

C++ STL (Standard Tag Library)

C++ STL Containers

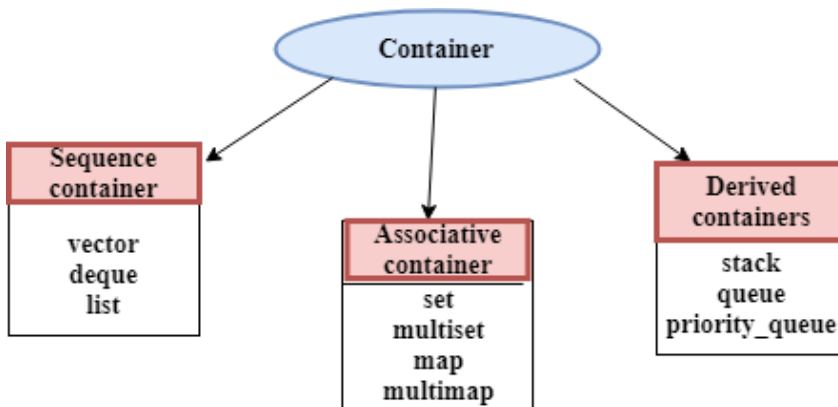
Containers can be described as the objects that hold the data of the same type. Containers are used to implement different data structures for example arrays, list, trees, etc.

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 :

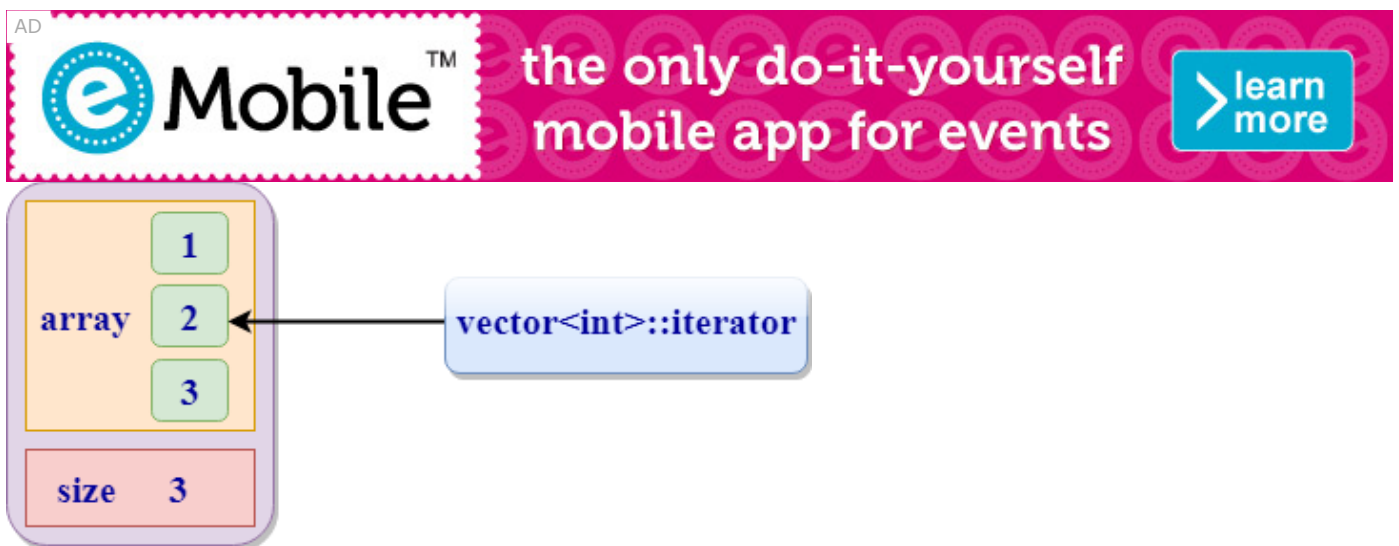
- Sequence containers
- Associative containers
- Derived containers



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

ITERATOR

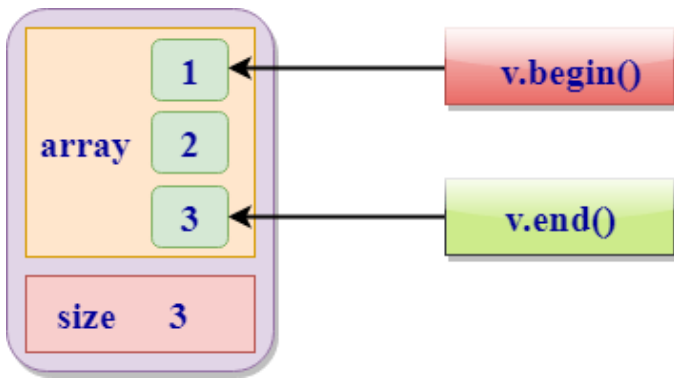
- Iterators are pointer-like entities used to access the individual elements in a container.
- Iterators are moved sequentially from one element to another element. This process is known as iterating through a container.



- Iterator contains mainly two functions:

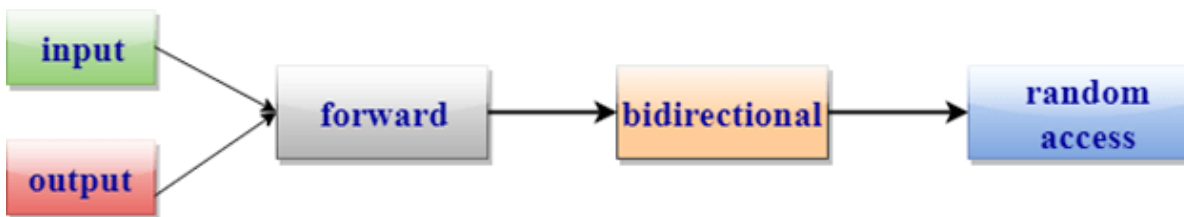
begin(): The member function `begin()` returns an iterator to the first element of the vector.

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.

Operations supported by iterators

iterator	Element access	Read	Write	Increment operation	Comparison
input	->	v = *p		++	==,!=
output			*p = v	++	
forward	->	v = *p	*p = v	++	==,!=
Bidirectional	->	v = *p	*p = v	++,--	==,!=
Random access	->,[]	v = *p	*p = v	++,--,+,--,+=,--=	==,!=,<,>,<=,>=

AD

UBER

TAP A BUTTON. GET A RIDE.
Sign up here for **\$20** off your first trip!

