map Class

Visual Studio 2015

The new home for Visual Studio documentation is Visual Studio 2017 Documentation on docs.microsoft.com.

The latest version of this topic can be found at map Class.

Used for the storage and retrieval of data from a collection in which each element is a pair that has both a data value and a sort key. The value of the key is unique and is used to automatically sort the data.

The value of an element in a map can be changed directly. The key value is a constant and cannot be changed. Instead, key values associated with old elements must be deleted, and new key values must be inserted for new elements.

Syntax

```
template <class Key,
    class Type,
    class Traits = less<Key>,
    class Allocator=allocator<pair <const Key, Type>>>
class map;
```

Parameters

`Key`
The key data type to be stored in the map.

`Type`
The element data type to be stored in the map.

`Traits`
The type that provides a function object that can compare two element values as sort keys to determine their relative order in the map. This argument is optional and the binary predicate `less<``Key``>` is the default value.

In C++14 you can enable heterogeneous lookup by specifying the std::less<> predicate that has no type parameters. For more information, see Heterogeneous Lookup in Associative Containers

`Allocator`
The type that represents the stored allocator object that encapsulates details about the map's allocation and deallocation of memory. This argument is optional and the default value is `allocator<pair``<const``Key, Type``> >`.

Remarks

The Standard Template Library (STL) map class is:

- A container of variable size that efficiently retrieves element values based on associated key values.
- Reversible, because it provides bidirectional iterators to access its elements.
- Sorted, because its elements are ordered by key values according to a specified comparison function.
- Unique. because each of its elements must have a unique key.
- A pair-associative container, because its element data values are distinct from its key values.
- A template class, because the functionality it provides is generic and independent of element or key type. The data types used for elements and keys are specified as parameters in the class template together with the comparison function and allocator.

The iterator provided by the map class is a bidirectional iterator, but the insert and map class member functions have versions that take as template parameters a weaker input iterator, whose functionality requirements are fewer than those guaranteed by the class of bidirectional iterators. The different iterator concepts are related by refinements in their functionality. Each iterator concept has its own set of requirements, and the algorithms that work with it must be limited by those requirements. An input iterator may be dereferenced to refer to some object and may be incremented to the next iterator in the sequence.

We recommend that you base the choice of container type on the kind of searching and inserting that is required by the application. Associative containers are optimized for the operations of lookup, insertion, and removal. The member functions that explicitly support these operations perform them in a worst-case time that is proportional to the logarithm of the number of elements in the container. Inserting elements invalidates no iterators, and removing elements invalidates only those iterators that specifically pointed to the removed elements.

We recommend that you make the map the associative container of choice when conditions that associate values with keys are satisfied by the application. A model for this kind of structure is an ordered list of uniquely occurring key words that have associated string values that provide definitions. If a word has more than one correct definition, so that key is not unique, then a multimap would be the container of choice. If just the list of words is being stored, then a set would be the appropriate container. If multiple occurrences of the words are allowed, then a multiset would be appropriate.

The map orders the elements it controls by calling a stored function object of type key_compare. This stored object is a comparison function that is accessed by calling the key_comp method. In general, any two given elements are compared to determine whether one is less than the other or whether they are equivalent. As all elements are compared, an ordered sequence of non-equivalent elements is created.

Note

The comparison function is a binary predicate that induces a strict weak ordering in the standard mathematical sense. A binary predicate f(x,y) is a function object that has two argument objects x and y, and a return value of `true` or `false`. An ordering imposed on a set is a strict weak ordering if the binary predicate is irreflexive, antisymmetric, and transitive, and if equivalence is transitive, where two objects x and y are defined to be equivalent when both f(x,y)and f(y,x) are `false`. If the stronger condition of equality between keys replaces that of equivalence, the ordering becomes total (in the sense that all the elements are ordered with regard to one other), and the keys matched will be indiscernible from one other.

In C++14 you can enable heterogeneous lookup by specifying the `std::less<>` or `std::greater<>` predicate that has no type parameters. For more information, see Heterogeneous Lookup in Associative Containers

Members

Constructors

map	Constructs a list of a specific size or with elements of a specific value or with a specific `allocator` or as a copy of some other map.

Typedefs

allocator_type	A typedef for the `allocator` class for the map object.
const_iterator	A typedef for a bidirectional iterator that can read a `const` element in the map.
const_pointer	A typedef for a pointer to a `const` element in a map.
const_reference	A typedef for a reference to a `const` element stored in a map for reading and performing `const` operations.
const_reverse_iterator	A type that provides a bidirectional iterator that can read any `const` element in the map.
difference_type	A signed integer typedef for the number of elements of a map in a range between elements pointed to by iterators.
iterator	A typedef for a bidirectional iterator that can read or modify any element in a map.
key_compare	A typedef for a function object that can compare two sort keys to determine the relative order of two elements in the map.
key_type	A typedef for the sort key stored in each element of the map.
mapped_type	A typedef for the data stored in each element of a map.
pointer	A typedef for a pointer to a `const` element in a map.
reference	A typedef for a reference to an element stored in a map.
reverse_iterator	A typedef for a bidirectional iterator that can read or modify an element in a reversed map.

size_type	An unsigned integer typedef for the number of elements in a map
value_type	A typedef for the type of object stored as an element in a map.

