C++
Information
Tutorials
Reference
Articles
Forum

Reference
C library:
Containers:
<array>
<deque>
<forward_list>
<list>
<map>
<queue>
<set>
<stack>
<unordered_map>
<unordered_set>
<vector>
Input/Output:
Multi-threading:
Other:

<map>
map
multimap

map
map::map
map::~map
member functions:
map::at
map::begin
map::cbegin
map::cend
map::clear
map::count
map::crbegin
map::crend
map::emplace
map::emplace_hint
map::empty
map::end
map::equal_range
map::erase
map::find
map::get_allocator
map::insert
map::key_comp
map::lower_bound
map::max_size
map::operator=
map::operator[]
map::rbegin
map::rend
map::size
map::swap
map::upper_bound
map::value_comp
non-member overloads:
relational operators (map)
swap (map)

class template

std::map

<map>

```
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;
```

Map

Maps are associative containers that store elements formed by a combination of a *key value* and a *mapped value*, following a specific order.

In a `map`, the *key values* are generally used to sort and uniquely identify the elements, while the *mapped values* store the content associated to this *key*. The types of *key* and *mapped value* may differ, and are grouped together in member type `value_type`, which is a pair type combining both:

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

Internally, the elements in a `map` are always sorted by its *key* following a specific *strict weak ordering* criterion indicated by its internal comparison object (of type `Compare`).

`map` containers are generally slower than unordered_map containers to access individual elements by their *key*, but they allow the direct iteration on subsets based on their order.

The mapped values in a map can be accessed directly by their corresponding key using the *bracket operator* ((operator[]).

Maps are typically implemented as *binary search trees*.

Container properties

Associative
Elements in associative containers are referenced by their *key* and not by their absolute position in the container.

Ordered
The elements in the container follow a strict order at all times. All inserted elements are given a position in this order.

Map
Each element associates a *key* to a *mapped value*: Keys are meant to identify the elements whose main content is the *mapped value*.

Unique keys
No two elements in the container can have equivalent *keys*.

Allocator-aware
The container uses an allocator object to dynamically handle its storage needs.

Template parameters

`Key`
Type of the *keys*. Each element in a `map` is uniquely identified by its key value.
Aliased as member type `map::key_type`.

`T`
Type of the mapped value. Each element in a `map` stores some data as its mapped value.
Aliased as member type `map::mapped_type`.

`Compare`
A binary predicate that takes two element keys as arguments and returns a `bool`. The expression `comp(a,b)`, where *comp* is an object of this type and *a* and *b* are key values, shall return `true` if *a* is considered to go before *b* in the *strict weak ordering* the function defines.
The `map` object uses this expression to determine both the order the elements follow in the container and whether two element keys are equivalent (by comparing them reflexively: they are equivalent if `!comp(a,b) && !comp(b,a)`). No two elements in a `map` container can have equivalent keys.
This can be a function pointer or a function object (see constructor for an example). This defaults to `less<T>`, which returns the same as applying the *less-than operator* (`a<b`).
Aliased as member type `map::key_compare`.

`Alloc`
Type of the allocator object used to define the storage allocation model. By default, the allocator class template is used, which defines the simplest memory allocation model and is value-independent.
Aliased as member type `map::allocator_type`.

Member types

C++98 C++11

member type	definition	notes
key_type	The first template parameter (Key)	
mapped_type	The second template parameter (T)	
value_type	pair<const key_type,mapped_type>	
key_compare	The third template parameter (Compare)	defaults to: less<key_type>
value_compare	*Nested function class to compare elements*	see value_comp
allocator_type	The fourth template parameter (Alloc)	defaults to: allocator<value_type>
reference	allocator_type::reference	for the default allocator: value_type&
const_reference	allocator_type::const_reference	for the default allocator: const value_type&
pointer	allocator_type::pointer	for the default allocator: value_type*
const_pointer	allocator_type::const_pointer	for the default allocator: const value_type*
iterator	a bidirectional iterator to value_type	convertible to const_iterator
const_iterator	a bidirectional iterator to const value_type	
reverse_iterator	reverse_iterator<iterator>	
const_reverse_iterator	reverse_iterator<const_iterator>	
difference_type	a signed integral type, identical to: iterator_traits<iterator>::difference_type	usually the same as ptrdiff_t
size_type	an unsigned integral type that can represent any non-negative value of difference_type	usually the same as size_t

Member functions

(constructor)	Construct map (public member function)
(destructor)	Map destructor (public member function)
operator=	Copy container content (public member function)

Iterators:

begin	Return iterator to beginning (public member function)
end	Return iterator to end (public member function)
rbegin	Return reverse iterator to reverse beginning (public member function)
rend	Return reverse iterator to reverse end (public member function)
cbegin	Return const_iterator to beginning (public member function)
cend	Return const_iterator to end (public member function)
crbegin	Return const_reverse_iterator to reverse beginning (public member function)
crend	Return const_reverse_iterator to reverse end (public member function)

Capacity:

empty	Test whether container is empty (public member function)
size	Return container size (public member function)
max_size	Return maximum size (public member function)

Element access:

operator[]	Access element (public member function)
at	Access element (public member function)

Modifiers:

insert	Insert elements (public member function)
erase	Erase elements (public member function)
swap	Swap content (public member function)
clear	Clear content (public member function)
emplace	Construct and insert element (public member function)
emplace_hint	Construct and insert element with hint (public member function)

Observers:

key_comp	Return key comparison object (public member function)
value_comp	Return value comparison object (public member function)

Operations:

find	Get iterator to element (public member function)

count	Count elements with a specific key (public member function)
lower_bound	Return iterator to lower bound (public member function)
upper_bound	Return iterator to upper bound (public member function)
equal_range	Get range of equal elements (public member function)

Allocator:

get_allocator	Get allocator (public member function)