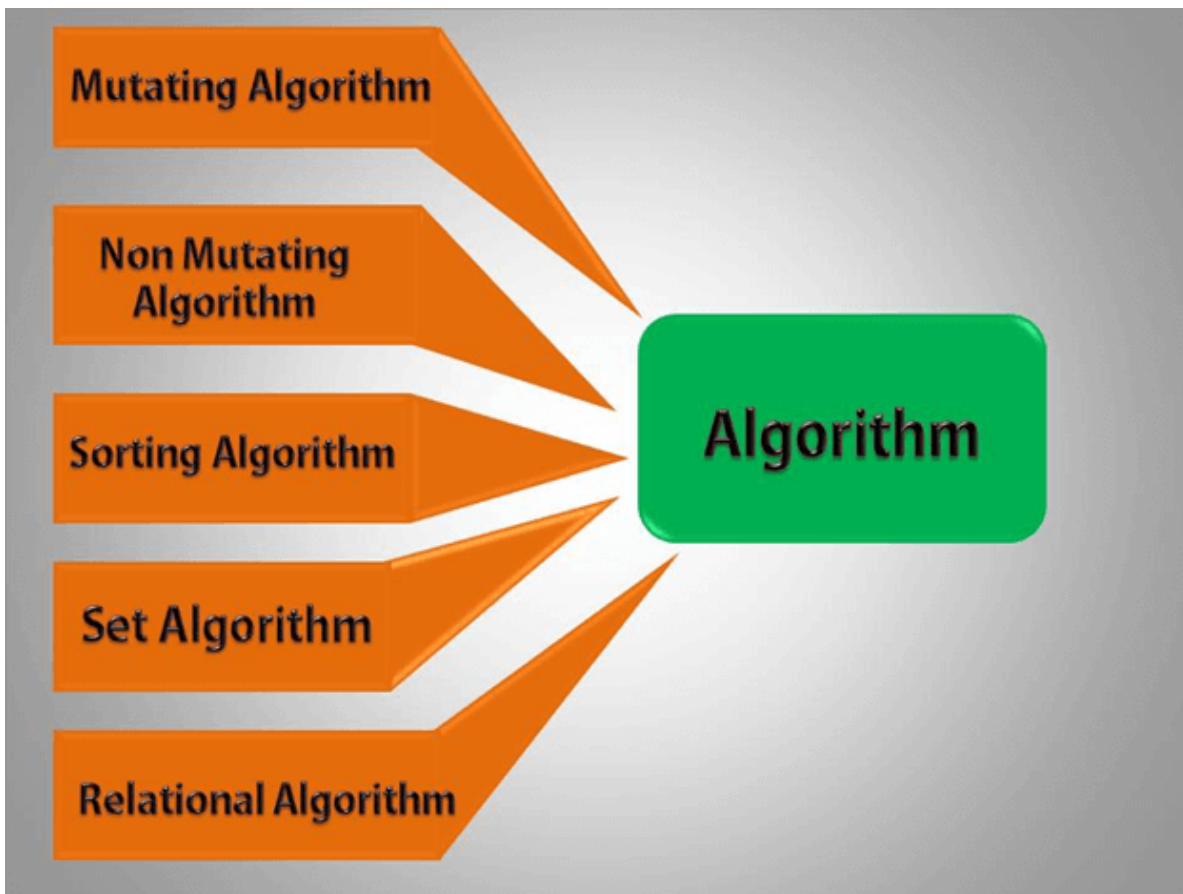
Algorithms

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.

FUNCTION OBJECTS

A Function object is a function wrapped in a class so that it looks like an object. A function object extends the characteristics of a regular function by using the feature of an object oriented such as generic programming. Therefore, we can say that the function object is a smart pointer that has many advantages over the normal function.

Following are the advantages of function objects over a regular function:

- Function objects can have member functions as well as member attributes.
- Function objects can be initialized before their usage.
- Regular functions can have different types only when the signature differs. Function objects can have different types even when the signature is the same.
- Function objects are faster than the regular function.

A function object is also known as a '**functor**'. A function object is an object that contains atleast one definition of **operator()** function. It means that if we declare the object 'd' of a class in which **operator()** function is defined, we can use the object 'd' as a regular function.

Suppose 'd' is an object of a class, operator() function can be called as:

```
d();
```

which is same as:

```
d.operator() ( );
```

Let's see a simple example:

```
#include <iostream>
using namespace std;
class function_object
{
    public:
    int operator()(int a, int b)
    {
        return a+b;
    }
};

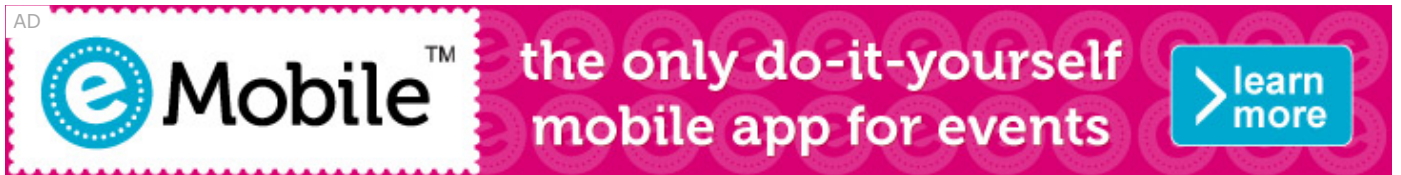
int main()
{
    function_object f;
    int result = f(5,5);
    cout<<"Addition of a and b is : "<<result;
```

```
return 0;  
}
```

Output:

Addition of a and b is : 10

In the above example, 'f' is an object of a function_object class which contains the definition of operator() function. Therefore, 'f' can be used as an ordinary function to call the operator() function.

[← Prev](#)[Next →](#)

AD

 [For Videos Join Our Youtube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

C++ Vector

A vector is a sequence container class that implements dynamic array, means size automatically changes when appending elements. A vector stores the elements in contiguous memory locations and allocates the memory as needed at run time.

Difference between vector and array

An array follows static approach, means its size cannot be changed during run time while vector implements dynamic array means it automatically resizes itself when appending elements.

Syntax

Consider a vector 'v1'. Syntax would be:

```
vector<object_type> v1;
```

Example

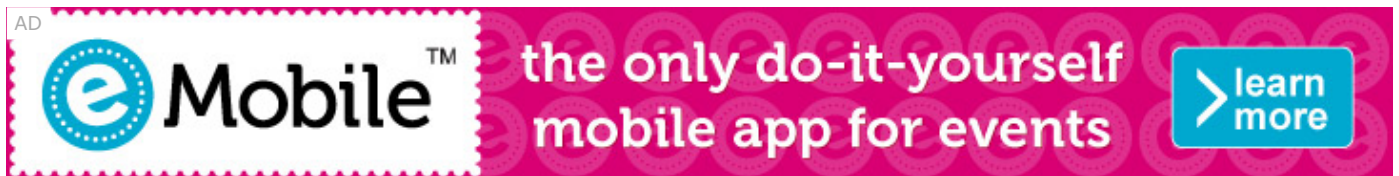
Let's see a simple example.

```
#include<iostream>
#include<vector>
using namespace std;
int main()
{
vector<string> v1;
v1.push_back("javaTpoint ");
v1.push_back("tutorial");
for(vector<string>::iterator itr=v1.begin();itr!=v1.end();++itr)
cout<<*itr;
return 0;
}
```

Output:

```
javaTpoint tutorial
```


In this example, vector class has been used to display the string.



