(15);

**for**(**int** i=0;i<n;i++){

        cout<<"number of column in row number "<<i+1<<" is "<<vc[i].size()<<endl;

**for**(**int** j=0;j<vc[i].size();j++){

            cout<<vc[i][j]<<" ";

        }

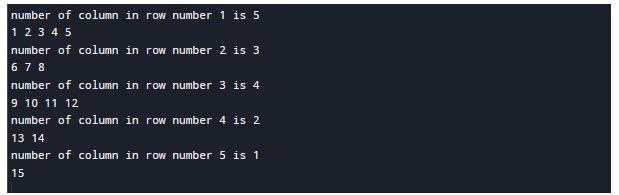
        cout<<endl;

    }

**return** 0;

}

**Output:**

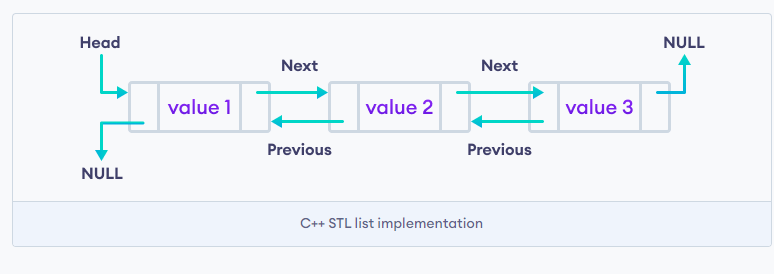


# C++ List

* List is a contiguous container while vector is a non-contiguous container i.e list stores the elements on a contiguous memory and vector stores on a non-contiguous memory.
* Insertion and deletion in the middle of the vector is very costly as it takes lot of time in shifting all the elements. Linklist overcome this problem and it is implemented using list container.
* List supports a bidirectional and provides an efficient way for insertion and deletion operations.
* Traversal is slow in list as list elements are accessed sequentially while vector supports a random access.

C++ List is a STL container that stores elements randomly in unrelated locations. To maintain sequential ordering, every list element includes two links:

* one that points to the previous element
* another that points to the next element



Template for list

1. #include<iostream>
2. #include<list>
3. **using** **namespace** std;
4. **int** main()
5. {
6. list<**int**> l;
7. }

It creates an empty list of integer type values.

**List can also be initalised with the parameters.**

1. #include<iostream>
2. #include<list>
3. **using** **namespace** std;
4. **int** main()
5. {
6. list<**int**> l{1,2,3,4};
7. }

**List can be initialised in two ways.**

1. list<**int**>  new\_list{1,2,3,4};
2. or
3. list<**int**> new\_list = {1,2,3,4};

## C++ List Functions

Following are the member functions of the list:

|  |  |
| --- | --- |
| **Method** | **Description** |
| [insert()](https://www.javatpoint.com/post/cpp-list-insert-function) | It inserts the new element before the position pointed by the iterator. |
| [push\_back()](https://www.javatpoint.com/post/cpp-list-push_back-function) | It adds a new element at the end of the vector. |
| [push\_front()](https://www.javatpoint.com/post/cpp-list-push_front-function) | It adds a new element to the front. |
| [pop\_back()](https://www.javatpoint.com/post/cpp-list-pop_back-function) | It deletes the last element. |
| [pop\_front()](https://www.javatpoint.com/post/cpp-list-pop_front-function) | It deletes the first element. |
| [empty()](https://www.javatpoint.com/post/cpp-list-empty-function) | It checks whether the list is empty or not. |
| [size()](https://www.javatpoint.com/post/cpp-list-size-function) | It finds the number of elements present in the list. |
| [max\_size()](https://www.javatpoint.com/post/cpp-list-max_size-function) | It finds the maximum size of the list. |
| [front()](https://www.javatpoint.com/post/cpp-list-front-function) | It returns the first element of the list. |
| [back()](https://www.javatpoint.com/post/cpp-list-back-function) | It returns the last element of the list. |
| [swap()](https://www.javatpoint.com/post/cpp-list-swap-function) | It swaps two list when the type of both the list are same. |
| [reverse()](https://www.javatpoint.com/post/cpp-list-reverse-function) | It reverses the elements of the list. |
| [sort()](https://www.javatpoint.com/post/cpp-list-sort-function) | It sorts the elements of the list in an increasing order. |
| [merge()](https://www.javatpoint.com/post/cpp-list-merge-function) | It merges the two sorted list. |
| [splice()](https://www.javatpoint.com/post/cpp-list-splice-function) | It inserts a new list into the invoking list. |
| [unique()](https://www.javatpoint.com/post/cpp-list-unique-function) | It removes all the duplicate elements from the list. |
| [resize()](https://www.javatpoint.com/post/cpp-list-resize-function) | It changes the size of the list container. |
| [assign()](https://www.javatpoint.com/post/cpp-list-assign-function) | It assigns a new element to the list container. |
| [emplace()](https://www.javatpoint.com/post/cpp-list-emplace-function) | It inserts a new element at a specified position. |
| [emplace\_back()](https://www.javatpoint.com/post/cpp-list-emplace_back-function) | It inserts a new element at the end of the vector. |
| [emplace\_front()](https://www.javatpoint.com/post/cpp-list-emplace_front-function) | It inserts a new element at the beginning of the list. |

