C++ Library - <map>

Introduction to map

Map is dictionary like data structure. It is a sequence of (key, value) pair, where only single value is associated with each unique key. It is often referred as `associative array`.

In map key values generally used to sort the elements. For map data type of key and value can differ and it is represented as

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

Maps are typically implemented as Binary Search Tree.

Zero sized maps are also valid. In that case map.begin() and map.end() points to same location.

Definition

Below is definition of std::map from <map> header file

```
template < class Key,
           class T,
           class Compare = less<Key>,
           class Alloc = allocator<pair<const Key,T> >
           > class map;
```

Parameters

- **Key** − Type of the key.
- **T** − Type of the mapped values.
- **Compare** − A binary predicate that takes two element keys as arguments and returns a bool.
- **Alloc** − Type of the allocator object.

T may be substituted by any other data type including user-defined type.

Member types

Following member types can be used as parameters or return type by member functions.

S.N.	Member types	Definition
1	key_type	Key (First parameter of the template)
2	mapped_type	T (Second parameter of the template)
3	key_compare	Compare (Third parameter of the template)
4	allocator_type	Alloc (Fourth parameter of the template)
5	value_type	pair<const key_type,mapped_type>
6	value_compare	Nested function class to compare elements
7	reference	allocator_type::reference
8	const_reference	allocator_type::const_reference
9	pointer	allocator_type::pointer
10	const_pointer	allocator_type::const_pointer
11	iterator	bi-directional iterator to `value_type`
12	const_iterator	bi-directional iterator to const `value_type`
13	reverse_iterator	reverse iterator
14	const_reverse_iterator	constant reverse iterator
15	difference_type	ptrdiff_t
16	size_type	size_t

Functions from <map>

Below is list of all methods from <map> header.

Constructors

S.N.	Method & Description

1	**map::map** `default constructor` Constructs an empty map with zero elements.
2	**map::map** `range constructor` Constructs a map with as many elements as in range of `first` to `last`.
3	**map::map** `copy constructor` Constructs a map with copy of each elements present in existing map.
4	**map::map** `move constructor` Constructs a map with the contents of other using move semantics.
5	**map::map** `initializer list constructor` Constructs a map from initialize list.

Destructor

S.N.	Method & Description
1	**map::~map** Destroys map object by deallocating it's memory.

Member functions

S.N.	Method & Description
1	**map::at** Returns a reference to the mapped value associated with key `k`.
2	**map::begin** Returns a iterator which refers to the first element of the map.
3	**map::cbegin** Returns a constant iterator which refers to the first element of the map.
4	**map::cend** Returns a constant iterator which points to `past-the-end` element of the map.
5	**map::clear** Destroys the map by removing all elements and sets size of map to zero.
6	**map::count** Returns number of mapped values associated with key `k`.
7	**map::crbegin** Returns a constant reverse iterator which points to the last element of the container i.
8	**map::crend** Returns a constant reverse iterator which points to the theoretical element preceding the first element in the container i.
9	**map::emplace** Extends container by inserting new element.
10	**map::emplace_hint** `hint version` Inserts a new element in a map using hint as a position for element.
11	**map::empty** Tests whether map is empty or not.
12	**map::end** Returns an iterator which points to `past-the-end` element in the map.
13	**map::equal_range** Returns range of elements that matches specific key.
14	**map::erase** `position version` Removes single element of the map from `position`.
15	**map::erase** `position version` Removes single element of the map from `position`.
16	**map::erase** `key` Removes mapped value associated with key `k`.
17	**map::erase** `range version` Removes range of element from the the map.