C++ Vector Functions

Function	Description
<code>at()</code>	It provides a reference to an element.
<code>back()</code>	It gives a reference to the last element.
<code>front()</code>	It gives a reference to the first element.
<code>swap()</code>	It exchanges the elements between two vectors.
<code>push_back()</code>	It adds a new element at the end.
<code>pop_back()</code>	It removes a last element from the vector.
<code>empty()</code>	It determines whether the vector is empty or not.
<code>insert()</code>	It inserts new element at the specified position.
<code>erase()</code>	It deletes the specified element.
<code>resize()</code>	It modifies the size of the vector.
<code>clear()</code>	It removes all the elements from the vector.
<code>size()</code>	It determines a number of elements in the vector.
<code>capacity()</code>	It determines the current capacity of the vector.
<code>assign()</code>	It assigns new values to the vector.
<code>operator=()</code>	It assigns new values to the vector container.
<code>operator[]()</code>	It access a specified element.
<code>end()</code>	It refers to the past-lats-element in the vector.
<code>emplace()</code>	It inserts a new element just before the position pos.

<code>emplace_back()</code>	It inserts a new element at the end.
<code>rend()</code>	It points the element preceding the first element of the vector.
<code>rbegin()</code>	It points the last element of the vector.
<code>begin()</code>	It points the first element of the vector.
<code>max_size()</code>	It determines the maximum size that vector can hold.
<code>cend()</code>	It refers to the past-last-element in the vector.
<code>cbegin()</code>	It refers to the first element of the vector.
<code>crbegin()</code>	It refers to the last character of the vector.
<code>crend()</code>	It refers to the element preceding the first element of the vector.
<code>data()</code>	It writes the data of the vector into an array.
<code>shrink_to_fit()</code>	It reduces the capacity and makes it equal to the size of the vector.

[← Prev](#)[Next →](#)

AD

 [For Videos Join Our Youtube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Initialize Vector in C++

A vector can store multiple data values like arrays, but they can only store object references and not primitive data types. They store an object's reference means that they point to the objects that contain the data, instead of storing them. Unlike an array, vectors need not be initialized with size. They have the flexibility to adjust according to the number of object references, which is possible because their storage is handled automatically by the container. The container will keep an internal copy of alloc, which is used to allocate storage for lifetime. Vectors can be located and traversed using iterators, so they are placed in contiguous storage. Vector has safety features also, which saves programs from crashing, unlike Array. We can give reserve space to vector, but not to arrays. An array is not a class, but a vector is a class. In vector, elements can be deleted, but not in arrays.

With the parent 'Collection class,' vector is sent in the form of a template class. The array is the lower level data structure with their specific properties. Vectors have functions & constructors; they are not index-based. They are the opposite of Arrays, which are index-based data structures. Here, the lowest address is provided to the first element, and the highest address is provided to the last element. Vector is used for insertion and deletion of an object, whereas Arrays used for frequent access of objects. Arrays are memory saving data structures, while Vector utilizes much more memory in exchange to manage storage and grow dynamically. Vector takes more time to access the elements, but this not the case with Arrays.

There are four ways of initializing a **vector in C++**:

- By entering the values one-by-one
- By using an overloaded constructor of the vector class
- By the help of arrays
- By using another initialized vector

By entering the values one-by-one -

All the elements in a vector can be inserted one-by-one using the vector class method 'push_back.'

Algorithm

Begin

Declare v of vector type.

Then we call push_back() function. This is done to insert values into vector v.

Then we print "Vector elements: \n".

" **for** (**int** a: v)

print all the elements of variable a."

Code -

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int> vec;
    vec.push_back(1);
    vec.push_back(2);
    vec.push_back(3);
    vec.push_back(4);
    vec.push_back(5);
    vec.push_back(6);
    vec.push_back(7);
    vec.push_back(8);
    vec.push_back(9);
    vec.push_back(101);
    for (int i = 0; i < vec.size(); i++)
    {
        cout << vec[i] << " ";
    }
    return 0;
}
```

Output

The screenshot shows a web browser window with the URL `programiz.com/cpp-programming/online-compiler/`. The page features a header with the Programiz logo and a banner for IBM. The main content area displays a C++ code editor with the following code:

```

1 #include <iostream>
2 #include <vector>
3 using namespace std;
4 int main()
5 {
6     vector<int> vec;
7     vec.push_back(1);
8     vec.push_back(2);
9     vec.push_back(3);
10    vec.push_back(4);
11    vec.push_back(5);
12    vec.push_back(6);
13    vec.push_back(7);
14    vec.push_back(8);
15    vec.push_back(9);
16    vec.push_back(101);
17    for (int i = 0; i < vec.size(); i++)
18    {
19        cout << vec[i] << " ";
20    }
21    return 0;
22 }
23

```

The output window on the right shows the result of the program execution: `1 2 3 4 5 6 7 8 9 101`. A small video player is visible in the bottom right corner of the browser window.

Using an overloaded constructor -

When a vector has multiple elements with the same values, then we use this method.

By using an overloaded constructor of the vector class -

This method is mainly used when a vector is filled with multiple elements with the same value.

Algorithm

Begin

First, we initialize a variable say 's'.

Then we have to create a vector say 'v' with size 's'.

Then we initialize vector v1.

Then initialize v2 by v1.

Then we print the elements.

End.

Code -

```

#include <iostream>
#include <vector>
using namespace std;
int main()

```

```
{  
    int elements = 12;  
    vector<int> vec(elements, 8);  
    for (int i = 0; i < vec.size(); i++)  
    {  
        cout << vec[i] << " \n";  
    }  
    return 0;  
}
```

Output

```
8  
8  
8  
8  
8  
8  
8  
8  
8  
8  
8  
8  
8
```

By the help of arrays -

We pass an array to the constructor of the vector class. The Array contains the elements which will fill the vector.

Algorithm -

Begin

First, we create a vector say v.

Then, we initialize the vector.

In the end, print the elements.

End.

Code -

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int> vectr{9,8,7,6,5,4,3,2,1,0};
    for (int i = 0; i < vectr.size(); i++)
    {
        cout << vectr[i] << " \n";
    }
    return 0;
}
```

Output

```
9
8
7
6
5
4
3
2
1
0
```

Using another initialized vector -

Here, we have to pass the begin() and end() iterators of an initialized vector to a vector class constructor. Then we initialize a new vector and fill it with the old vector.

Algorithm -

Begin

First, we have to create a vector v1.

Then, we have to initialize vector v1 by an array.

Then we initialize vector v2 by v1.

We have to print the elements.

End.

Code -

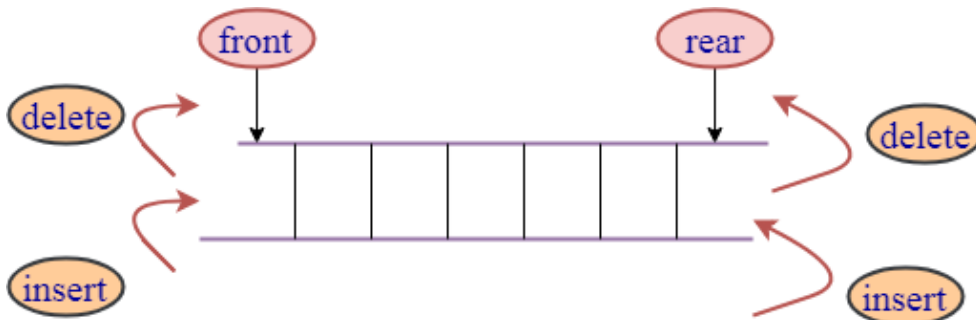
```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int> vec_1{1,2,3,4,5,6,7,8};
    vector<int> vec_2(vec_1.begin(), vec_1.end());
    for (int i = 0; i < vec_2.size(); i++)
    {
        cout << vec_2[i] << " \n";
    }
    return 0;
}
```

Output

```
1
2
3
4
5
6
7
8
```


C++ Deque

Deque stands for double ended queue. It generalizes the queue data structure i.e insertion and deletion can be performed from both the ends either front or back.



