**Maps and Multimaps**

Maps and multimaps are containers that manage key/value pairs as elements. These containers sort their elements automatically, according to a certain sorting criterion that is used for the key.

**Difference**: multimaps allow duplicates, whereas maps do not.

header file **#include <map>**

Types are defined as class templates inside namespace std:

namespace std {

template< typename Key,

typename T,

typename Compare = less<Key>,

typename Allocator = allocator<pair<const Key,T> > >

class map;

template < typename Key,

typename T,

typename Compare = less<Key>,

typename Allocator = allocator<pair<const Key,T> > >

class multimap;

}

**Key**: type of the element’s key

**T**: type of the element’s associated value

The elements of a map or a multimap may have any types Key and T that meet the following two requirements:

1. Both key and value must be copyable or movable.
2. The key must be comparable with the sorting criterion.

**Compare**: optional third template parameter defines the sorting criterion. As for sets, this sorting criterion must define a “strict weak ordering”. The elements are sorted according to their keys, so the value doesn’t matter for the order of the elements.

For multimaps, the order of elements with equivalent keys is random but stable. Thus, insertions and erasures preserve the relative ordering of equivalent elements (guaranteed since C++11).

**Allocator**: optional fourth template parameter defines the memory model.

The element type (value\_type) is a pair <const Key, T>. The key of all elements inside a map and a multimap is considered to be constant.

# Abilities of Maps and Multimaps

maps and multimaps are usually implemented as balanced binary trees. It follows from the complexity of the map and multimap operations.

Maps and multimaps sort their elements automatically, according to the element’s keys, and so have good performance when searching for elements that have a certain key.

You may not change the key of an element directly, because doing so might compromise the correct order.

To modify the key of an element, you must remove the element that has the old key and insert a new element that has the new key and the old value.

As a consequence, from the iterator’s point of view, the element’s key is constant. However, a direct modification of the value of the element is still possible, provided that the type of the value is not constant.

# Map and Multimap Operations

## Create, Copy, and Destroy

constructors and destructors of maps and multimaps:

|  |  |
| --- | --- |
| Operation | Effect |
| map c | Default constructor; creates an empty map/multimap without any elements |
| map c(op) | Creates an empty map/multimap that uses op as the sorting criterion |
| map c(c2) | Copy constructor; creates a copy of another map/multimap of the same type (all elements are copied) |
| map c = c2 | Copy constructor; creates a copy of another map/multimap of the same type (all elements are copied) |
| map c(rv) | Move constructor; creates a new map/multimap of the same type, taking the contents of the rvalue rv (since C++11) |
| map c = rv | Move constructor; creates a new map/multimap of the same type, taking the contents of the rvalue rv (since C++11) |
| map c(beg, end) | Creates a map/multimap initialized by the elements of the range [beg,end) |
| map c(beg, end, op) | Creates a map/multimap with the sorting criterion op initialized by the elements of the range [beg,end) |
| map c(initlist) | Creates a map/multimap initialized with the elements of initializer list initlist (since C++11) |
| map c = initlist | Creates a map/multimap initialized with the elements of initializer list initlist (since C++11) |
| c.~map() | Destroys all elements and frees the memory |

map may be one of the following types:

|  |  |
| --- | --- |
| map | Effect |
| map<Key, Val> | A map that by default sorts keys with less<> (operator <) |
| map<Key, Val, Op> | A map that by default sorts keys with Op |
| multimap<Key, Val> | A multimap that by default sorts keys with less<> (operator <) |
| multimap<Key, Val, Op> | A multimap that by default sorts keys with Op |

sorting criterion can be defined in two ways:

1. **As a template parameter.**

std::map<float, std::string, **std::greater<float>**> coll;

In this case, the sorting criterion is part of the type. Thus, the type system ensures that only containers with the same sorting criterion can be combined.

1. **As a constructor parameter.**

In this case, you might have a type for several sorting criteria, and the initial value or state of the sorting criteria might differ.

This is useful when processing the sorting criterion at runtime or when sorting criteria are needed that are different but of the same data type.

Default sorting criterion is function object less<>.