Member Functions

at	Finds an element with a specified key value.
begin	Returns an iterator that points to the first element in the map.
cbegin	Returns a const iterator that points to the first element in the map.
cend	Returns a const past-the-end iterator.
clear	Erases all the elements of a map.
count	Returns the number of elements in a map whose key matches the key specified in a parameter.
crbegin	Returns a const iterator that points to the first element in a reversed map.
crend	Returns a const iterator that points to the location after the last element in a reversed map.
emplace	Inserts an element constructed in place into the map.
emplace_hint	Inserts an element constructed in place into the map, with a placement hint.
empty	Returns `true` if a map is empty.
end	Returns the past-the-end iterator.
equal_range	Returns a pair of iterators. The first iterator in the pair points to the first element in a `map` with a key that is greater than a specified key. The second iterator in the pair points to the first element in the `map` with a key that is equal to or greater than the key.
erase	Removes an element or a range of elements in a map from the specified positions.
find	Returns an iterator that points to the location of an element in a map that has a key equal to a specified key.
get_allocator	Returns a copy of the `allocator` object that is used to construct the map.
insert	Inserts an element or a range of elements into the map at a specified position.
key_comp	Returns a copy of the comparison object that used to order keys in a map.

lower_bound	Returns an iterator to the first element in a map that has a key value that is equal to or greater than that of a specified key.
max_size	Returns the maximum length of the map.
rbegin	Returns an iterator that points to the first element in a reversed map.
rend	Returns an iterator that points to the location after the last element in a reversed map.
size	Returns the number of elements in the map.
swap	Exchanges the elements of two maps.
upper_bound	Returns an iterator to the first element in a map that has a key value that is greater than that of a specified key.
value_comp	Retrieves a copy of the comparison object that is used to order element values in a map.

Operators

operator[]	Inserts an element into a map with a specified key value.
operator=	Replaces the elements of a map with a copy of another map.

Requirements

Header: <map>

Namespace: std

map::allocator_type

A type that represents the allocator class for the map object.

```
typedef Allocator allocator_type;
```

Example

See example for get_allocator for an example that uses `allocator_type`.

map::at

Finds an element with a specified key value.

```
Type& at(const Key& key);

const Type& at(const Key& key) const;
```

Parameters

Parameter	Description
key	The key value to find.

Return Value

A reference to the data value of the element found.

Remarks

If the argument key value is not found, then the function throws an object of class out_of_range Class.

Example

```
// map_at.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

typedef std::map<char, int> Mymap;
int main()
    {
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

// find and show elements
    std::cout << "c1.at('a') == " << c1.at('a') << std::endl;
    std::cout << "c1.at('b') == " << c1.at('b') << std::endl;
    std::cout << "c1.at('c') == " << c1.at('c') << std::endl;

    return (0);
```

```
    }
```

map::begin

Returns an iterator addressing the first element in the map.

```
const_iterator begin() const;

iterator begin();
```

Return Value

A bidirectional iterator addressing the first element in the map or the location succeeding an empty map.

Example

```
// map_begin.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map <int, int> m1;

   map <int, int> :: iterator m1_Iter;
   map <int, int> :: const_iterator m1_cIter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 0, 0 ) );
   m1.insert ( Int_Pair ( 1, 1 ) );
   m1.insert ( Int_Pair ( 2, 4 ) );

   m1_cIter = m1.begin ( );
   cout << "The first element of m1 is " << m1_cIter -> first << endl;

   m1_Iter = m1.begin ( );
   m1.erase ( m1_Iter );

   // The following 2 lines would err because the iterator is const
   // m1_cIter = m1.begin ( );
   // m1.erase ( m1_cIter );

   m1_cIter = m1.begin( );
   cout << "The first element of m1 is now " << m1_cIter -> first << endl;
```

```
}
```

Output

```
The first element of m1 is 0
The first element of m1 is now 1
```

map::cbegin

Returns a `const` iterator that addresses the location just beyond the last element in a range.

```
const_iterator cbegin() const;
```

Return Value

A `const` bidirectional iterator addressing the first element in the range, or the location just beyond the end of an empty range (for an empty range, `cbegin() == cend()`).

Remarks

With the return value of `cbegin`, the elements in the range cannot be modified.

You can use this member function in place of the `begin()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the auto type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- `const`) container of any kind that supports `begin()` and `cbegin()`.

C++

```
auto i1 = Container.begin();
// i1 is Container<T>::iterator
auto i2 = Container.cbegin();

// i2 is Container<T>::const_iterator
```

map::cend

Returns a `const` iterator that addresses the location just beyond the last element in a range.

```
const_iterator cend() const;
```

Return Value

A `const` bidirectional-access iterator that points just beyond the end of the range.

Remarks

cend is used to test whether an iterator has passed the end of its range.

You can use this member function in place of the `end()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the auto type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- `const`) container of any kind that supports `end()` and `cend()`.