Syntax for creating a deque object:

```
deque<object_type> deque_name;
```

C++ Deque Functions

Method	Description
<code>assign()</code>	It assigns new content and replacing the old one.
<code>emplace()</code>	It adds a new element at a specified position.
<code>emplace_back()</code>	It adds a new element at the end.
<code>emplace_front()</code>	It adds a new element in the beginning of a deque.
<code>insert()</code>	It adds a new element just before the specified position.
<code>push_back()</code>	It adds a new element at the end of the container.
<code>push_front()</code>	It adds a new element at the beginning of the container.
<code>pop_back()</code>	It deletes the last element from the deque.
<code>pop_front()</code>	It deletes the first element from the deque.
<code>swap()</code>	It exchanges the contents of two deques.
<code>clear()</code>	It removes all the contents of the deque.

<code>empty()</code>	It checks whether the container is empty or not.
<code>erase()</code>	It removes the elements.
<code>max_size()</code>	It determines the maximum size of the deque.
<code>resize()</code>	It changes the size of the deque.
<code>shrink_to_fit()</code>	It reduces the memory to fit the size of the deque.
<code>size()</code>	It returns the number of elements.
<code>at()</code>	It access the element at position pos.
<code>operator[]()</code>	It access the element at position pos.
<code>operator=()</code>	It assigns new contents to the container.
<code>back()</code>	It access the last element.
<code>begin()</code>	It returns an iterator to the beginning of the deque.
<code>cbegin()</code>	It returns a constant iterator to the beginning of the deque.
<code>end()</code>	It returns an iterator to the end.
<code>cend()</code>	It returns a constant iterator to the end.
<code>rbegin()</code>	It returns a reverse iterator to the beginning.
<code>crbegin()</code>	It returns a constant reverse iterator to the beginning.
<code>rend()</code>	It returns a reverse iterator to the end.
<code>crend()</code>	It returns a constant reverse iterator to the end.
<code>front()</code>	It access the last element.

[← Prev](#)[Next →](#)

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.

Template for list

```
#include<iostream>
#include<list>
using namespace std;
int main()
{
    list<int> l;
}
```

It creates an empty list of integer type values.

List can also be initialised with the parameters.

```
#include<iostream>
#include<list>
using namespace std;
int main()
{
    list<int> l{1,2,3,4};
}
```

List can be initialised in two ways.

```
list<int> new_list{1,2,3,4};  
or  
list<int> new_list = {1,2,3,4};
```

C++ List Functions

Following are the member functions of the list:

Method	Description
<code>insert()</code>	It inserts the new element before the position pointed by the iterator.
<code>push_back()</code>	It adds a new element at the end of the vector.
<code>push_front()</code>	It adds a new element to the front.
<code>pop_back()</code>	It deletes the last element.
<code>pop_front()</code>	It deletes the first element.
<code>empty()</code>	It checks whether the list is empty or not.
<code>size()</code>	It finds the number of elements present in the list.
<code>max_size()</code>	It finds the maximum size of the list.
<code>front()</code>	It returns the first element of the list.
<code>back()</code>	It returns the last element of the list.
<code>swap()</code>	It swaps two list when the type of both the list are same.
<code>reverse()</code>	It reverses the elements of the list.
<code>sort()</code>	It sorts the elements of the list in an increasing order.
<code>merge()</code>	It merges the two sorted list.
<code>splice()</code>	It inserts a new list into the invoking list.
<code>unique()</code>	It removes all the duplicate elements from the list.
<code>resize()</code>	It changes the size of the list container.
<code>assign()</code>	It assigns a new element to the list container.

<code>emplace()</code>	It inserts a new element at a specified position.
<code>emplace_back()</code>	It inserts a new element at the end of the vector.
<code>emplace_front()</code>	It inserts a new element at the beginning of the list.

[Next →](#)

AD

Help Others, Please Share



C++ STL Set

Introduction to set

Sets are part of the **C++ STL (Standard Template Library)**. Sets are the associative containers that stores sorted key, in which each key is unique and it can be inserted or deleted but cannot be altered.

Syntax

```
template < class T,                // set::key_type/value_type
          class Compare = less<T>, // set::key_compare/value_compare
          class Alloc = allocator<T> // set::allocator_type
> class set;
```

Parameter

T: Type of element stored in the container set.

Compare: A comparison class that takes two arguments of the same type bool and returns a value. This argument is optional and the binary predicate less<T>, is the default value.

Alloc: Type of the allocator object which is used to define the storage allocation model.



Member Functions

Below is the list of all member functions of set:

Constructor/Destructor

Functions	Description
(constructor)	Construct set
(destructor)	Set destructor

<code>operator=</code>	Copy elements of the set to another set.
------------------------	--

Iterators

Functions	Description
<code>Begin</code>	Returns an iterator pointing to the first element in the set.
<code>cbegin</code>	Returns a const iterator pointing to the first element in the set.
<code>End</code>	Returns an iterator pointing to the past-end.
<code>Cend</code>	Returns a constant iterator pointing to the past-end.
<code>rbegin</code>	Returns a reverse iterator pointing to the end.
<code>Rend</code>	Returns a reverse iterator pointing to the beginning.
<code>crbegin</code>	Returns a constant reverse iterator pointing to the end.
<code>Crend</code>	Returns a constant reverse iterator pointing to the beginning.

Capacity

Functions	Description
<code>empty</code>	Returns true if set is empty.
<code>Size</code>	Returns the number of elements in the set.
<code>max_size</code>	Returns the maximum size of the set.

Modifiers

Functions	Description
<code>insert</code>	Insert element in the set.
<code>Erase</code>	Erase elements from the set.
<code>Swap</code>	Exchange the content of the set.
<code>Clear</code>	Delete all the elements of the set.

<code>emplace</code>	Construct and insert the new elements into the set.
<code>emplace_hint</code>	Construct and insert new elements into the set by hint.

Observers

Functions	Description
<code>key_comp</code>	Return a copy of key comparison object.
<code>value_comp</code>	Return a copy of value comparison object.

Operations

Functions	Description
<code>Find</code>	Search for an element with given key.
<code>count</code>	Gets the number of elements matching with given key.
<code>lower_bound</code>	Returns an iterator to lower bound.
<code>upper_bound</code>	Returns an iterator to upper bound.
<code>equal_range</code>	Returns the range of elements matches with given key.

Allocator

Functions	Description
<code>get_allocator</code>	Returns an allocator object that is used to construct the set.

Non-Member Overloaded Functions

Functions	Description
<code>operator==</code>	Checks whether the two sets are equal or not.
<code>operator!=</code>	Checks whether the two sets are equal or not.
<code>operator<</code>	Checks whether the first set is less than other or not.

<code>operator<=</code>	Checks whether the first set is less than or equal to other or not.
<code>operator></code>	Checks whether the first set is greater than other or not.
<code>operator>=</code>	Checks whether the first set is greater than equal to other or not.
<code>swap()</code>	Exchanges the element of two sets.