## Nonmodifying Operations

|  |  |
| --- | --- |
| Operation | Effect |
| c.key\_comp() | Returns the comparison criterion |
| c.value\_comp() | Returns the comparison criterion for values as a whole (an object that compares the key in a key/value pair) |
| c.empty() | Returns whether the container is empty (equivalent to size()==0 but might be faster) |
| c.size() | Returns the current number of elements |
| c.max\_size() | Returns the maximum number of elements possible |
| c1 == c2 | Returns whether c1 is equal to c2 (calls == for the elements) |
| c1 != c2 | Returns whether c1 is not equal to c2 (equivalent to !(c1==c2)) |
| c1 < c2 | Returns whether c1 is less than c2 |
| c1 > c2 | Returns whether c1 is greater than c2 (equivalent to c2<c1) |
| c1 <= c2 | Returns whether c1 is less than or equal to c2 (equivalent to !(c2<c1)) |
| c1 >= c2 | Returns whether c1 is greater than or equal to c2 (equivalent to !(c1<c2)) |

Comparisons are provided only for containers of the same type. Thus, the key, the value, and the sorting criterion must be of the same type. Otherwise, a type error occurs at compile time.

std::map<float, std::string> c1; // sorting criterion: less<>

std::map<float, std::string, std::greater<float>> c2;

// CE error: no match for 'operator==' (operand types are 'std::map<float, std::\_\_cxx11::basic\_string<char> >' and 'std::map<float, std::\_\_cxx11::basic\_string<char>, std::greater<float> >')

if (c1 == c2) { cout << "c1 and c2 or equal" << endl;}

Checking whether a container is less than another container is done by a lexicographical comparison.

bool lexicographical\_compare (InputIterator1 beg1, InputIterator1 end1, InputIterator2 beg2, InputIterator2 end2)

bool lexicographical\_compare (InputIterator1 beg1, InputIterator1 end1, InputIterator2 beg2, InputIterator2 end2, CompFunc op)

Lexicographical comparison means that sequences are compared element-by-element until any of the following occurs:

1. When two elements are not equal, the result of their comparison is the result of the whole comparison.
2. When one sequence has no more elements, the sequence that has no more elements is less than the other. Thus, the comparison yields true if the first sequence is the one that has no more elements.
3. When both sequences have no more elements, both sequences are equal, and the result of the comparison is false.

To compare containers of different types (different sorting criterion), you must use the comparing algorithms:

bool equal (InputIterator1 beg, InputIterator1 end, InputIterator2 cmpBeg)

bool equal (InputIterator1 beg, InputIterator1 end, InputIterator2 cmpBeg, BinaryPredicate op)

## Assignments

maps and multimaps provide only the fundamental assignment operations that all containers provide

|  |  |
| --- | --- |
| Operation | Effect |
| c = c2 | Assigns all elements of c2 to c |
| c = rv | Move assigns all elements of the rvalue rv to c (since C++11) |
| c = initlist | Assigns all elements of the initializer list initlist to c (since C++11) |
| c1.swap(c2) | Swaps the data of c1 and c2 |
| swap(c1,c2) | Swaps the data of c1 and c2 |

For these operations, both containers must have the same type. In particular, the type of the comparison criteria must be the same, although the comparison criteria themselves may be different. If the criteria are different, they also get assigned or swapped.

## Iterator Functions and Element Access

|  |  |
| --- | --- |
| Operation | Effect |
| c.begin() | Returns a bidirectional iterator for the first element |
| c.end() | Returns a bidirectional iterator for the position after the last element |
| c.cbegin() | Returns a constant bidirectional iterator for the first element (since C++11) |
| c.cend() | Returns a constant bidirectional iterator for the position after the last element (since C++11) |
| c.rbegin() | Returns a reverse iterator for the first element of a reverse iteration |
| c.rend() | Returns a reverse iterator for the position after the last element of a reverse iteration |
| c.crbegin() | Returns a constant reverse iterator for the first element of a reverse iteration (since C++11) |
| c.crend() | Returns a constant reverse iterator for the position after the last element of a reverse iteration (since C++11) |

Maps and multimaps do not provide direct element access. To access elements use range-based for loops or iterators.

**Exception:** maps provide at() and the subscript operator([]) to access elements directly.

As for all associative container classes, the iterators are bidirectional. Thus, you can’t use them in algorithms that are provided only for random-access iterators, such as algorithms for sorting or random shuffling.

Since key is constant any modifying algorithm cannot be called if the destination is a map or a multimap. To remove elements in maps and multimaps, you can use only member functions provided by the container.

Manipulation algorithms (those that remove elements and those that reorder or modify elements) cannot work with associative or unordered containers: Associative and unordered containers can’t be used as a destination.

Reason: If they would work for associative or unordered containers, modifying algorithms could change the value or position of elements, thereby violating the order maintained by the container.

In order to avoid compromising the internal order, every iterator for an associative and unordered container is declared as an iterator for a constant value or key. Thus, manipulating elements of or in associative and unordered containers results in a failure at compile time.

This problem also prevents you from calling removing algorithms for associative containers, because these algorithms manipulate elements implicitly. The values of “removed” elements are overwritten by the following elements that are not removed.