18	**map::erase** `range version` Removes range of element from the the map.
19	**map::find** Finds an element associated with key `k`.
20	**map::get_allocator** Returns an allocator associated with map.
21	**map::insert** `single element` Extends container by inserting new element in map.
22	**map::insert** `hint version` Extends container by inserting new element in map.
23	**map::insert** `range version` Extends container by inserting new elements in the map.
24	**map::insert** `move hint verstion` Extends map by inserting new element.
25	**map::insert** `initializer list version` Extends map by inserting new element from initializer list.
26	**map::key_comp** Returns a function object that compares the keys, which is a copy of this container's constructor argument `comp`.
27	**map::lower_bound** Returns an iterator pointing to the first element which is not less than key `k`.
28	**map::max_size** Returns the maximum number of elements can be held by map.
29	**map::operator=** `copy version` Assign new contents to the map by replacing old ones and modifies size if necessary.
30	**map::operator=** `move version` Move the contents of one map into another and modifies size if necessary.
31	**map::operator=** `initializer list version` Copy elements from initializer list to map.
32	**map::operator[]** `copy version` If key `k` matches an element in the container, then method returns a reference to the element.
33	**map::operator[]** `move version` If key `k` matches an element in the container, then method returns a reference to the element.
34	**map::rbegin** Returns a reverse iterator which points to the last element of the map.
35	**map::rend** Returns a reverse iterator which points to the reverse end of the map i.
36	**map::size** Returns the number of elements present in the map.
37	**map::swap** Exchanges the content of map with contents of map `x`.
38	**map::upper_bound** Returns an iterator pointing to the first element which is greater than key `k`.
39	**map::value_comp** Returns a function object that compares objects of type `std::map::value_type`.

Non-member overloaded functions

S.N.	Method & Description
1	**operator==** Tests whether two maps are equal or not.
2	**operator!=**

	Tests whether two maps are equal or not.
3	**operator<** Tests whether first map is less than other or not.
4	**map::operator<=** Tests whether first map is less than or equal to other or not.
5	**operator>** Tests whether first map is greater than other or not.
6	**operator>=** Tests whether first map is greater than or equal to other or not.
7	**swap()** Exchanges the content of map x.

Introduction to multimap

Multimap is dictionary like data structure. It is a sequence of `(key, value)` pair, where multiple values can be associate with equivalent keys. It is often referred as `associative array`.

In multimap key values generally used to sort the elements. For multimap data type of key and value can differ and it is represented as

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

Multimaps are typically implemented as Binary Search Tree.

Zero sized multimaps are also valid. In that case multimap.begin() and multimap.end() points to same location.

Definition

Below is definition of std::multimap from <multimap> header file

```
template < class Key,
           class T,
           class Compare = less<Key>,
           class Alloc = allocator<pair<const Key,T> >
           > class multimap;
```

Parameters

- **Key** − Type of the key.
- **T** − Type of the mapped values.
- **Compare** − A binary predicate that takes two element keys as arguments and returns a bool.
- **Alloc** − Type of the allocator object.
- T may be substituted by any other data type including user-defined type.

Member types

Following member types can be used as parameters or return type by member functions.

S.N.	Member types	Definition
1	key_type	Key (First parameter of the template)
2	mapped_type	T (Second parameter of the template)
3	key_compare	Compare (Third parameter of the template)
4	allocator_type	Alloc (Fourth parameter of the template)
5	value_type	pair<const key_type,mapped_type>
6	value_compare	Nested function class to compare elements
7	reference	allocator_type::reference
8	const_reference	allocator_type::const_reference
9	pointer	allocator_type::pointer
10	const_pointer	allocator_type::const_pointer
11	iterator	bi-directional iterator to `value_type`
12	const_iterator	bi-directional iterator to const `value_type`

13	reverse_iterator	reverse iterator
14	const_reverse_iterator	constant reverse iterator
15	difference_type	ptrdiff_t
16	size_type	size_t

Functions from <multimap>

Below is list of all methods from <multimap> header.

Constructors

S.N.	Method & Description
1	**multimap::multimap** `default constructor` Constructs an empty multimap with zero elements.
2	**multimap::multimap** `range constructor` Constructs a multimap with as many elements as in range of `first` to `last`.
3	**multimap::multimap** `copy constructor` Constructs a multimap with copy of each elements present in existing multimap.
4	**multimap::multimap** `move constructor` Constructs a multimap with the contents of other using move semantics.
5	**multimap::multimap** `initializer list constructor` Constructs a multimap from initialize list.