C++

```
auto i1 = Container.end();
// i1 is Container<T>::iterator
auto i2 = Container.cend();

// i2 is Container<T>::const_iterator
```

The value returned by cend should not be dereferenced.

map::clear

Erases all the elements of a map.

```
void clear();
```

Example

The following example demonstrates the use of the map::clear member function.

```
// map_clear.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main()
{
```

```
    using namespace std;
    map<int, int> m1;
    map<int, int>::size_type i;
    typedef pair<int, int> Int_Pair;

    m1.insert(Int_Pair(1, 1));
    m1.insert(Int_Pair(2, 4));

    i = m1.size();
    cout << "The size of the map is initially "
         << i << "." << endl;

    m1.clear();
    i = m1.size();
    cout << "The size of the map after clearing is "
         << i << "." << endl;
}
```

Output

```
The size of the map is initially 2.
The size of the map after clearing is 0.
```

map::const_iterator

A type that provides a bidirectional iterator that can read a **const** element in the map.

```
typedef implementation-defined const_iterator;
```

Remarks

A type `const_iterator` cannot be used to modify the value of an element.

The `const_iterator` defined by map points to elements that are objects of value_type, that is of type `pair<` **constKey**, **Type**>, whose first member is the key to the element and whose second member is the mapped datum held by the element.

To dereference a `const_iterator``cIter` pointing to an element in a map, use the **->** operator.

To access the value of the key for the element, use `cIter` -> **first**, which is equivalent to (* `cIter`). **first**.

To access the value of the mapped datum for the element, use `cIter` -> **second**, which is equivalent to (* `cIter`). **second**.

Example

See example for begin for an example that uses `const_iterator`.

map::const_pointer

A type that provides a pointer to a **const** element in a map.

```
typedef typename allocator_type::const_pointer const_pointer;
```

Remarks

A type `const_pointer` cannot be used to modify the value of an element.

In most cases, an iterator should be used to access the elements in a map object.

map::const_reference

A type that provides a reference to a **const** element stored in a map for reading and performing **const** operations.

```
typedef typename allocator_type::const_reference const_reference;
```

Example

```
// map_const_ref.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map <int, int> m1;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );

   // Declare and initialize a const_reference &Ref1
   // to the key of the first element
   const int &Ref1 = ( m1.begin( ) -> first );
```

```
    // The following line would cause an error as the
    // non-const_reference cannot be used to access the key
    // int &Ref1 = ( m1.begin( ) -> first );

    cout << "The key of first element in the map is "
         << Ref1 << "." << endl;

    // Declare and initialize a reference &Ref2
    // to the data value of the first element
    int &Ref2 = ( m1.begin( ) -> second );

    cout << "The data value of first element in the map is "
         << Ref2 << "." << endl;
}
```

Output

```
The key of first element in the map is 1.
The data value of first element in the map is 10.
```

map::const_reverse_iterator

A type that provides a bidirectional iterator that can read any **const** element in the map.

```
typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
```

Remarks

A type `const_reverse_iterator` cannot modify the value of an element and is use to iterate through the map in reverse.

The `const_reverse_iterator` defined by map points to elements that are objects of value_type, that is of type `pair<`**constKey**, **Type**>, whose first member is the key to the element and whose second member is the mapped datum held by the element.

To dereference a `const_reverse_iterator``crIter` pointing to an element in a map, use the **->** operator.

To access the value of the key for the element, use `crIter` -> **first**, which is equivalent to (* `crIter`). **first**.

To access the value of the mapped datum for the element, use `crIter` -> **second**, which is equivalent to (* `crIter`). **first**.

Example

See the example for rend for an example of how to declare and use `const_reverse_iterator`.

map::count

Returns the number of elements in a map whose key matches a parameter-specified key.

```
size_type count(const Key& key) const;
```

Parameters

key
The key value of the elements to be matched from the map.

Return Value

1 if the map contains an element whose sort key matches the parameter key; 0 if the map does not contain an element with a matching key.

Remarks

The member function returns the number of elements *x* in the range

[`lower_bound` (_ *Key*), `upper_bound` (_ *Key*))

which is 0 or 1 in the case of map, which is a unique associative container.

Example

The following example demonstrates the use of the map::count member function.

```
// map_count.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main()
{
    using namespace std;
    map<int, int> m1;
    map<int, int>::size_type i;
    typedef pair<int, int> Int_Pair;

    m1.insert(Int_Pair(1, 1));
    m1.insert(Int_Pair(2, 1));
    m1.insert(Int_Pair(1, 4));
    m1.insert(Int_Pair(2, 1));

    // Keys must be unique in map, so duplicates are ignored
    i = m1.count(1);
    cout << "The number of elements in m1 with a sort key of 1 is: "
```

```
          << i << "." << endl;

    i = m1.count(2);
    cout << "The number of elements in m1 with a sort key of 2 is: "
          << i << "." << endl;

    i = m1.count(3);
    cout << "The number of elements in m1 with a sort key of 3 is: "
          << i << "." << endl;
}
```

Output

```
The number of elements in m1 with a sort key of 1 is: 1.
The number of elements in m1 with a sort key of 2 is: 1.
The number of elements in m1 with a sort key of 3 is: 0.
```

map::crbegin

Returns a const iterator addressing the first element in a reversed map.

```
const_reverse_iterator crbegin() const;
```

Return Value

A const reverse bidirectional iterator addressing the first element in a reversed map or addressing what had been the last element in the unreversed map.