How does one remove elements in associative containers? Call their member functions.

**Modifying Key and Value**

Modifying Key results in compile time failure. However, changing the value of the element is no problem, as long as elem is declared as a nonconstant reference and the type of the value is not constant.

#include <iostream>

#include <string>

#include <map>

int main () {

std::map<std::string,int> mymap = { { "alpha", 0 },

{ "beta", 0 },

{ "gamma", 0 } };

mymap.at("alpha") = 10;

mymap.at("beta") = 20;

mymap.at("gamma") = 30;

for (auto & x : mymap) {

std::cout << x.first << ": " << x.second << std::endl;

}

for (auto pos = mymap.begin(); pos != mymap.end(); ++pos) {

//pos->first = "update";

// CE error: passing 'const std::\_\_cxx11::basic\_string<char>' as 'this' argument discards qualifiers

pos->second = (pos->second) + 30;

}

for (auto & x : mymap) {

std::cout << x.first << ": " << x.second << std::endl;

}

return 0;

}

Output:

alpha: 10

beta: 20

gamma: 30

alpha: 40

beta: 50

gamma: 60

To change the key of an element, replace the old element with a new element that has the same value.

Method - 1

more convenient way to modify the key of an element

coll["new\_key"] = coll["old\_key"]; // insert new element with value of old element

coll.erase("old\_key"); // remove old element

Method - 2

namespace MyLib {

template <typename Cont>

inline bool replace\_key (Cont& c,

const typename Cont::key\_type& old\_key,

const typename Cont::key\_type& new\_key) {

typename Cont::iterator pos;

pos = c.find(old\_key);

if (pos != c.end()) {

// insert new element with value of old element

c.insert(typename Cont::value\_type(new\_key, pos->second));

// remove old element

c.erase(pos);

return true;

}

else {

// key not found

return false;

}

}

}

## Inserting and Removing Elements

|  |  |
| --- | --- |
| Operation | Effect |
| c.insert(val) | Inserts a copy of val and  returns the position of the new element and, for maps, whether it succeeded |
| c.insert(pos, val) | Inserts a copy of val and  returns the position of the new element (pos is used as a hint pointing to where the insert should start the search) |
| c.insert(beg, end) | Inserts a copy of all elements of the range [beg,end) (returns nothing) |
| c.insert(initlist) | Inserts a copy of all elements in the initializer list initlist (returns nothing; since C++11) |
| c.emplace(args...) | Inserts a copy of an element initialized with args and  returns the position of the new element and, for maps, whether it succeeded (since C++11) |
| c.emplace\_hint(pos, args...) | Inserts a copy of an element initialized with args and  returns the position of the new element (pos is used as a hint pointing to where the insert should start the search) |
| c.erase(val) | Removes all elements equal to val and  returns the number of removed elements |
| c.erase(pos) | Removes the element at iterator position pos and  returns the following position (returned nothing before C++11) |
| c.erase(beg, end) | Removes all elements of the range [beg,end) and  returns the following position (returned nothing before C++11) |
| c.clear() | Removes all elements (empties the container) |

### Insertion

For multimaps, since C++11 it is guaranteed that insert(), emplace(), and erase() preserve the relative ordering of equivalent elements, and that inserted elements are placed at the end of existing equivalent values.

Since C++11, the most convenient way to insert elements is to pass them as an initializer list

std::map<std::string,float> coll;

coll.insert({"otto", 22.3});

Alternatively, there are three other ways to pass a value into a map or a multimap:

1. value\_type()
2. pair<>
3. make\_pair()
4. **value\_type()**

To avoid implicit type conversion, you could pass the correct type explicitly by using value\_type

std::map<std::string,float> coll;

...

coll.insert(std::map<std::string,float>::value\_type("otto", 22.3));

or

coll.insert(decltype(coll)::value\_type("otto", 22.3));

1. **pair<>**

std::map<std::string,float> coll;

...

// use implicit conversion:

coll.insert(std::pair<std::string, float>("otto", 22.3));

// use no implicit conversion:

coll.insert(std::pair<const std::string, float>("otto", 22.3));

1. **make\_pair()**

most convenient way before C++11 was to use make\_pair()

std::map<std::string,float> coll;

...

coll.insert(std::make\_pair("otto", 22.3));

The necessary type conversions are performed by the insert() member template.

