std::map

Defined in header <map>

```
template<
    class Key,
    class T,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<std::pair<const Key, T>>
> class map;
```
(1)

```
namespace pmr {
    template<
        class Key,
        class T,
        class Compare = std::less<Key>
    > using map = std::map<Key, T, Compare,
                           std::pmr::polymorphic_allocator<std::pair<const Key, T>>>;
}
```
(2) (since C++17)

`std::map` is a sorted associative container that contains key-value pairs with unique keys. Keys are sorted by using the comparison function `Compare`. Search, removal, and insertion operations have logarithmic complexity. Maps are usually implemented as Red–black trees .

Iterators of `std::map` iterate in ascending order of keys, where ascending is defined by the comparison that was used for construction. That is, given

- `m`, a `std::map`
- `it_l` and `it_r`, dereferenceable iterators to `m`, with `it_l < it_r`.

`m.value_comp()(*it_l, *it_r) == true` (least to greatest if using the default comparison).

Everywhere the standard library uses the *Compare* requirements, uniqueness is determined by using the equivalence relation. In imprecise terms, two objects `a` and `b` are considered equivalent (not unique) if neither compares less than the other: `!comp(a, b) && !comp(b, a)`.

`std::map` meets the requirements of *Container*, *AllocatorAwareContainer*, *AssociativeContainer* and *ReversibleContainer*.

All member functions of `std::map` are `constexpr`: it is possible to create and use `std::map` objects in the evaluation of a constant expression. (since C++26)

However, `std::map` objects generally cannot be `constexpr`, because any dynamically allocated storage must be released in the same evaluation of constant expression.

Template parameters

This section is incomplete
Reason: Add descriptions of the template parameters.

Member types

Type	Definition
`key_type`	`Key`
`mapped_type`	`T`
`value_type`	`std::pair<const Key, T>`
`size_type`	Unsigned integer type (usually `std::size_t`)
`difference_type`	Signed integer type (usually `std::ptrdiff_t`)
`key_compare`	`Compare`
`allocator_type`	`Allocator`
`reference`	`value_type&`
`const_reference`	`const value_type&`
`pointer`	`Allocator::pointer` (until C++11) `std::allocator_traits<Allocator>::pointer` (since C++11)
`const_pointer`	`Allocator::const_pointer` (until C++11) `std::allocator_traits<Allocator>::const_pointer` (since C++11)

`iterator`	*LegacyBidirectionalIterator* and *ConstexprIterator* (since C++26) to `value_type`
`const_iterator`	*LegacyBidirectionalIterator* and *ConstexprIterator* (since C++26) to `const value_type`
`reverse_iterator`	`std::reverse_iterator<iterator>`
`const_reverse_iterator`	`std::reverse_iterator<const_iterator>`
`node_type` (since C++17)	a specialization of node handle representing a container node

`insert_return_type` (since C++17)	type describing the result of inserting a `node_type`, a specialization of
	`template<class Iter, class NodeType>`
	`struct /*unspecified*/`
	`{`
	`  Iter position;`
	`  bool inserted;`
	`  NodeType node;`
	`};`
	instantiated with template arguments `iterator` and `node_type`.

Member classes

`value_compare`	compares objects of type `value_type` (class)

Member functions

(constructor)	constructs the `map` (public member function)
(destructor)	destructs the `map` (public member function)
`operator=`	assigns values to the container (public member function)
`get_allocator`	returns the associated allocator (public member function)

Element access

`at`	access specified element with bounds checking (public member function)
`operator[]`	access or insert specified element (public member function)

Iterators

`begin` **`cbegin`** (C++11)	returns an iterator to the beginning (public member function)
`end` **`cend`** (C++11)	returns an iterator to the end (public member function)
`rbegin` **`crbegin`** (C++11)	returns a reverse iterator to the beginning (public member function)
`rend` **`crend`** (C++11)	returns a reverse iterator to the end (public member function)

Capacity

`empty`	checks whether the container is empty (public member function)
`size`	returns the number of elements (public member function)
`max_size`	returns the maximum possible number of elements (public member function)

Modifiers

`clear`	clears the contents (public member function)
`insert`	inserts elements or nodes (since C++17) (public member function)
`insert_range` (C++23)	inserts a range of elements (public member function)

insert_or_assign (C++17)	inserts an element or assigns to the current element if the key already exists (public member function)
emplace (C++11)	constructs element in-place (public member function)
emplace_hint (C++11)	constructs elements in-place using a hint (public member function)
try_emplace (C++17)	inserts in-place if the key does not exist, does nothing if the key exists (public member function)
erase	erases elements (public member function)
swap	swaps the contents (public member function)
extract (C++17)	extracts nodes from the container (public member function)
merge (C++17)	splices nodes from another container (public member function)