[Next →](#)

AD

 [For Videos Join Our Youtube Channel: Join Now](#)


Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share




Learn Latest Tutorials

 [Splunk tutorial](#)
Splunk

 [SPSS tutorial](#)
SPSS

 [Swagger tutorial](#)

 [T-SQL tutorial](#)
Transact-SQL

C++ stack

In computer science we go for working on a large variety of programs. Each of them has their own domain and utility. Based on the purpose and environment of the program creation, we have a large number of data structures available to choose from. One of them is 'stack'. Before discussing about this data type let us take a look at its syntax.

Syntax

```
template<class T, class Container = deque<T> > class stack;
```

This data structure works on the LIFO technique, where LIFO stands for Last In First Out. The element which was first inserted will be extracted at the end and so on. There is an element called as 'top' which is the element at the upper most position. All the insertion and deletion operations are made at the top element itself in the stack.

Stacks in the application areas are implied as the container adaptors.

The containers should have a support for the following list of operations:

- empty
- size
- back
- push_back
- pop_back

Template Parameters

T: The argument specifies the type of the element which the container adaptor will be holding.

Container: The argument specifies an internal object of container where the elements of the stack are hold.

Member Types

Given below is a list of the stack member types with a short description of the same.

Member Types	Description
value_type	Element type is specified.

container_type	Underlying container type is specified.
size_type	It specifies the size range of the elements.

AD

UBER

TAP A BUTTON. GET A RIDE.
Sign up here for **\$20** off your first trip!

Functions

With the help of functions, an object or variable can be played with in the field of programming. Stacks provide a large number of functions that can be used or embedded in the programs. A list of the same is given below:

Function	Description
(constructor)	The function is used for the construction of a stack container.
empty	The function is used to test for the emptiness of a stack. If the stack is empty the function returns true else false.
size	The function returns the size of the stack container, which is a measure of the number of elements stored in the stack.
top	The function is used to access the top element of the stack. The element plays a very important role as all the insertion and deletion operations are performed at the top element.
push	The function is used for the insertion of a new element at the top of the stack.
pop	The function is used for the deletion of element, the element in the stack is deleted from the top.
emplace	The function is used for insertion of new elements in the stack above the current top element.
swap	The function is used for interchanging the contents of two containers in reference.
relational operators	The non member function specifies the relational operators that are needed for the stacks.

uses
allocator<stack>

As the name suggests the non member function uses the allocator for the stacks.

Example: A simple program to show the use of basic stack functions.

```
#include <iostream>
#include <stack>
using namespace std;
void newstack(stack <int> ss)
{
    stack <int> sg = ss;
    while (!sg.empty())
    {
        cout << '\t' << sg.top();
        sg.pop();
    }
    cout << '\n';
}
int main ()
{
    stack <int> newst;
    newst.push(55);
    newst.push(44);
    newst.push(33);
    newst.push(22);
    newst.push(11);

    cout << "The stack newst is : ";
    newstack(newst);
    cout << "\n newst.size() : " << newst.size();
    cout << "\n newst.top() : " << newst.top();
    cout << "\n newst.pop() : ";
    newst.pop();
    newstack(newst);
    return 0;
}
```

Output:

```
The stack newst is :    11    22    33    44    55

newst.size() : 5
newst.top() : 11
newst.pop() : 22    33    44    55
```

[← Prev](#)[Next →](#)

AD

 [For Videos Join Our Youtube Channel: Join Now](#)


Feedback


- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share




Learn Latest Tutorials

 [Splunk tutorial](#)
Splunk

 [SPSS tutorial](#)
SPSS

 [Swagger tutorial](#)

 [T-SQL tutorial](#)
Transact-SQL

C++ queue

In computer science we go for working on a large variety of programs. Each of them has their own domain and utility. Based on the purpose and environment of the program creation, we have a large number of data structures available to choose from. One of them is 'queues. Before discussing about this data type let us take a look at its syntax.

Syntax

```
template<class T, class Container = deque<T> > class queue;
```

This data structure works on the FIFO technique, where FIFO stands for First In First Out. The element which was first inserted will be extracted at the first and so on. There is an element called as 'front' which is the element at the front most position or say the first position, also there is an element called as 'rear' which is the element at the last position. In normal queues insertion of elements take at the rear end and the deletion is done from the front.

Queues in the application areas are implied as the container adaptors.

The containers should have a support for the following list of operations:

- empty
- size
- push_back
- pop_front
- front
- back

Template Parameters

T: The argument specifies the type of the element which the container adaptor will be holding.

Container: The argument specifies an internal object of container where the elements of the queues are held.

Member Types

Given below is a list of the queue member types with a short description of the same.

Member Types	Description
value_type	Element type is specified.
container_type	Underlying container type is specified.
size_type	It specifies the size range of the elements.
reference	It is a reference type of a container.
const_reference	It is a reference type of a constant container.

AD

**Cruit**

a new way to hire talent

Functions

With the help of functions, an object or variable can be played with in the field of programming. Queues provide a large number of functions that can be used or embedded in the programs. A list of the same is given below:

Function	Description
(constructor)	The function is used for the construction of a queue container.
empty	The function is used to test for the emptiness of a queue. If the queue is empty the function returns true else false.
size	The function returns the size of the queue container, which is a measure of the number of elements stored in the queue.
front	The function is used to access the front element of the queue. The element plays a very important role as all the deletion operations are performed at the front element.
back	The function is used to access the rear element of the queue. The element plays a very important role as all the insertion operations are performed at the rear element.
push	The function is used for the insertion of a new element at the rear end of the queue.

pop	The function is used for the deletion of element; the element in the queue is deleted from the front end.
emplace	The function is used for insertion of new elements in the queue above the current rear element.
swap	The function is used for interchanging the contents of two containers in reference.
relational operators	The non member function specifies the relational operators that are needed for the queues.
uses allocator<queue>	As the name suggests the non member function uses the allocator for the queues.

Example: A simple program to show the use of basic queue functions.

```
#include <iostream>
#include <queue>
using namespace std;
void showsg(queue <int> sg)
{
    queue <int> ss = sg;
    while (!ss.empty())
    {
        cout << '\t' << ss.front();
        ss.pop();
    }
    cout << '\n';
}

int main()
{
    queue <int> fquiz;
    fquiz.push(10);
    fquiz.push(20);
    fquiz.push(30);

    cout << "The queue fquiz is : ";
```



```
showsg(fquiz);

cout << "\nquiz.size() : " << fquiz.size();
cout << "\nquiz.front() : " << fquiz.front();
cout << "\nquiz.back() : " << fquiz.back();

cout << "\nquiz.pop() : ";
fquiz.pop();
showsg(fquiz);

return 0;
}
```

Output:

```
The queue fquiz is :    10    20    30

quiz.size() : 3
quiz.front() : 10
quiz.back() : 30
quiz.pop() :    20    30
```

[< Prev](#)[Next >](#)

AD

Priority Queue in C++

The priority queue in C++ is a derived container in STL that considers only the highest priority element. The queue follows the FIFO policy while priority queue pops the elements based on the priority, i.e., the highest priority element is popped first.