When using emplace() to insert a new element by passing the values for its construction, you have to pass two lists of arguments: one for the key and one for the element. The most convenient way to do this is as follows:

std::map<std::string,std::complex<float>> m;

m.emplace(std::piecewise\_construct, // pass tuple elements as arguments

std::make\_tuple("hello"), // elements for the key

std::make\_tuple(3.4,7.8)); // elements for the value

## Special Search Operations

maps and multimaps provide special search member functions that perform better because of their internal tree structure

|  |  |
| --- | --- |
| Operation | Effect |
| c.count(key) | Returns the number of elements with key val |
| c.find(key) | Returns the position of the first element with key val (or end() if none found) |
| c.lower\_bound(key) | Returns the first position where an element with key val would get inserted (the first element with a key >= val) |
| c.upper\_bound(key) | Returns the last position where an element with key val would get inserted (the first element with a key > val) |
| c.equal\_range(key) | Returns a range with all elements with a key equal to val (i.e., the first and last positions, where an element with key val would get inserted) |

To search for an element that has a certain value use find\_if() algorithm, or program an explicit loop.

std::multimap<std::string,float> coll;

std::multimap<std::string,float>::iterator pos;

for (pos = coll.begin(); pos != coll.end(); ++pos) {

if (pos->second == value) {

do\_something();

}

}

Be careful when using a loop to remove elements. It might happen that you saw off the branch on which you are sitting. Check topic [Removal](#_Removal)

Using the find\_if() algorithm is even more complicated than writing a loop, because you have to provide a function object that compares the value of an element with a certain value.

### Removal

coll.erase(key); // remove all elements with the passed key

// to remove first element with passed key

auto pos = coll.find(key); // member function find

if (pos != coll.end()) { coll.erase(pos); }

\*\*\*

When removing elements, be careful not to saw off the branch on which you are sitting. There is a big danger that you will remove an element to which your iterator is referring. Example:

std::map<std::string,float> coll;

...

for (auto pos = coll.begin(); pos != coll.end(); ++pos) {

if (pos->second == value) {

coll.erase(pos); // RUNTIME ERROR !!!

}

}

Calling erase() for the element to which you are referring with pos invalidates pos as an iterator of coll. Thus, if you use pos after removing its element without any reinitialization, all bets are off. In fact, calling ++pos results in undefined behavior.

Correct way to remove elements to which an iterator refers before C++11:

for (pos = coll.begin(); pos != coll.end(); ) {

if (pos->second == value) {

coll.erase(pos++); // pos++ increments pos so that it refers to the next element but yields a copy of its original value

}

else {

++pos;

}

}

Since C++11, a solution is easy because erase() always returns the value of the following element:

for (auto pos = coll.begin(); pos != coll.end(); ) {

if (pos->second == value) {

pos = coll.erase(pos); // possible only since C++11

}

else {

++pos;

}

}

For multimaps, all insert(), emplace(), and erase() operations preserve the relative order of equivalent elements. Since C++11, calling insert(val) or emplace(args...) guarantees that the new element is inserted at the end of the range of equivalent elements.

# Using Maps as Associative Arrays

Nonconstant maps provide a subscript operator for direct element access.

In addition, since C++11, a corresponding member function at() is provided for constant and nonconstant maps.

|  |  |
| --- | --- |
| Operation | Effect |
| c[key] | Inserts an element with key, if it does not yet exist, and returns a reference to the value of the element with key (only for nonconstant maps) |
| c.at(key) | Returns a reference to the value of the element with key (since C++11) |

at() yields the value of the element with the passed key and throws an exception object of type out\_of\_range if no such element is present.

For operator [ ], the index may have any type rather than only an integral type. Such an interface is the interface of a so-called associative array.

For operator [ ], the type of the index is not the only difference from ordinary arrays. In addition, you can’t have a wrong index. If you use a key as the index for which no element yet exists, a new element gets inserted into the map automatically. The value of the new element is initialized by the default constructor of its type. Thus, to use this feature, you can’t use a value type that has no default constructor. Note that the fundamental data types provide a default constructor that initializes their values to zero.

This behavior of an associative array has both advantages and disadvantages:

**Advantage:** Insert new elements into a map with a more convenient interface.

std::map<std::string,float> coll; // empty collection

coll["otto"] = 7.7;

This way of inserting elements is slower than the usual way for maps. The reason is that the new value is first initialized by the default value of its type, which is then overwritten by the correct value.

**Disadvantage:** You might insert new elements by accident or mistake.

std::cout << coll["ottto"];

# Exception Handling

For multiple-element insert operations, the need to keep elements sorted makes full recovery from throws impractical.

Thus, all single-element insert operations support commit or rollback behavior. That is, they either succeed or have no effect.

In addition, it is guaranteed that all multiple-element delete operations always succeed or have no effect, provided that the comparison criterion does not throw.

If copying/assigning the comparison criterion may throw, swap() may throw.

# END