**Example:**

1. #include <iostream>
2. #include<list>
3. **using** **namespace** std;
4. **int** main()
5. {
6. list<**int**> li={1,2,3,4,5};
7. list<**int**> li1={6,7,8,9};
8. list<**int**>::iterator itr=li.begin();
9. li.insert(itr,li1.begin(),li1.end());
10. **for**(itr=li.begin();itr!=li.end();++itr)
11. {
12. cout<<\*itr;
13. cout<<? ?;
14. }
15. **return** 0;
16. }

**Output:**

6 7 8 9 1 2 3 4 5

*Example explaining List STL functions*

*// Example explaining List STL functions*

#include <iostream>

#include <iterator>

#include <list>

using namespace std;

*// printing the elements of a list*

void print(list < int > lst) {

list < int > ::iterator it;

for (it = lst.begin(); it != lst.end(); ++it)

cout << \* it << " ";

cout << '\n';

}

*// Driver Code*

int main() {

list < int > list1, list2;

for (int i = 0; i < 5; ++i) {

list1.push\_back(i);

list2.push\_front(i + 5);

}

cout << "\nList 1 (list1) is : ";

print(list1);

cout << "\nList 2 (list2) is : ";

print(list2);

cout << "\nlist1.front() : " << list1.front();

cout << "\nlist1.back() : " << list1.back();

cout << "\nlist1.pop\_front() : ";

list1.pop\_front();

print(list1);

cout << "\nlist2.pop\_back() : ";

list2.pop\_back();

print(list2);

cout << "\nlist1.reverse() : ";

list1.reverse();

print(list1);

return 0;

}

**Output:**

List 1 (list1) is : 0 1 2 3 4

List 2 (list2) is : 9 8 7 6 5

list1.front() : 0

list1.back() : 4

list1.pop\_front() : 1 2 3 4

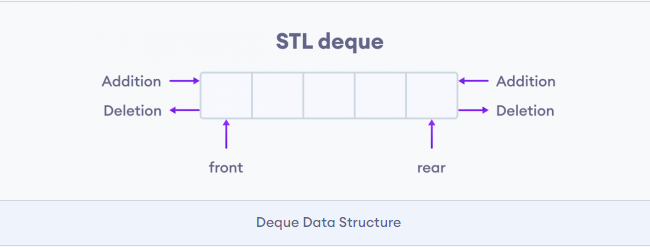
list2.pop\_back() : 9 8 7 6

list1.reverse() : 4 3 2 1

# Deque in C++ Standard Template Library (STL)

Double Ended Queues are basically an implementation of the data structure double-ended queue. A queue data structure allows insertion only at the end and deletion from the front. This is like a queue in real life, wherein people are removed from the front and added at the back. Double-ended queues are a special case of queues where insertion and deletion operations are possible at both the ends.

The functions for deque are same as [vector](https://www.geeksforgeeks.org/vector-in-cpp-stl/), with an addition of push and pop operations for both front and back.



**C++ Deque Methods**

In C++, the deque class provides various methods to perform different operations on a deque.

|  |  |
| --- | --- |
| Methods | Description |
| push\_back() | inserts element at the back |
| push\_front() | inserts element at the front |
| pop\_back() | removes element from the back |
| pop\_front() | removes element from the front |
| front() | returns the element at the front |
| back() | returns the element at the back |
| at() | sets/returns the element at specified index |
| size() | returns the number of elements |
| empty() | returns true if the deque is empty |

#include <deque>

#include <iostream>

using namespace std;

void showdq(deque<int> g)

{

    deque<int>::iterator it;

    for (it = g.begin(); it != g.end(); ++it)

        cout << '\t' << \*it;

    cout << '\n';

}

int main()

{

    deque<int> gquiz;

    gquiz.push\_back(10);

    gquiz.push\_front(20);

    gquiz.push\_back(30);

    gquiz.push\_front(15);

    cout << "The deque gquiz is : ";

    showdq(gquiz);

    cout << "\ngquiz.size() : " << gquiz.size();

    cout << "\ngquiz.max\_size() : " << gquiz.max\_size();

    cout << "\ngquiz.at(2) : " << gquiz.at(2);

    cout << "\ngquiz.front() : " << gquiz.front();

    cout << "\ngquiz.back() : " << gquiz.back();

    cout << "\ngquiz.pop\_front() : ";

    gquiz.pop\_front();

    showdq(gquiz);

    cout << "\ngquiz.pop\_back() : ";

    gquiz.pop\_back();

    showdq(gquiz);

    return 0;

}

**Output**

The deque gquiz is : 15 20 10 30

gquiz.size() : 4

gquiz.max\_size() : 4611686018427387903

gquiz.at(2) : 10

gquiz.front() : 15

gquiz.back() : 30

gquiz.pop\_front() : 20 10 30

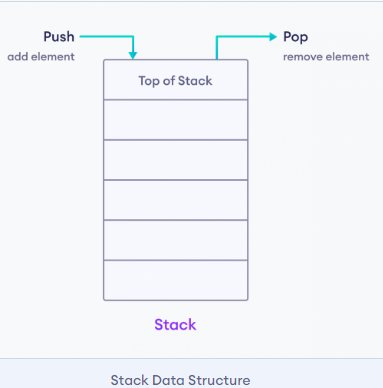
gquiz.pop\_back() : 20 10

# Stack in C++ STL

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.  Stack uses an encapsulated object of either [vector](https://www.geeksforgeeks.org/vector-in-cpp-stl/)or [deque](https://www.geeksforgeeks.org/deque-cpp-stl/)(by default) or [list](https://www.geeksforgeeks.org/list-cpp-stl/)(sequential container class) as its underlying container, providing a specific set of member functions to access its elements.