Remarks

`crbegin` is used with a reversed `map` just as begin is used with a `map`.

With the return value of `crbegin`, the `map` object cannot be modified

`crbegin` can be used to iterate through a `map` backwards.

Example

```
// map_crbegin.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
```

```
{
   using namespace std;
   map <int, int> m1;

   map <int, int> :: const_reverse_iterator m1_crIter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 3, 30 ) );

   m1_crIter = m1.crbegin( );
   cout << "The first element of the reversed map m1 is "
        << m1_crIter -> first << "." << endl;
}
```

Output

```
The first element of the reversed map m1 is 3.
```

map::crend

Returns a const iterator that addresses the location succeeding the last element in a reversed map.

```
const_reverse_iterator crend() const;
```

Return Value

A const reverse bidirectional iterator that addresses the location succeeding the last element in a reversed map (the location that had preceded the first element in the unreversed map).

Remarks

crend is used with a reversed map just as end is used with a map.

With the return value of crend, the map object cannot be modified.

crend can be used to test to whether a reverse iterator has reached the end of its map.

The value returned by crend should not be dereferenced.

Example

```
// map_crend.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map <int, int> m1;

   map <int, int> :: const_reverse_iterator m1_crIter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 3, 30 ) );

   m1_crIter = m1.crend( );
   m1_crIter--;
   cout << "The last element of the reversed map m1 is "
        << m1_crIter -> first << "." << endl;
}
```

Output

```
The last element of the reversed map m1 is 1.
```

map::difference_type

A signed integer type that can be used to represent the number of elements of a map in a range between elements pointed to by iterators.

```
typedef allocator_type::difference_type difference_type;
```

Remarks

The `difference_type` is the type returned when subtracting or incrementing through iterators of the container. The `difference_type` is typically used to represent the number of elements in the range *[first, last)* between the iterators `first` and `last`, includes the element pointed to by `first` and the range of elements up to, but not including, the element pointed to by `last`.

Note that although `difference_type` is available for all iterators that satisfy the requirements of an input iterator, which includes the class of bidirectional iterators supported by reversible containers such as set, subtraction between iterators is only supported by random access iterators provided by a random access container such as vector.

Example

```
// map_diff_type.cpp
// compile with: /EHsc
#include <iostream>
#include <map>
#include <algorithm>

int main( )
{
   using namespace std;
   map <int, int> m1;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 3, 20 ) );
   m1.insert ( Int_Pair ( 2, 30 ) );

   map <int, int>::iterator m1_Iter, m1_bIter, m1_eIter;
   m1_bIter = m1.begin( );
   m1_eIter = m1.end( );

   // Count the number of elements in a map
   map <int, int>::difference_type  df_count = 1;
   m1_Iter = m1.begin( );
   while ( m1_Iter != m1_eIter)
   {
      df_count++;
      m1_Iter++;
   }

   cout << "The number of elements in the map m1 is: "
        << df_count << "." << endl;
}
```

Output

```
The number of elements in the map m1 is: 4.
```

map::emplace

Inserts an element constructed in place (no copy or move operations are performed) into a map.

```
template <class... Args>
pair<iterator, bool>
emplace(
    Args&&... args);
```

Parameters

Parameter	Description
`args`	The arguments forwarded to construct an element to be inserted into the map unless it already contains an element whose value is equivalently ordered.

Return Value

A pair whose `bool` component is true if an insertion was made, and false if the map already contained an element of equivalent value in the ordering. The iterator component of the return-value pair points to the newly inserted element if the `bool` component is true, or to the existing element if the `bool` component is false.

To access the iterator component of a `pair``pr`, use `pr.first`; to dereference it, use `*pr.first`. To access the `bool` component, use `pr.second`. For an example, see the sample code later in this article.

Remarks

No iterators or references are invalidated by this function.

During emplacement, if an exception is thrown, the container's state is not modified.

The value_type of an element is a pair, so that the value of an element will be an ordered pair with the first component equal to the key value and the second component equal to the data value of the element.