It is similar to the ordinary queue in certain aspects but differs in the following ways:

- In a priority queue, every element in the queue is associated with some priority, but priority does not exist in a queue data structure.
- The element with the highest priority in a priority queue will be removed first while queue follows the **FIFO(First-In-First-Out)** policy means the element which is inserted first will be deleted first.
- If more than one element exists with the same priority, then the order of the element in a queue will be taken into consideration.

Note: The priority queue is the extended version of a normal queue except that the element with the highest priority will be removed first from the priority queue.

Syntax of Priority Queue

```
priority_queue<int> variable_name;
```

Let's understand the priority queue through a simple example.

Operation	Priority Queue	Return Value
Push(1)	<div><div>1</div><div></div><div></div><div></div><div></div></div>	
Push(4)	<div><div>1</div><div>4</div><div></div><div></div><div></div></div>	
Push(2)	<div><div>1</div><div>4</div><div>2</div><div></div><div></div></div>	
Pop()	<div><div>1</div><div>2</div><div></div><div></div><div></div></div>	4
Push(3)	<div><div>1</div><div>2</div><div>3</div><div></div><div></div></div>	
Pop()	<div><div>1</div><div>4</div><div>2</div><div></div><div></div></div>	3

In the above illustration, we have inserted the elements by using a push() function, and the insert operation is identical to the normal queue. But when we delete the element from the queue by using a pop() function, then the element with the highest priority will be deleted first.

Member Function of Priority Queue

Function	Description
push()	It inserts a new element in a priority queue.
pop()	It removes the top element from the queue, which has the highest priority.
top()	This function is used to address the topmost element of a priority queue.
size()	It determines the size of a priority queue.
empty()	It verifies whether the queue is empty or not. Based on the verification, it returns the status.
swap()	It swaps the elements of a priority queue with another queue having the same type and size.
emplace()	It inserts a new element at the top of the priority queue.

Let's create a simple program of priority queue.

```

#include <iostream>
#include<queue>
using namespace std;
int main()
{
    priority_queue<int> p; // variable declaration.
    p.push(10); // inserting 10 in a queue, top=10
    p.push(30); // inserting 30 in a queue, top=30
    p.push(20); // inserting 20 in a queue, top=20
    cout<<"Number of elements available in 'p' : "<<p.size()<<endl;
    while(!p.empty())
    {
        std::cout << p.top() << std::endl;
        p.pop();
    }
    return 0;
}

```

In the above code, we have created a priority queue in which we insert three elements, i.e., 10, 30, 20. After inserting the elements, we display all the elements of a priority queue by using a while loop.

Output

```

Number of elements available in 'p' :3
30
20
10 zzzzz/

```

Let's see another example of a priority queue.

```

#include <iostream>
#include<queue>
using namespace std;
int main()
{
    priority_queue<int> p; // priority queue declaration
    priority_queue<int> q; // priority queue declaration
    p.push(1); // inserting element '1' in p.
    p.push(2); // inserting element '2' in p.
    p.push(3); // inserting element '3' in p.
    p.push(4); // inserting element '4' in p.
    q.push(5); // inserting element '5' in q.
    q.push(6); // inserting element '6' in q.
    q.push(7); // inserting element '7' in q.
    q.push(8); // inserting element '8' in q.
    p.swap(q);
    std::cout << "Elements of p are : " << std::endl;
    while(!p.empty())
    {
        std::cout << p.top() << std::endl;
        p.pop();
    }
    std::cout << "Elements of q are : " << std::endl;
}

```

```
while(!q.empty())
{
    std::cout << q.top() << std::endl;
    q.pop();
}
return 0;
}
```

In the above code, we have declared two priority queues, i.e., p and q. We inserted four elements in 'p' priority queue and four in 'q' priority queue. After inserting the elements, we swap the elements of 'p' queue with 'q' queue by using a swap() function.

Output

```
Elements of p are :
8
7
6
5
Elements of q are :
4
3
2
1
```

[< Prev](#)[Next >](#)

AD

 [For Videos Join Our Youtube Channel: Join Now](#)


Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share





Learn Latest Tutorials

 [Splunk tutorial](#)
Splunk

 [SPSS tutorial](#)
SPSS

 [Swagger tutorial](#)

 [T-SQL tutorial](#)
Transact-SQL

 [Tumblr tutorial](#)
Tumblr

 [React tutorial](#)
ReactJS

C++ map function

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.

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

Syntax

```
template < class Key,                // map::key_type
          class T,                  // map::mapped_type
          class Compare = less<Key>, // map::key_compare
          class Alloc = allocator<pair<const Key,T> > // map::allocator_type
> class map;
```

Parameter

key: The key data type to be stored in the map.

type: The data type of value to be stored in the map.

compare: A comparison class that takes two arguments of the same type bool and returns a value. This argument is optional and the binary predicate less<"key"> is the default value.

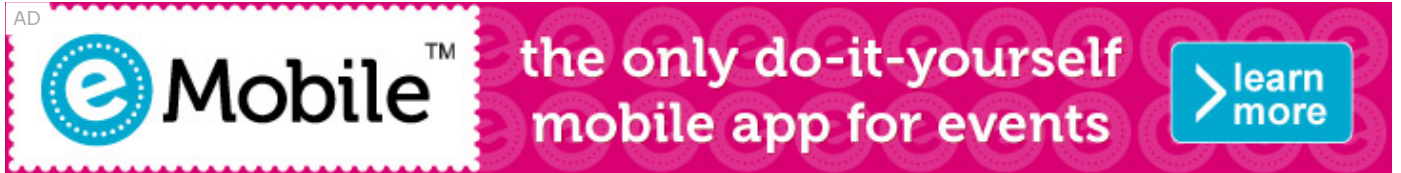
alloc: Type of the allocator object. This argument is optional and the default value is allocator .

Creating a map

Maps can easily be created using the following statement:

```
typedef pair<const Key, T> value_type;
```

The above form will use to create a map with key of type **Key type** and value of type **value type**. One important thing 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.



Example 1

```
#include <string.h>
#include <iostream>
#include <map>
#include <utility>
using namespace std;
int main()
{
    map<int, string> Employees;
    // 1) Assignment using array index notation
    Employees[101] = "Nikita";
    Employees[105] = "John";
    Employees[103] = "Dolly";
    Employees[104] = "Deep";
    Employees[102] = "Aman";
    cout << "Employees[104]=" << Employees[104] << endl << endl;
    cout << "Map size: " << Employees.size() << endl;
    cout << endl << "Natural Order:" << endl;
    for( map<int,string>::iterator ii=Employees.begin(); ii!=Employees.end(); ++ii)
    {
        cout << (*ii).first << ": " << (*ii).second << endl;
    }
    cout << endl << "Reverse Order:" << endl;
    for( map<int,string>::reverse_iterator ii=Employees.rbegin(); ii!=Employees.rend(); ++ii)
    {
        cout << (*ii).first << ": " << (*ii).second << endl;
    }
}
```

```
}  
}
```

Output:

```
Employees[104]=Deep
```

```
Map size: 5
```

```
Natural Order:
```

```
101: Nikita
```

```
102: Aman
```

```
103: Dolly
```

```
104: Deep
```

```
105: John
```

```
Reverse Order:
```

```
105: John
```

```
104: Deep
```

```
103: Dolly
```

```
102: Aman
```

```
101: Nikita
```

Member Functions

Below is the list of all member functions of map:

Constructor/Destructor

Functions	Description
constructors	Construct map
destructors	Map destructor
operator=	Copy elements of the map to another map.