Once we import this file, we can create a stack using the following syntax:

stack<type> st;



**Stack Methods**

In C++, the stack class provides various methods to perform different operations on a stack.

|  |  |
| --- | --- |
| Operation | Description |
| push() | adds an element into the stack |
| pop() | removes an element from the stack |
| top() | returns the element at the top of the stack |
| size() | returns the number of elements in the stack |
| empty() | returns true if the stack is empty |

#include <iostream>

#include <stack>

using namespace std;

int main() {

stack<int> stack;

stack.push(21);// The values pushed in the stack should be of the same data which is written during declaration of stack

stack.push(22);

stack.push(24);

stack.push(25);

int num=0;

stack.push(num);

stack.pop();

stack.pop();

stack.pop();

while (!stack.empty()) {

cout << stack.top() <<" ";

stack.pop();

}

}

Output

22 21

#include <iostream>

#include <stack>

using namespace std;

int main() {

// create a stack of strings

stack<string> languages;

// add element to the Stack

languages.push("C++");

languages.push("Java");

languages.push("Python");

// print top element

cout << languages.top();

return 0;

}

**Output**

Python

In the above example, we have created a stack of strings named languages.

Here, we have used the push() method to add elements to the stack. We have then used the top() method to display the top element.

**Associative containers**

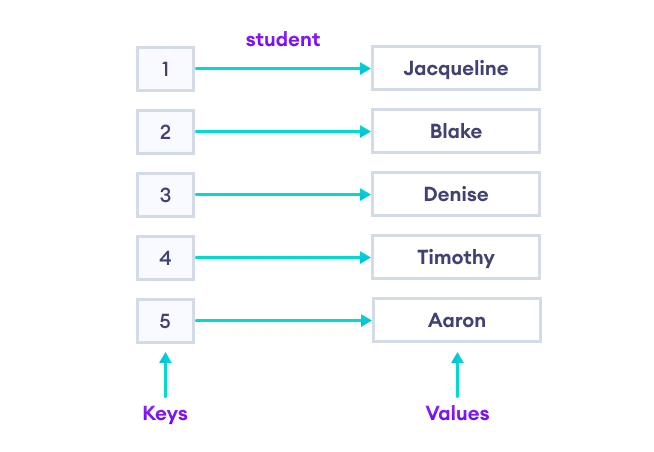
Associative containers are those that provide direct access to its elements for storage and retrieval purposes. The elements are accessed via keys, also known as search keys. The set associative containers store values in such a way that the storage and retrieval of the container object is fast.

**Types of Associative Containers**

* *Set*
* *Map*
* *Multiset*
* *Multimap*

**MAP**

Maps are part of the C++ STL (Standard Template Library). Maps are the associative containers that store sorted key-value pair, in which each key is unique and it can be inserted or deleted but cannot be altered. Values associated with keys can be changed.

s

**For example:** A map of Employees where employee ID is the key and name is the value can be represented as:

|  |  |
| --- | --- |
| Keys | Values |
| 101 | Nikita |
| 102 | Robin |
| 103 | Deep |
| 104 | John |

## Creating a Map in C++ STL

Maps can easily be created using the following statement :

map<key\_type , value\_type> map\_name;

## This will create a map with key of type Key\_type and value of type value\_type. One thing which is to remembered is that key of a map and corresponding values are always inserted as a pair, you cannot insert only key or just a value in a map.

## Header for Map in C++

#include <map>

## Creating a Map

A map in C++ can be easily created using the header file and the syntax that we discussed above, let us take a look at an example to see how it can be created.

We can create a map to store the roll numbers corresponding to the names of every student in the class.

#include <iostream>

// To use the map data structure

#include <map>

using namespace std;

int main ()

{

// Create a map with strings as the key and the integer type as the values

// Names of the students will be of the data type string and the roll numbers as the integer types

map<string, int> mp;

// Assign values to the map

mp["Asia"] = 1; // Inserts key = "Asia" with value = 1

mp["Europe"] = 2; // Inserts key = "Europe" with value = 2

mp["Australia"] = 3; // Inserts key = "Australia" with value = 3

mp["Antarctica"] = 4 ; // Inserts key = "Antarctica" with value = 4

// We can retrieve the roll number or the values corresponding to the keys quickly without going through the whole array

cout << "The roll number of Antarctica is: " << mp["Antarctica"] << endl;

cout << "The roll number of Europe is: " << mp["Europe"] << endl;