Example

C++

```
// map_emplace.cpp
// compile with: /EHsc
#include <map>
#include <string>
#include <iostream>

using namespace std;

template <typename M> void print(const M& m) {
    cout << m.size() << " elements: ";

    for (const auto& p : m) {
        cout << "(" << p.first << ", " << p.second << ") ";
    }
```

```
    cout << endl;
}

int main()
{
    map<int, string> m1;

    auto ret = m1.emplace(10, "ten");

    if (!ret.second){
        auto pr = *ret.first;
        cout << "Emplace failed, element with key 10 already exists."
            << endl << "  The existing element is (" << pr.first << ", " <<
pr.second << ")"
            << endl;
        cout << "map not modified" << endl;
    }
    else{
        cout << "map modified, now contains ";
        print(m1);
    }
    cout << endl;

    ret = m1.emplace(10, "one zero");

    if (!ret.second){
        auto pr = *ret.first;
        cout << "Emplace failed, element with key 10 already exists."
            << endl << "  The existing element is (" << pr.first << ", " <<
pr.second << ")"
            << endl;
    }
    else{
        cout << "map modified, now contains ";
        print(m1);
    }
    cout << endl;
}
```

map::emplace_hint

Inserts an element constructed in place (no copy or move operations are performed), with a placement hint.

```
template <class... Args>
iterator emplace_hint(
    const_iterator where,
    Args&&... args);
```

Parameters

Parameter	Description
`args`	The arguments forwarded to construct an element to be inserted into the map unless the map already contains that element or, more generally, unless it already contains an element whose key is equivalently ordered.
`where`	The place to start searching for the correct point of insertion. (If that point immediately precedes `where`, insertion can occur in amortized constant time instead of logarithmic time.)

Return Value

An iterator to the newly inserted element.

If the insertion failed because the element already exists, returns an iterator to the existing element with its key.

Remarks

No iterators or references are invalidated by this function.

During emplacement, if an exception is thrown, the container's state is not modified.

The value_type of an element is a pair, so that the value of an element will be an ordered pair with the first component equal to the key value and the second component equal to the data value of the element.

Example

C++

```
// map_emplace.cpp
// compile with: /EHsc
#include <map>
#include <string>
#include <iostream>

using namespace std;

template <typename M> void print(const M& m) {
    cout << m.size() << " elements: " << endl;

    for (const auto& p : m) {
        cout << "(" << p.first <<  "," << p.second << ") ";
    }

    cout << endl;
}

int main()
{
```

```
    map<string, string> m1;

    // Emplace some test data
    m1.emplace("Anna", "Accounting");
    m1.emplace("Bob", "Accounting");
    m1.emplace("Carmine", "Engineering");

    cout << "map starting data: ";
    print(m1);
    cout << endl;

    // Emplace with hint
    // m1.end() should be the "next" element after this emplacement
    m1.emplace_hint(m1.end(), "Doug", "Engineering");

    cout << "map modified, now contains ";
    print(m1);
    cout << endl;
}
```

map::empty

Tests if a map is empty.

```
bool empty() const;
```

Return Value

true if the map is empty; **false** if the map is nonempty.

Example

```
// map_empty.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map <int, int> m1, m2;

   typedef pair <int, int> Int_Pair;
   m1.insert ( Int_Pair ( 1, 1 ) );
```

```
    if ( m1.empty( ) )
       cout << "The map m1 is empty." << endl;
    else
       cout << "The map m1 is not empty." << endl;

    if ( m2.empty( ) )
       cout << "The map m2 is empty." << endl;
    else
       cout << "The map m2 is not empty." << endl;
}
```

Output

```
The map m1 is not empty.
The map m2 is empty.
```

map::end

Returns the past-the-end iterator.

```
const_iterator end() const;


iterator end();
```

Return Value

The past-the-end iterator. If the map is empty, then `map::end() == map::begin()`.

Remarks

end is used to test whether an iterator has passed the end of its map.

The value returned by **end** should not be dereferenced.

For a code example, see map::find.

map::equal_range

Returns a pair of iterators that represent the lower_bound of the key and the upper_bound of the key.

```
pair <const_iterator, const_iterator> equal_range (const Key& key) const;

pair <iterator, iterator> equal_range (const Key& key);
```

Parameters

key
The argument key value to be compared with the sort key of an element from the map being searched.

Return Value

To access the first iterator of a pair `pr` returned by the member function, use `pr.` **first**, and to dereference the lower bound iterator, use *(`pr.` **first**). To access the second iterator of a pair `pr` returned by the member function, use `pr.` **second**, and to dereference the upper bound iterator, use *(`pr.` **second**).

Example

```
// map_equal_range.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   typedef map <int, int, less<int> > IntMap;
   IntMap m1;
   map <int, int> :: const_iterator m1_RcIter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 3, 30 ) );

   pair <IntMap::const_iterator, IntMap::const_iterator> p1, p2;
   p1 = m1.equal_range( 2 );

   cout << "The lower bound of the element with "
        << "a key of 2 in the map m1 is: "
        << p1.first -> second << "." << endl;

   cout << "The upper bound of the element with "
        << "a key of 2 in the map m1 is: "
        << p1.second -> second << "." << endl;

   // Compare the upper_bound called directly
   m1_RcIter = m1.upper_bound( 2 );

   cout << "A direct call of upper_bound( 2 ) gives "
        << m1_RcIter -> second << "," << endl
        << " matching the 2nd element of the pair"
```

```
         << " returned by equal_range( 2 )." << endl;

   p2 = m1.equal_range( 4 );

   // If no match is found for the key,
   // both elements of the pair return end( )
   if ( ( p2.first == m1.end( ) ) && ( p2.second == m1.end( ) ) )
      cout << "The map m1 doesn't have an element "
           << "with a key less than 40." << endl;
   else
      cout << "The element of map m1 with a key >= 40 is: "
           << p2.first -> first << "." << endl;
}
```