Iterators

Functions	Description
<code>begin</code>	Returns an iterator pointing to the first element in the map.
<code>cbegin</code>	Returns a const iterator pointing to the first element in the map.
<code>end</code>	Returns an iterator pointing to the past-end.
<code>cend</code>	Returns a constant iterator pointing to the past-end.
<code>rbegin</code>	Returns a reverse iterator pointing to the end.
<code>rend</code>	Returns a reverse iterator pointing to the beginning.
<code>crbegin</code>	Returns a constant reverse iterator pointing to the end.
<code>crend</code>	Returns a constant reverse iterator pointing to the beginning.

Capacity

Functions	Description
<code>empty</code>	Returns true if map is empty.
<code>size</code>	Returns the number of elements in the map.
<code>max_size</code>	Returns the maximum size of the map.

Element Access

Functions	Description
<code>operator[]</code>	Retrieve the element with given key.
<code>at</code>	Retrieve the element with given key.

Modifiers

Functions	Description
<code>insert</code>	Insert element in the map.

<code>erase</code>	Erase elements from the map.
<code>swap</code>	Exchange the content of the map.
<code>clear</code>	Delete all the elements of the map.
<code>emplace</code>	Construct and insert the new elements into the map.
<code>emplace_hint</code>	Construct and insert new elements into the map by hint.

Observers

Functions	Description
<code>key_comp</code>	Return a copy of key comparison object.
<code>value_comp</code>	Return a copy of value comparison object.

Operations

Functions	Description
<code>find</code>	Search for an element with given key.
<code>count</code>	Gets the number of elements matching with given key.
<code>lower_bound</code>	Returns an iterator to lower bound.
<code>upper_bound</code>	Returns an iterator to upper bound.
<code>equal_range</code>	Returns the range of elements matches with given key.

Allocator

Functions	Description
<code>get_allocator</code>	Returns an allocator object that is used to construct the map.

Non-Member Overloaded Functions

Functions	Description
-----------	-------------

operator==	Checks whether the two maps are equal or not.
operator!=	Checks whether the two maps are equal or not.
operator<	Checks whether the first map is less than other or not.
operator<=	Checks whether the first map is less than or equal to other or not.
operator>	Checks whether the first map is greater than other or not.
operator>=	Checks whether the first map is greater than equal to other or not.
swap()	Exchanges the element of two maps.


[< Prev](#)[Next >](#)

AD


Help Others, Please Share



AD

 **eMobile™**

the only do-it-yourself
mobile app for events

 learn
more

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

Multimap employee has duplicate keys age.

Syntax

```
template < class Key,                // multimap::key_type
          class T,                  // multimap::mapped_type
          class Compare = less<Key>, // multimap::key_compare
          class Alloc = allocator<pair<const Key,T> > // multimap::allocator_type
> class multimap;
```

Parameter

key: The key data type to be stored in the multimap.

type: The data type of value to be stored in the multimap.

compare: A comparison class that takes two arguments of the same type bool and returns a value. This argument is optional and the binary predicate less<"key"> is the default value.

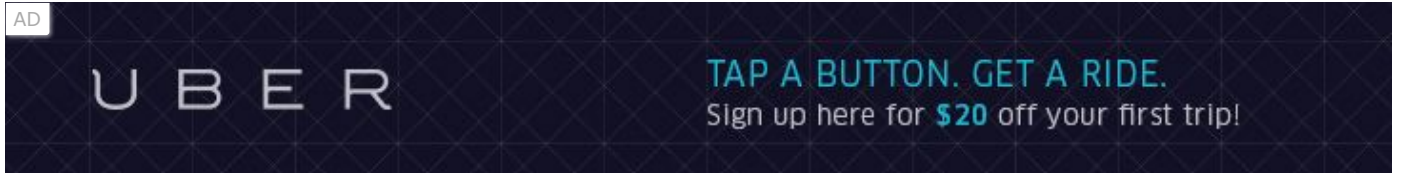
alloc: Type of the allocator object. This argument is optional and the default value is allocator .

Creating a multimap

Multimaps can easily be created using the following statement:

```
typedef pair<const Key, T> value_type;
```

The above form will use to create a multimap with key of type **Key_type** and value of type **value_type**. One important thing is that key of a multimap and corresponding values are always inserted as a pair, you cannot insert only key or just a value in a multimap.



Example

```
#include <iostream>
#include <map>
#include <string>

using namespace std;

int main()
{
    multimap<string, string> m = {
        {"India", "New Delhi"},
        {"India", "Hyderabad"},
        {"United Kingdom", "London"},
        {"United States", "Washington D.C"}
    };

    cout << "Size of map m: " << m.size() << endl;
    cout << "Elements in m: " << endl;

    for (multimap<string, string>::iterator it = m.begin(); it != m.end(); ++it)
    {
        cout << " [" << (*it).first << ", " << (*it).second << "]" << endl;
    }
}
```

```
    return 0;  
}
```

Output:

```
Size of map m: 4  
Elements in m:  
    [India, New Delhi]  
    [India, Hyderabad]  
    [United Kingdom, London]  
    [United States, Washington D.C]
```

Member Functions

Below is the list of all member functions of multimap:

Constructor/Destructor

Functions	Description
constructor	Construct multimap
destructor	Multimap destructor
operator=	Copy elements of the multimap to another multimap.

Iterators

Functions	Description
<code>begin</code>	Returns an iterator pointing to the first element in the multimap.
<code>cbegin</code>	Returns a <code>const_iterator</code> pointing to the first element in the multimap.
<code>end</code>	Returns an iterator pointing to the past-end.
<code>cend</code>	Returns a constant iterator pointing to the past-end.
<code>rbegin</code>	Returns a reverse iterator pointing to the end.

<code>rend</code>	Returns a reverse iterator pointing to the beginning.
<code>crbegin</code>	Returns a constant reverse iterator pointing to the end.
<code>crend</code>	Returns a constant reverse iterator pointing to the beginning.

Capacity

Functions	Description
<code>empty</code>	Return true if multimap is empty.
<code>size</code>	Returns the number of elements in the multimap.
<code>max_size</code>	Returns the maximum size of the multimap.

Modifiers

Functions	Description
<code>insert</code>	Insert element in the multimap.
<code>erase</code>	Erase elements from the multimap.
<code>swap</code>	Exchange the content of the multimap.
<code>clear</code>	Delete all the elements of the multimap.
<code>emplace</code>	Construct and insert the new elements into the multimap.
<code>emplace_hint</code>	Construct and insert new elements into the multimap by hint.

Observers

Functions	Description
<code>key_comp</code>	Return a copy of key comparison object.
<code>value_comp</code>	Return a copy of value comparison object.

Operations

Functions	Description
find	Search for an element with given key.
count	Gets the number of elements matching with given key.
lower_bound	Returns an iterator to lower bound.
upper_bound	Returns an iterator to upper bound.
equal_range()	Returns the range of elements matches with given key.