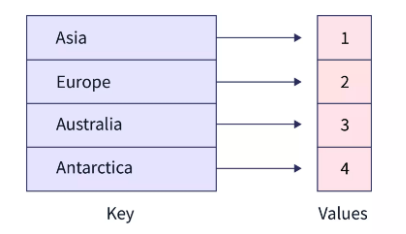
return 0;

}

**Output**

The roll number of Antarctica is: 4

The roll number of Europe is: 2



**Access Keys and Values**

#include <iostream>

#include <map>

using namespace std;

int main() {

map<int, string> student;

student[1] = "Jacqueline";

student[2] = "Blake";

student[3] = "Denise";

student[4] = "Aaron";

// declare map iterator

map<int, string>::iterator iter;

// use iterator to display map elements

for (iter = student.begin(); iter != student.end(); ++iter) {

cout << iter->first << " - " << iter->second << endl;

}

return 0;

}

**Output**

1 - Jacqueline

2 - Blake

3 - Denise

4 - Aaron

### erase() to Remove a Range of Elements

#include <iostream>

#include <map>

using namespace std;

int main() {

// create a map named student

map <int, string> student {{1, "Denise"}, {2, "Blake"}, {3, "Courtney"}, {4, "John"}, {5, "Jennifer"}};

// create a map iterator

map <int, string>::iterator iter;

// display initial map values

cout << "Initial Values:" << endl;

for(iter = student.begin(); iter != student.end(); ++iter) {

cout << iter->second << ", ";

}

cout << endl;

// remove a range of elements

student.erase(student.find(2),student.find(5));

// display final map values

cout << "\nFinal Values:" << endl;

for(iter = student.begin(); iter != student.end(); ++iter) {

cout << iter->second << ", ";

}

return 0;

}

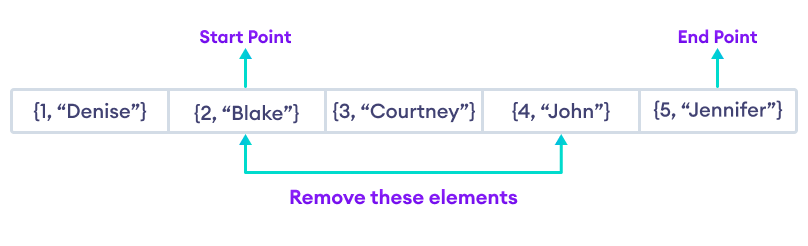
**Output**

Initial Values:

Denise, Blake, Courtney, John, Jennifer,

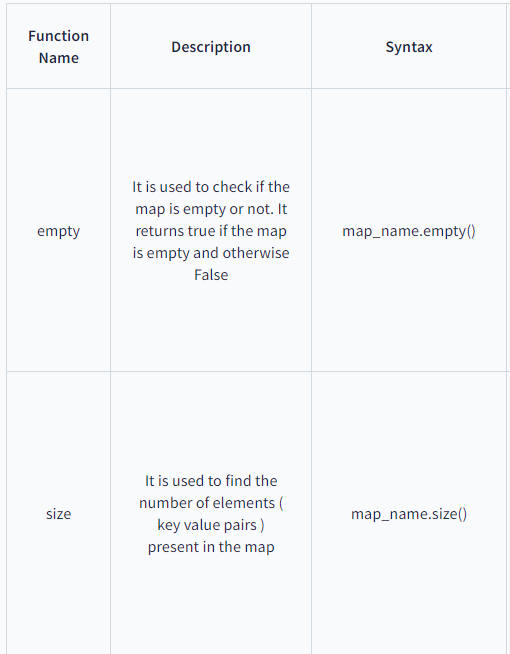
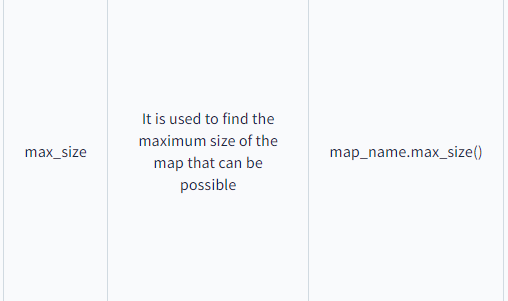
Final Values:

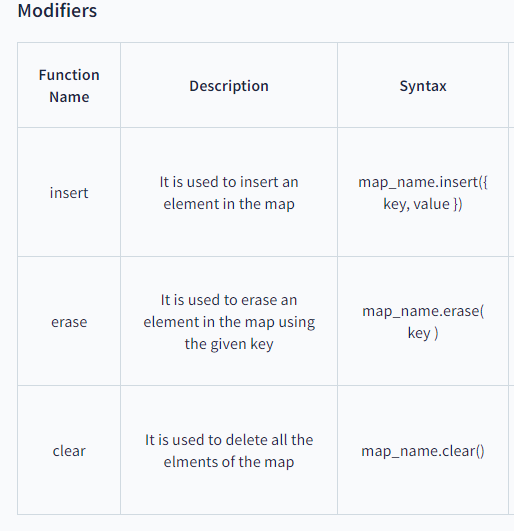
Denise, Jennifer,



**Some basic functions associated with Map:**

* [begin()](https://www.geeksforgeeks.org/mapbegin-end-c-stl/) – Returns an iterator to the first element in the map.
* [end()](https://www.geeksforgeeks.org/mapbegin-end-c-stl/) – Returns an iterator to the theoretical element that follows the last element in the map.
* [size()](https://www.geeksforgeeks.org/mapsize-c-stl/) – Returns the number of elements in the map.
* [max\_size()](https://www.geeksforgeeks.org/map-max_size-in-c-stl/) – Returns the maximum number of elements that the map can hold.
* [empty()](https://www.geeksforgeeks.org/mapempty-c-stl/) – Returns whether the map is empty.
* [pair insert(keyvalue, mapvalue)](https://www.geeksforgeeks.org/map-insert-in-c-stl/) – Adds a new element to the map.
* [erase(iterator position)](https://www.geeksforgeeks.org/map-erase-function-in-c-stl/) – Removes the element at the position pointed by the iterator.
* [erase(const g)](https://www.geeksforgeeks.org/map-erase-function-in-c-stl/)– Removes the key-value ‘g’ from the map.
* [clear()](https://www.geeksforgeeks.org/mapclear-c-stl/) – Removes all the elements from the map.



# C++ multimap

**Multimaps** are part of the **C++ STL (Standard Template Library)**. Multimaps are the associative containers like map that stores sorted key-value pair, but unlike maps which store only unique keys, **multimap can have duplicate keys**. By default it uses < operator to compare the keys.

**For example**: A multimap of Employees where employee age is the key and name is the value can be represented as:

|  |  |
| --- | --- |
| **Keys** | **Values** |
| 23 | Nikita |
| 28 | Robin |
| 25 | Deep |
| 25 | Aman |

**Some Basic Functions associated with multimap:**

* [begin()](https://www.geeksforgeeks.org/multimapbegin-and-multimapend-in-c-stl/) – Returns an iterator to the first element in the multimap
* [end()](https://www.geeksforgeeks.org/multimapbegin-and-multimapend-in-c-stl/) – Returns an iterator to the theoretical element that follows last element in the multimap
* [size()](https://www.geeksforgeeks.org/multimap-size-function-in-c-stl/) – Returns the number of elements in the multimap
* [max\_size()](https://www.geeksforgeeks.org/multimap-maxsize-in-c-stl/) – Returns the maximum number of elements that the multimap can hold
* [empty()](https://www.geeksforgeeks.org/multimap-empty-function-in-c-stl/) – Returns whether the multimap is empty
* [pair<int,int> insert(keyvalue,multimapvalue)](https://www.geeksforgeeks.org/multimap-insert-in-c-stl/) – Adds a new element to the multimap

**Example**

// CPP Program to demonstrate the implementation of multimap

#include <iostream>

#include <iterator>

#include <map>

using namespace std;

// Driver Code

int main()

{

multimap<int, int> gquiz1; // empty multimap container

// insert elements in random order

gquiz1.insert(pair<int, int>(1, 40));

gquiz1.insert(pair<int, int>(2, 30));

gquiz1.insert(pair<int, int>(3, 60));

gquiz1.insert(pair<int, int>(6, 50));

gquiz1.insert(pair<int, int>(6, 10));

// printing multimap gquiz1

multimap<int, int>::iterator itr;

cout << "\nThe multimap gquiz1 is : \n";

cout << "\tKEY\tELEMENT\n";

for (itr = gquiz1.begin(); itr != gquiz1.end(); ++itr) {

cout << '\t' << itr->first << '\t' << itr->second

<< '\n';

}

cout << endl;

// adding elements randomly,

// to check the sorted keys property

gquiz1.insert(pair<int, int>(4, 50));

gquiz1.insert(pair<int, int>(5, 10));

// printing multimap gquiz1 again

cout << "\nThe multimap gquiz1 after adding extra "

"elements is : \n";

cout << "\tKEY\tELEMENT\n";

for (itr = gquiz1.begin(); itr != gquiz1.end(); ++itr) {

cout << '\t' << itr->first << '\t' << itr->second

<< '\n';

}

cout << endl;

// assigning the elements from gquiz1 to gquiz2

multimap<int, int> gquiz2(gquiz1.begin(), gquiz1.end());

// print all elements of the multimap gquiz2

cout << "\nThe multimap gquiz2 after assign from "

"gquiz1 is : \n";

cout << "\tKEY\tELEMENT\n";

for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr) {

cout << '\t' << itr->first << '\t' << itr->second

<< '\n';

}

cout << endl;

// remove all elements up to

// key with value 3 in gquiz2

cout << "\ngquiz2 after removal of elements less than "

"key=3 : \n";

cout << "\tKEY\tELEMENT\n";

gquiz2.erase(gquiz2.begin(), gquiz2.find(3));

for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr) {

cout << '\t' << itr->first << '\t' << itr->second

<< '\n';

}

// remove all elements with key = 4

int num;

num = gquiz2.erase(4);

cout << "\ngquiz2.erase(4) : ";

cout << num << " removed \n";

cout << "\tKEY\tELEMENT\n";

for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr) {

cout << '\t' << itr->first << '\t' << itr->second

<< '\n';

}

cout << endl;

// lower bound and upper bound for multimap gquiz1 key =

// 5

cout << "gquiz1.lower\_bound(5) : "