Output

```
The lower bound of the element with a key of 2 in the map m1 is: 20.
The upper bound of the element with a key of 2 in the map m1 is: 30.
A direct call of upper_bound( 2 ) gives 30,
 matching the 2nd element of the pair returned by equal_range( 2 ).
The map m1 doesn't have an element with a key less than 40.
```

map::erase

Removes an element or a range of elements in a map from specified positions or removes elements that match a specified key.

```
iterator erase(
    const_iterator Where);


iterator erase(
    const_iterator First,
    const_iterator Last);


size_type erase(
    const key_type& Key);
```

Parameters

`Where`
Position of the element to be removed.

`First`
Position of the first element to be removed.

`Last`
Position just beyond the last element to be removed.

`Key`
The key value of the elements to be removed.

Return Value

For the first two member functions, a bidirectional iterator that designates the first element remaining beyond any elements removed, or an element that is the end of the map if no such element exists.

For the third member function, returns the number of elements that have been removed from the map.

Example

C++

```
// map_erase.cpp
// compile with: /EHsc
#include <map>
#include <string>
#include <iostream>
#include <iterator> // next() and prev() helper functions
#include <utility>  // make_pair()

using namespace std;

using mymap = map<int, string>;

void printmap(const mymap& m) {
    for (const auto& elem : m) {
        cout << " [" << elem.first << ", " << elem.second << "]";
    }
    cout << endl << "size() == " << m.size() << endl << endl;
}

int main()
{
    mymap m1;

    // Fill in some data to test with, one at a time
    m1.insert(make_pair(1, "A"));
    m1.insert(make_pair(2, "B"));
    m1.insert(make_pair(3, "C"));
    m1.insert(make_pair(4, "D"));
    m1.insert(make_pair(5, "E"));

    cout << "Starting data of map m1 is:" << endl;
    printmap(m1);
    // The 1st member function removes an element at a given position
    m1.erase(next(m1.begin()));
    cout << "After the 2nd element is deleted, the map m1 is:" << endl;
    printmap(m1);

    // Fill in some data to test with, one at a time, using an intializer list
```

```
    mymap m2
    {
        { 10, "Bob" },
        { 11, "Rob" },
        { 12, "Robert" },
        { 13, "Bert" },
        { 14, "Bobby" }
    };

    cout << "Starting data of map m2 is:" << endl;
    printmap(m2);
    // The 2nd member function removes elements
    // in the range [First, Last)
    m2.erase(next(m2.begin()), prev(m2.end()));
    cout << "After the middle elements are deleted, the map m2 is:" << endl;
    printmap(m2);

    mymap m3;

    // Fill in some data to test with, one at a time, using emplace
    m3.emplace(1, "red");
    m3.emplace(2, "yellow");
    m3.emplace(3, "blue");
    m3.emplace(4, "green");
    m3.emplace(5, "orange");
    m3.emplace(6, "purple");
    m3.emplace(7, "pink");

    cout << "Starting data of map m3 is:" << endl;
    printmap(m3);
    // The 3rd member function removes elements with a given Key
    mymap::size_type count = m3.erase(2);
    // The 3rd member function also returns the number of elements removed
    cout << "The number of elements removed from m3 is: " << count << "." <<
endl;
    cout << "After the element with a key of 2 is deleted, the map m3 is:" <<
endl;
    printmap(m3);
}
```

map::find

Returns an iterator that refers to the location of an element in a map that has a key equivalent to a specified key.

```
iterator find(const Key& key);
```

```
const_iterator find(const Key& key) const;
```

Parameters

key
The key value to be matched by the sort key of an element from the map being searched.

Return Value

An iterator that refers to the location of an element with a specified key, or the location succeeding the last element in the map (`map::end()`) if no match is found for the key.

Remarks

The member function returns an iterator that refers to an element in the map whose sort key is equivalent to the argument key under a binary predicate that induces an ordering based on a less than comparability relation.

If the return value of **find** is assigned to a **const_iterator**, the map object cannot be modified. If the return value of **find** is assigned to an **iterator**, the map object can be modified

Example

C++

```
// compile with: /EHsc /W4 /MTd
#include <map>
#include <iostream>
#include <vector>
#include <string>
#include <utility>  // make_pair()

using namespace std;

template <typename A, typename B> void print_elem(const pair<A, B>& p) {
    cout << "(" << p.first << ", " << p.second << ") ";
}

template <typename T> void print_collection(const T& t) {
    cout << t.size() << " elements: ";

    for (const auto& p : t) {
        print_elem(p);
    }
    cout << endl;
}

template <typename C, class T> void findit(const C& c, T val) {
    cout << "Trying find() on value " << val << endl;
    auto result = c.find(val);
    if (result != c.end()) {
        cout << "Element found: "; print_elem(*result); cout << endl;
    } else {
        cout << "Element not found." << endl;
    }
```

```
}

int main()
{
    map<int, string> m1({ { 40, "Zr" }, { 45, "Rh" } });
    cout << "The starting map m1 is (key, value):" << endl;
    print_collection(m1);

    vector<pair<int, string>> v;
    v.push_back(make_pair(43, "Tc"));
    v.push_back(make_pair(41, "Nb"));
    v.push_back(make_pair(46, "Pd"));
    v.push_back(make_pair(42, "Mo"));
    v.push_back(make_pair(44, "Ru"));
    v.push_back(make_pair(44, "Ru")); // attempt a duplicate

    cout << "Inserting the following vector data into m1:" << endl;
    print_collection(v);

    m1.insert(v.begin(), v.end());

    cout << "The modified map m1 is (key, value):" << endl;
    print_collection(m1);
    cout << endl;
    findit(m1, 45);
    findit(m1, 6);
}
```