Lookup

count	returns the number of elements matching specific key (public member function)
find	finds element with specific key (public member function)
contains (C++20)	checks if the container contains element with specific key (public member function)
equal_range	returns range of elements matching a specific key (public member function)
lower_bound	returns an iterator to the first element *not less* than the given key (public member function)
upper_bound	returns an iterator to the first element *greater* than the given key (public member function)

Observers

key_comp	returns the function that compares keys (public member function)
value_comp	returns the function that compares keys in objects of type `value_type` (public member function)

Non-member functions

operator== **operator!=** (removed in C++20) **operator<** (removed in C++20) **operator<=** (removed in C++20) **operator>** (removed in C++20) **operator>=** (removed in C++20) **operator<=>** (C++20)	lexicographically compares the values of two `map`s (function template)
std::swap(`std::map`)	specializes the `std::swap` algorithm (function template)
erase_if(`std::map`) (C++20)	erases all elements satisfying specific criteria (function template)

Deduction guides (since C++17)

Notes

Feature-test macro	Value	Std	Feature
`__cpp_lib_containers_ranges`	`202202L`	(C++23)	Ranges construction and insertion for containers
`__cpp_lib_constexpr_map`	`202502L`	(C++26)	`constexpr std::map`

Example

Run this code

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

```
#include <string>
#include <string_view>

void print_map(std::string_view comment, const std::map<std::string, int>& m)
{
    std::cout << comment;
    // Iterate using C++17 facilities
    for (const auto& [key, value] : m)
        std::cout << '[' << key << "] = " << value << "; ";

// C++11 alternative:
//  for (const auto& n : m)
//      std::cout << n.first << " = " << n.second << "; ";
//
// C++98 alternative:
//  for (std::map<std::string, int>::const_iterator it = m.begin(); it != m.end(); ++it)
//      std::cout << it->first << " = " << it->second << "; ";

    std::cout << '\n';
}

int main()
{
    // Create a map of three (string, int) pairs
    std::map<std::string, int> m{{"CPU", 10}, {"GPU", 15}, {"RAM", 20}};

    print_map("1) Initial map: ", m);

    m["CPU"] = 25; // update an existing value
    m["SSD"] = 30; // insert a new value
    print_map("2) Updated map: ", m);

    // Using operator[] with non-existent key always performs an insert
    std::cout << "3) m[UPS] = " << m["UPS"] << '\n';
    print_map("4) Updated map: ", m);

    m.erase("GPU");
    print_map("5) After erase: ", m);

    std::erase_if(m, [](const auto& pair){ return pair.second > 25; });
    print_map("6) After erase: ", m);
    std::cout << "7) m.size() = " << m.size() << '\n';

    m.clear();
    std::cout << std::boolalpha << "8) Map is empty: " << m.empty() << '\n';
}
```

Output:

```
1) Initial map: [CPU] = 10; [GPU] = 15; [RAM] = 20;
2) Updated map: [CPU] = 25; [GPU] = 15; [RAM] = 20; [SSD] = 30;
3) m[UPS] = 0
4) Updated map: [CPU] = 25; [GPU] = 15; [RAM] = 20; [SSD] = 30; [UPS] = 0;
5) After erase: [CPU] = 25; [RAM] = 20; [SSD] = 30; [UPS] = 0;
6) After erase: [CPU] = 25; [RAM] = 20; [UPS] = 0;
7) m.size() = 3
8) Map is empty: true
```

Defect reports

The following behavior-changing defect reports were applied retroactively to previously published C++ standards.

DR	Applied to	Behavior as published	Correct behavior
LWG 230 (https://cplusplus.github.io/LWG/issue230)	C++98	Key was not required to be *CopyConstructible* (a key of type Key might not be able to be constructed)	Key is also required to be *CopyConstructible*
LWG 464 (https://cplusplus.github.io/LWG/issue464)	C++98	accessing a const `map` by key was inconvenient	`at` function provided

See also

`multimap`	collection of key-value pairs, sorted by keys (class template)

unordered_map (C++11)	collection of key-value pairs, hashed by keys, keys are unique (class template)
flat_map (C++23)	adapts two containers to provide a collection of key-value pairs, sorted by unique keys (class template)