<< "\tKEY = ";

cout << gquiz1.lower\_bound(5)->first << '\t';

cout << "\tELEMENT = " << gquiz1.lower\_bound(5)->second

<< endl;

cout << "gquiz1.upper\_bound(5) : "

<< "\tKEY = ";

cout << gquiz1.upper\_bound(5)->first << '\t';

cout << "\tELEMENT = " << gquiz1.upper\_bound(5)->second

<< endl;

return 0;

}

The multimap gquiz1 is :

KEY ELEMENT

1 40

2 30

3 60

6 50

6 10

The multimap gquiz1 after adding extra elements is :

KEY ELEMENT

1 40

2 30

3 60

4 50

5 10

6 50

6 10

The multimap gquiz2 after assign from gquiz1 is :

KEY ELEMENT

1 40

2 30

3 60

4 50

5 10

6 50

6 10

gquiz2 after removal of elements less than key=3 :

KEY ELEMENT

3 60

4 50

5 10

6 50

6 10

gquiz2.erase(4) : 1 removed

KEY ELEMENT

3 60

5 10

6 50

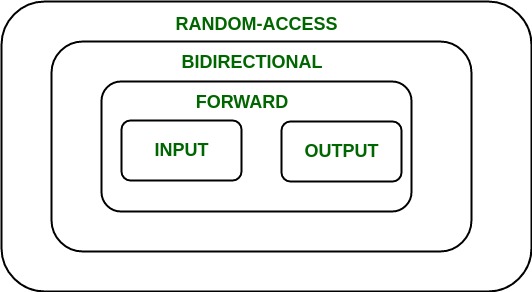
6 10

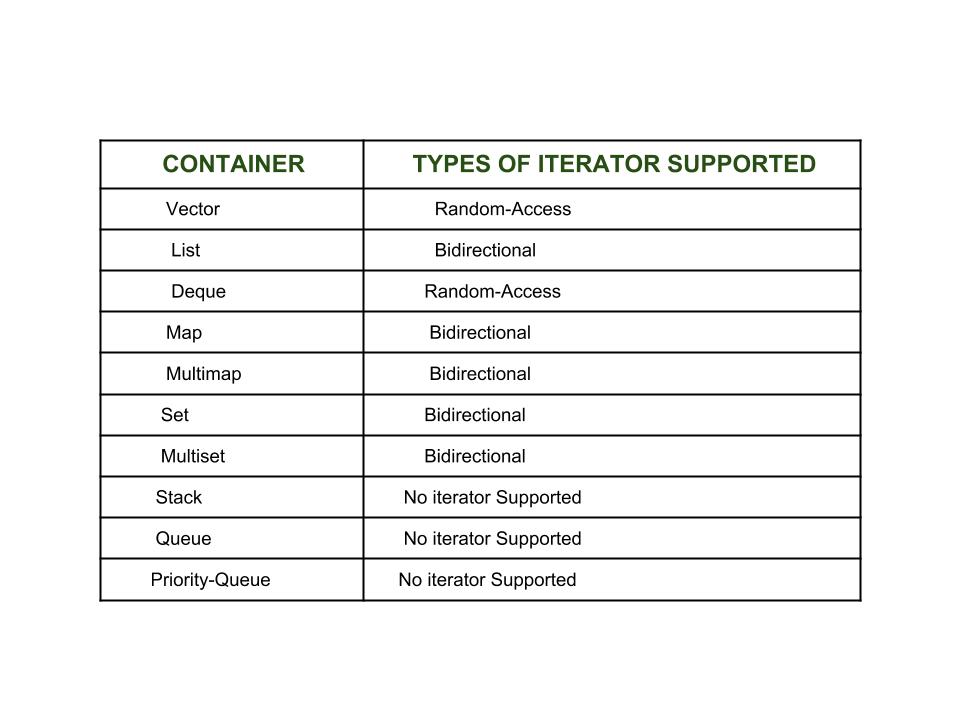
gquiz1.lower\_bound(5) : KEY = 5 ELEMENT = 10

gquiz1.upper\_bound(5) : KEY = 6 ELEMENT = 50

**ITERATOR**

An **iterator**is an object (like a pointer) that points to an element inside the container. We can use iterators to move through the contents of the container. They can be visualized as something similar to a pointer pointing to some location and we can access the content at that particular location using them. Iterators play a critical role in connecting algorithm with containers along with the manipulation of data stored inside the containers. The most obvious form of an iterator is a pointer. A pointer can point to elements in an array and can iterate through them using the increment operator (++). But, all iterators do not have similar functionality as that of pointers. Depending upon the functionality of iterators they can be classified into five categories, as shown in the diagram below with the outer one being the most powerful one and consequently the inner one is the least powerful in terms of functionality.