map::get_allocator

Returns a copy of the allocator object used to construct the map.

```
allocator_type get_allocator() const;
```

Return Value

The allocator used by the map.

Remarks

Allocators for the map class specify how the class manages storage. The default allocators supplied with STL container classes is sufficient for most programming needs. Writing and using your own allocator class is an advanced C++ topic.

Example

```
// map_get_allocator.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map <int, int>::allocator_type m1_Alloc;
   map <int, int>::allocator_type m2_Alloc;
   map <int, double>::allocator_type m3_Alloc;
   map <int, int>::allocator_type m4_Alloc;

   // The following lines declare objects
   // that use the default allocator.
   map <int, int> m1;
   map <int, int, allocator<int> > m2;
   map <int, double, allocator<double> > m3;

   m1_Alloc = m1.get_allocator( );
   m2_Alloc = m2.get_allocator( );
   m3_Alloc = m3.get_allocator( );

   cout << "The number of integers that can be allocated\n"
        << "before free memory is exhausted: "
        << m2.max_size( ) << ".\n" << endl;

   cout << "The number of doubles that can be allocated\n"
        << "before free memory is exhausted: "
        << m3.max_size( ) <<  ".\n" << endl;

   // The following line creates a map m4
   // with the allocator of map m1.
   map <int, int> m4( less<int>( ), m1_Alloc );

   m4_Alloc = m4.get_allocator( );

   // Two allocators are interchangeable if
   // storage allocated from each can be
   // deallocated with the other
   if( m1_Alloc == m4_Alloc )
   {
      cout << "The allocators are interchangeable." << endl;
   }
   else
   {
      cout << "The allocators are not interchangeable." << endl;
   }
}
```

map::insert

Inserts an element or a range of elements into a map.

```
// (1) single element
pair<iterator, bool> insert(
    const value_type& Val);


// (2) single element, perfect forwarded
template <class ValTy>
pair<iterator, bool>
insert(
    ValTy&& Val);


// (3) single element with hint
iterator insert(
    const_iterator Where,
    const value_type& Val);


// (4) single element, perfect forwarded, with hint
template <class ValTy>
iterator insert(
    const_iterator Where,
    ValTy&& Val);


// (5) range
template <class InputIterator>
void insert(
    InputIterator First,
    InputIterator Last);


// (6) initializer list
void insert(
    initializer_list<value_type>
IList);
```

Parameters

Parameter	Description
`Val`	The value of an element to be inserted into the map unless it already contains an element whose key is equivalently ordered.
`Where`	

	The place to start searching for the correct point of insertion. (If that point immediately precedes `Where`, insertion can occur in amortized constant time instead of logarithmic time.)
`ValTy`	Template parameter that specifies the argument type that the map can use to construct an element of value_type, and perfect-forwards `Val` as an argument.
`First`	The position of the first element to be copied.
`Last`	The position just beyond the last element to be copied.
`InputIterator`	Template function argument that meets the requirements of an input iterator that points to elements of a type that can be used to construct value_type objects.
`IList`	The initializer_list from which to copy the elements.

Return Value

The single-element member functions, (1) and (2), return a pair whose `bool` component is true if an insertion was made, and false if the map already contained an element whose key had an equivalent value in the ordering. The iterator component of the return-value pair points to the newly inserted element if the `bool` component is true, or to the existing element if the `bool` component is false.

The single-element-with-hint member functions, (3) and (4), return an iterator that points to the position where the new element was inserted into the map or, if an element with an equivalent key already exists, to the existing element.

Remarks

No iterators, pointers, or references are invalidated by this function.

During the insertion of just one element, if an exception is thrown, the container's state is not modified. During the insertion of multiple elements, if an exception is thrown, the container is left in an unspecified but valid state.

To access the iterator component of a `pair``pr` that's returned by the single-element member functions, use `pr.first`; to dereference the iterator within the returned pair, use `*pr.first`, giving you an element. To access the `bool` component, use `pr.second`. For an example, see the sample code later in this article.

The value_type of a container is a typedef that belongs to the container, and for map, `map<K, V>::value_type` is `pair<const K, V>`. The value of an element is an ordered pair in which the first component is equal to the key value and the second component is equal to the data value of the element.