Allocator

Functions	Description
get_allocator	Returns an allocator object that is used to construct the multimap.

Non-Member Overloaded Functions

Functions	Description
operator==	Checks whether the two multimaps are equal or not.
operator!=	Checks whether the two multimaps are equal or not.
operator<	Checks whether the first multimap is less than other or not.
operator<=	Checks whether the first multimap is less than or equal to other or not.
operator>	Checks whether the first multimap is greater than other or not.
operator>=	Checks whether the first multimap is greater than equal to other or not.
swap()	Exchanges the element of two multimaps.

[← Prev](#)[Next →](#)

C++ Algorithm Functions

The library defines a large number of functions that are specially suited to be used on a large number of elements at a time or say a range. Now let's straightway take a look at these functions.

Non-modifying sequence operations:

Function	Description
<code>all_of</code>	The following function tests a condition to all the elements of the range.
<code>any_of</code>	The following function tests a condition to some or any of the elements of the range
<code>none_of</code>	The following function checks if none of the elements follow the condition or not.
<code>for_each</code>	The function applies an operation to all the elements of the range.
<code>find</code>	The function finds a value in the range.
<code>find_if</code>	The function finds for an element in the range.
<code>find_if_not</code>	The function finds an element in the range but in the opposite way as the above one.
<code>find_end</code>	The function is used to return the last element of the range.
<code>find_first_of</code>	The function finds for the element that satisfies a condition and occurs at the first.
<code>adjacent_find</code>	The function makes a search for finding the equal and adjacent elements in a range.
<code>count</code>	The function returns the count of a value in the range.
<code>count_if</code>	The function returns the count of values that satisfies a condition.
<code>mismatch</code>	The function returns the value in sequence which is the first mismatch.
<code>equal</code>	The function is used to check if the two ranges have all elements equal.
<code>is_permutation</code>	The function checks whether the range in reference is a permutation of some other range.

<code>search</code>	The function searches for the subsequence in a range.
<code>search_n</code>	The function searches the range for the occurrence of an element.

Modifying sequence operations

Function	Description
<code>copy</code>	The function copies the range of elements.
<code>copy_n</code>	The function copies n elements of the range
<code>copy_if</code>	The function copies the elements of the range if a certain condition is fulfilled.
<code>copy_backward</code>	The function copies the elements in a backward order
<code>move</code>	The function moves the ranges of elements.
<code>move_backward</code>	The function moves the range of elements in the backward order
<code>swap</code>	The function swaps the value of two objects.
<code>swap_ranges</code>	The function swaps the value of two ranges.
<code>iter_swap</code>	The function swaps the values of two iterators under reference.
<code>transform</code>	The function transforms all the values in a range.
<code>replace</code>	The function replaces the values in the range with a specific value.
<code>replace_if</code>	The function replaces the value of the range if a certain condition is fulfilled.
<code>replace_copy</code>	The function copies the range of values by replacing with an element.
<code>replace_copy_if</code>	The function copies the range of values by replacing with an element if a certain condition is fulfilled.
<code>fill</code>	The function fills the values in the range with a value.
<code>fill_n</code>	The function fills the values in the sequence.
<code>generate</code>	The function is used for the generation of values of the range.
<code>generate_n</code>	The function is used for the generation of values of the sequence.
<code>remove</code>	The function removes the values from the range.

<code>remove_if</code>	The function removes the values of the range if a condition is fulfilled.
<code>remove_copy</code>	The function copies the values of the range by removing them.
<code>remove_copy_if</code>	The function copies the values of the range by removing them if a condition is fulfilled.
<code>unique</code>	The function identifies the unique element of the range.
<code>unique_copy</code>	The function copies the unique elements of the range.
<code>reverse</code>	The function reverses the range.
<code>reverse_copy</code>	The function copies the range by reversing values.
<code>rotate</code>	The function rotates the elements of the range in left direction.
<code>rotate_copy</code>	The function copies the elements of the range which is rotated left.
<code>random_shuffle</code>	The function shuffles the range randomly.
<code>shuffle</code>	The function shuffles the range randomly with the help of a generator.

Partitions

Function	Description
<code>is_partitioned</code>	The function is used to deduce whether the range is partitioned or not.
<code>partition</code>	The function is used to partition the range.
<code>stable_partition</code>	The function partitions the range in two stable halves.
<code>partition_copy</code>	The function copies the range after partition.
<code>partition_point</code>	The function returns the partition point for a range.

AD



VIDEO STREAMING FOR THE AWAKENED MIND

GET STARTED NOW

Sorting

Function	Description
----------	-------------

<code>sort</code>	The function sorts all the elements in a range.
<code>stable_sort</code>	The function sorts the elements in the range maintaining the relative equivalent order.
<code>partial_sort</code>	The function partially sorts the elements of the range.
<code>partial_sort_copy</code>	The function copies the elements of the range after sorting it.
<code>is_sorted</code>	The function checks whether the range is sorted or not.
<code>is_sorted_until</code>	The function checks till which element a range is sorted.
<code>nth_element</code>	The functions sorts the elements in the range.

Binary search

Function	Description
<code>lower_bound</code>	Returns the lower bound element of the range.
<code>upper_bound</code>	Returns the upper bound element of the range.
<code>equal_range</code>	The function returns the subrange for the equal elements.
<code>binary_search</code>	The function tests if the values in the range exists in a sorted sequence or not.

Merge

Function	Description
<code>merge</code>	The function merges two ranges that are in a sorted order.
<code>inplace_merge</code>	The function merges two consecutive ranges that are sorted.
<code>includes</code>	The function searches whether the sorted range includes another range or not.
<code>set_union</code>	The function returns the union of two ranges that is sorted.
<code>set_intersection</code>	The function returns the intersection of two ranges that is sorted.
<code>set_difference</code>	The function returns the difference of two ranges that is sorted.

set_symmetric_difference	The function returns the symmetric difference of two ranges that is sorted.
--------------------------	---

Heap

Function	Description
push_heap	The function pushes new elements in the heap.
pop_heap	The function pops new elements in the heap.
make_heap	The function is used for the creation of a heap.
sort_heap	The function sorts the heap.
is_heap	The function checks whether the range is a heap.
is_heap_until	The function checks till which position a range is a heap.

Min/Max

Function	Description
min	Returns the smallest element of the range.
max	Returns the largest element of the range.
minmax	Returns the smallest and largest element of the range.
min_element	Returns the smallest element of the range.
max_element	Returns the largest element of the range.
minmax_element	Returns the smallest and largest element of the range.

Other functions

Function	Description
lexicographical_comapre	The function performs the lexicographical less-than comparison.
next_permutation	The function is used for the transformation of range into the next permutation.

perv_permutation

The function is used for the transformation of range into the previous permutation.

Next →

AD

 [For Videos Join Our Youtube Channel: Join Now](#)


Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share




Learn Latest Tutorials

 [Splunk tutorial](#)
Splunk

 [SPSS tutorial](#)
SPSS

 [Swagger tutorial](#)
Swagger

 [T-SQL tutorial](#)
Transact-SQL

 [Tumblr tutorial](#)

 [React tutorial](#)

 [Regex tutorial](#)