Destructor

S.N.	Method & Description
1	**multimap::~multimap** Destroys multimap object by deallocating it's memory.

Member functions

S.N.	Method & Description
1	**multimap::begin** Returns a iterator which refers to the first element of the multimap.
2	**multimap::cbegin** Returns a constant iterator which refers to the first element of the multimap.
3	**multimap::cend** Returns a constant iterator which points to `past-the-end` element of the multimap.
4	**multimap::clear** Destroys the multimap by removing all elements and sets size of multimap to zero.
5	**multimap::count** Returns number of multimapped values associated with key `k`.
6	**multimap::crbegin** Returns a constant reverse iterator which points to the last element of the container.
7	**multimap::crend** Returns a constant reverse iterator which points to the theoretical element preceding the first element in the container.
8	**multimap::emplace** Extends container by inserting new element.
9	**multimap::emplace_hint** `hint version` Inserts a new element in a multimap using hint as a position for element.
10	**multimap::empty** Tests whether multimap is empty or not.
11	**multimap::end** Returns an iterator which points to `past-the-end` element in the multimap.
12	**multimap::equal_range** Returns range of elements that matches specific key.

13	**multimap::erase** `position version` Removes single element of the multimap from `position`.
14	**multimap::erase** `position version` Removes single element of the multimap from `position`.
15	**multimap::erase** `key` Removes mapped value associated with key `k`.
16	**multimap::erase** `range version` Removes range of element from the the multimap.
17	**multimap::erase** `range version` Removes range of element from the the multimap.
18	**multimap::find** Finds an element associated with key `k`.
19	**multimap::get_allocator** Returns an allocator associated with multimap.
20	**multimap::insert** `single element` Extends container by inserting new element in multimap.
21	**multimap::insert** `hint version` Extends container by inserting new element in multimap.
22	**multimap::insert** `range version` Extends container by inserting new elements in the multimap.
23	**multimap::insert** `move hint verstion` Extends multimap by inserting new element.
24	**multimap::insert** `initializer list version` Extends multimap by inserting new element from initializer list.
25	**multimap::key_comp** Returns a function object that compares the keys, which is a copy of this container's constructor argument `comp`.
26	**multimap::lower_bound** Returns an iterator pointing to the first element which is not less than key `k`.
27	**multimap::max_size** Returns the maximum number of elements can be held by multimap.
28	**multimap::operator=** `copy version` Assign new contents to the multimap by replacing old ones and modifies size if necessary.
29	**multimap::operator=** `move version` Move the contents of one multimap into another and modifies size if necessary.
30	**multimap::operator=** `initializer list version` Copy elements from initializer list to multimap.
31	**multimap::rbegin** Returns a reverse iterator which points to the last element of the multimap.
32	**multimap::rend** Returns a reverse iterator which points to the reverse end of the multimap.
33	**multimap::size** Returns the number of elements present in the multimap.
34	**multimap::swap** Exchanges the content of multimap with contents of multimap `x`.
35	**multimap::upper_bound** Returns an iterator pointing to the first element which is greater than key `k`.
36	**multimap::value_comp** Returns a function object that compares objects of type `std::multimap::value_type`.

Non-member overloaded functions

S.N.	Method & Description
1	**operator==** Tests whether two multimaps are equal or not.
2	**operator!=** Tests whether two multimaps are equal or not.
3	**operator<** Tests whether first multimap is less than other or not.
4	**multimap::operator<=** Tests whether first multimap is less than or equal to other or not.
5	**operator>** Tests whether first multimap is greater than other or not.
6	**operator>=** Tests whether first multimap is greater than or equal to other or not.
7	**swap()** Exchanges the content of multimap with contents of multimap x.