Now each one of these iterators are not supported by all the containers in STL, different containers support different iterators, like vectors support [Random-access iterators](https://www.geeksforgeeks.org/random-access-iterators-in-cpp/), while lists support [bidirectional iterators.](https://www.geeksforgeeks.org/bidirectional-iterators-in-cpp/) The whole list is as given below: 

**Types of iterators**: Based upon the functionality of the iterators, they can be classified into five major categories:

1. [**Input Iterators**](https://www.geeksforgeeks.org/input-iterators-in-cpp/): They are the weakest of all the iterators and have very limited functionality. They can only be used in a single-pass algorithms, i.e., those algorithms which process the container sequentially, such that no element is accessed more than once.
2. [**Output Iterators**](https://www.geeksforgeeks.org/output-iterators-cpp/): Just like [input iterators](https://www.geeksforgeeks.org/input-iterators-in-cpp/), they are also very limited in their functionality and can only be used in single-pass algorithm, but not for accessing elements, but for being assigned elements.
3. [**Forward Iterator**](https://www.geeksforgeeks.org/forward-iterators-in-cpp/): They are higher in the hierarchy than[input](https://www.geeksforgeeks.org/input-iterators-in-cpp/) and [output iterators](https://www.geeksforgeeks.org/output-iterators-cpp/), and contain all the features present in these two iterators. But, as the name suggests, they also can only move in a forward direction and that too one step at a time.
4. [**Bidirectional Iterators**](https://www.geeksforgeeks.org/bidirectional-iterators-in-cpp/): They have all the features of[forward iterators](https://www.geeksforgeeks.org/forward-iterators-in-cpp/) along with the fact that they overcome the drawback of [forward iterators](https://www.geeksforgeeks.org/forward-iterators-in-cpp/), as they can move in both the directions, that is why their name is bidirectional.
5. [**Random-Access Iterators**](https://www.geeksforgeeks.org/random-access-iterators-in-cpp/): They are the most powerful iterators. They are not limited to moving sequentially, as their name suggests, they can randomly access any element inside the container. They are the ones whose functionality are same as pointers.

Benefits of Iterators

There are certainly quite a few ways which show that iterators are extremely useful to us and encourage us to use it profoundly. Some of the benefits of using iterators are as listed below:

* Convenience in programming: It is better to use iterators to iterate through the contents of containers as if we will not use an iterator and access elements using [ ] operator, then we need to be always worried about the size of the container, whereas with iterators we can simply use member function end() and iterate through the contents without having to keep anything in mind.

**FIND()**

## Description

The C++ function **std::algorithm::find()** finds the first occurrence of the element. It uses operator = for comparison.

## Declaration

Following is the declaration for std::algorithm::find() function form std::algorithm header.

template <class InputIterator, class T>

InputIterator find (InputIterator first, InputIterator last, const T& val);

## Parameters

* **first** − Input iterator to the initial position.
* **last** − Input iterator to the final position.
* **val** − Value to compare the elements.

## Return value

If element found it returns an iterator pointing to the first occurrence of the element otherwise returns last.

.

## Example

The following example shows the usage of std::algorithm::find() function.

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

int main(void) {

int val = 5;

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

auto result = find(v.begin(), v.end(), val);

if (result != end(v))

cout << "Vector contains element " << val << endl;

val = 15;

result = find(v.begin(), v.end(), val);

if (result == end(v))

cout << "Vector doesn't contain element " << val << endl;

return 0;

}

Let us compile and run the above program, this will produce the following result −

Vector contains element 5

Vector doesn't contain element 15

**SORT**

#include<iostream>

#include<algorithm>

#include<vector>

using namespace std;

int main()

{

vector<int> v1;

v1.push\_back(8);

v1.push\_back(4);

v1.push\_back(5);

v1.push\_back(1);

/\* now the vector v1 is 8,4,5,1 \*/

vector<int>::iterator i, j,itr;

i = v1.begin(); // i now points to beginning of the vector v1

j = v1.end(); // j now points to end of the vector v1

sort(i,j); //sort(v1.begin() , v1.end() ) can also be used

for(itr=v1.begin();itr!=v1.end();++itr)

cout<<\*itr;

/\* now the vector v1 is 1,4,5,8 \*/

}

# std::count() in C++ STL

std::count() returns the number of occurrences of an element in a given range. Returns the number of elements in the range [first, last) that compare equal to val.

If the val is not found at any occurrence then it returns 0(Integer value).

|  |
| --- |
| #include <bits/stdc++.h>  using namespace std;    int main()  {      vector<int> vect{ 3, 2, 1, 3, 3, 5, 3 };      cout << "Number of times 3 appears : "           << count(vect.begin(), vect.end(), 3);        return 0;  } |

**Output**

Number of times 3 appears : 4

# std::search in C++

**std::search** is defined in the header file <algorithm> and used to find out the presence of a subsequence satisfying a condition (equality if no such predicate is defined) with respect to another sequence

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

// C++ program to demonstrate the use of std::search

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

int main()

{

int i, j;

// Declaring the sequence to be searched into

vector<int> v1 = { 1, 2, 3, 4, 5, 6, 7 };

// Declaring the subsequence to be searched for

vector<int> v2 = { 3, 4, 5 };

// Declaring an iterator for storing the returning pointer

vector<int>::iterator i1;

// Using std::search and storing the result in

// iterator i1

i1 = std::search(v1.begin(), v1.end(), v2.begin(), v2.end());

// checking if iterator i1 contains end pointer of v1 or not

if (i1 != v1.end()) {

cout << "vector2 is present at index " << (i1 - v1.begin());

} else {

cout << "vector2 is not present in vector1";

}

return 0;

}

Output:

vector2 is present at index 2

# merge() in C++ STL

C++ offers in its STL library a merge() which is quite useful to **merge sort two containers** into a **single** container. It is defined in header “**algorithm**“. It is implemented in two ways

// C++ code to demonstrate the working of

// merge() implementation 1

#include <bits/stdc++.h>

using namespace std;

int main()

{

// initializing 1st container

vector<int> arr1 = { 1, 4, 6, 3, 2 };

// initializing 2nd container

vector<int> arr2 = { 6, 2, 5, 7, 1 };

// declaring resultant container

vector<int> arr3(10);

// sorting initial containers

sort(arr1.begin(), arr1.end());

sort(arr2.begin(), arr2.end());

// using merge() to merge the initial containers

merge(arr1.begin(), arr1.end(), arr2.begin(), arr2.end(), arr3.begin());

// printing the resultant merged container

cout << "The container after merging initial containers is : ";

for (int i = 0; i < arr3.size(); i++)

cout << arr3[i] << " ";

return 0;

}

Output:

The container after merging initial containers is : 1 1 2 2 3 4 5 6 6 7

**for\_each()**

* In C++, for\_each() is a function template defined in the <algorithm> header file of the standard library. It is used to apply a given function or function object to each element of a range of elements, such as a vector, an array, or any other container that supports the iterator concept.
* The for\_each() function takes three arguments:

1. The beginning iterator of the range to be processed.
2. The ending iterator of the range to be processed.
3. The function or function object to be applied to each element of the range.

When for\_each() is called, it applies the given function or function object to each element of the range in turn, from the beginning to the end, using the iterators to access the elements.

For example, suppose we have a vector v containing the integers {1, 2, 3, 4}. We can use for\_each() to print each element of the vector to the console like this:

**Example:**

#include <iostream>

#include <algorithm>

#include <vector>

using namespace std;

void print\_element(int n) {

cout << n << " ";

}

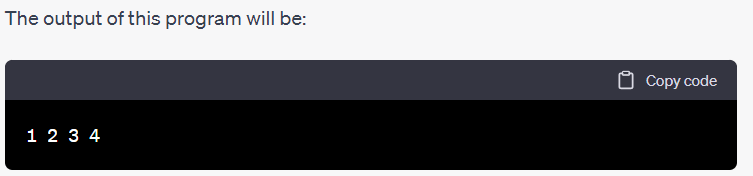
int main() {

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

for\_each(v.begin(), v.end(), print\_element);

return 0;

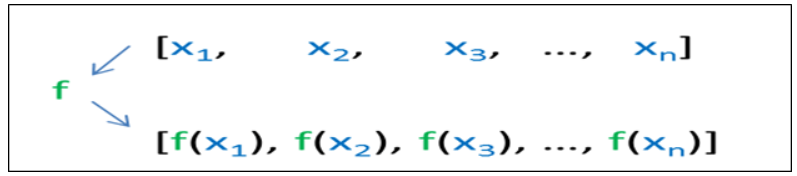
}



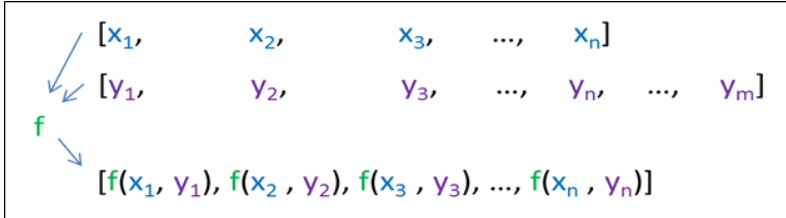
**C++ Algorithm transform():**

**1.unary operation**:- This method performs unary operation **op** on the elements in range [first1, last1] and stores the result in range starting from **result**.

This transform() applies a function to each element of a range:



**2.Binary operation**:- This method performs binary operation **binary\_op** on the elements in range [first1, last1] with the elements in the range starting with iterator **first2** and stores the result in range starting from **result**.



#include <iostream>

#include <algorithm>

#include <string>

using namespace std;

int main() {

string str = "hello, world!";

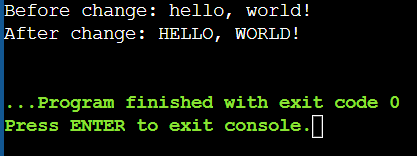
cout<<"Before change: "<<str<<endl;

transform(str.begin(), str.end(), str.begin(), [](unsigned char c) { return toupper(c); });

cout <<"After change: "<<str <<endl;

return 0;

}



[](unsigned char c) { return toupper(c); }

This is a lambda function that takes an unsigned char **c** as its argument and returns its uppercase equivalent using the **toupper** function. The square brackets **[]** indicate the start of the lambda function definition. Inside the square brackets, we can include variables or expressions that will be captured by the lambda function. In this case, we're not capturing anything, so the square brackets are empty.

After the square brackets, we have the parameter list **(unsigned char c)**, which defines the input parameter of the lambda function. In this case, we're taking a single unsigned char as input.

Finally, we have the function body enclosed in curly braces **{}**. In this case, we're calling the **toupper** function on the input character and returning the result.