The range member function (5) inserts the sequence of element values into a map that corresponds to each element addressed by an iterator in the range `[First, Last)`; therefore, `Last` does not get inserted. The container member function `end()` refers to the position just after the last element in the container—for example, the statement `m.insert(v.begin(), v.end());` attempts to insert all elements of `v` into `m`. Only elements that have unique values in the range are inserted; duplicates are ignored. To observe which elements are rejected, use the single-element versions of `insert`.

The initializer list member function (6) uses an initializer_list to copy elements into the map.

For insertion of an element constructed in place—that is, no copy or move operations are performed—see map::emplace and map::emplace_hint.

Example

C++

```
// map_insert.cpp
// compile with: /EHsc
#include <map>
#include <iostream>
#include <string>
#include <vector>
#include <utility>  // make_pair()

using namespace std;

template <typename M> void print(const M& m) {
    cout << m.size() << " elements: ";

    for (const auto& p : m) {
        cout << "(" << p.first << ", " << p.second << ") ";
    }

    cout << endl;
}

int main()
{

    // insert single values
    map<int, int> m1;
    // call insert(const value_type&) version
    m1.insert({ 1, 10 });
    // call insert(ValTy&&) version
    m1.insert(make_pair(2, 20));

    cout << "The original key and mapped values of m1 are:" << endl;
    print(m1);

    // intentionally attempt a duplicate, single element
    auto ret = m1.insert(make_pair(1, 111));
    if (!ret.second){
        auto pr = *ret.first;
        cout << "Insert failed, element with key value 1 already exists."
            << endl << "  The existing element is (" << pr.first << ", " <<
pr.second << ")"
            << endl;
    }
    else{
        cout << "The modified key and mapped values of m1 are:" << endl;
        print(m1);
    }
    cout << endl;
```

```
    // single element, with hint
    m1.insert(m1.end(), make_pair(3, 30));
    cout << "The modified key and mapped values of m1 are:" << endl;
    print(m1);
    cout << endl;

    // The templatized version inserting a jumbled range
    map<int, int> m2;
    vector<pair<int, int>> v;
    v.push_back(make_pair(43, 294));
    v.push_back(make_pair(41, 262));
    v.push_back(make_pair(45, 330));
    v.push_back(make_pair(42, 277));
    v.push_back(make_pair(44, 311));

    cout << "Inserting the following vector data into m2:" << endl;
    print(v);

    m2.insert(v.begin(), v.end());

    cout << "The modified key and mapped values of m2 are:" << endl;
    print(m2);
    cout << endl;

    // The templatized versions move-constructing elements
    map<int, string>  m3;
    pair<int, string> ip1(475, "blue"), ip2(510, "green");

    // single element
    m3.insert(move(ip1));
    cout << "After the first move insertion, m3 contains:" << endl;
    print(m3);

    // single element with hint
    m3.insert(m3.end(), move(ip2));
    cout << "After the second move insertion, m3 contains:" << endl;
    print(m3);
    cout << endl;

    map<int, int> m4;
    // Insert the elements from an initializer_list
    m4.insert({ { 4, 44 }, { 2, 22 }, { 3, 33 }, { 1, 11 }, { 5, 55 } });
    cout << "After initializer_list insertion, m4 contains:" << endl;
    print(m4);
    cout << endl;
}
```

map::iterator

A type that provides a bidirectional iterator that can read or modify any element in a map.

```
typedef implementation-defined iterator;
```

Remarks

The **iterator** defined by map points to elements that are objects of value_type, that is of type `pair<`**constKey**, **Type**>, whose first member is the key to the element and whose second member is the mapped datum held by the element.

To dereference an **iterator**`Iter` pointing to an element in a map, use the **->** operator.

To access the value of the key for the element, use `Iter ->` **first**, which is equivalent to (* `Iter`). **first**. To access the value of the mapped datum for the element, use `Iter ->` **second**, which is equivalent to (* `Iter`). **second**.

Example

See example for begin for an example of how to declare and use **iterator**.

map::key_comp

Retrieves a copy of the comparison object used to order keys in a map.

```
key_compare key_comp() const;
```

Return Value

Returns the function object that a map uses to order its elements.

Remarks

The stored object defines the member function

bool operator(**constKey&**`left`, **const Key&**`right`);

which returns **true** if `left` precedes and is not equal to `right` in the sort order.

Example

```
// map_key_comp.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
```

```
{
   using namespace std;

   map <int, int, less<int> > m1;
   map <int, int, less<int> >::key_compare kc1 = m1.key_comp( ) ;
   bool result1 = kc1( 2, 3 ) ;
   if( result1 == true )
   {
      cout << "kc1( 2,3 ) returns value of true, "
           << "where kc1 is the function object of m1."
           << endl;
   }
   else
   {
      cout << "kc1( 2,3 ) returns value of false "
           << "where kc1 is the function object of m1."
           << endl;
   }

   map <int, int, greater<int> > m2;
   map <int, int, greater<int> >::key_compare kc2 = m2.key_comp( );
   bool result2 = kc2( 2, 3 ) ;
   if( result2 == true )
   {
      cout << "kc2( 2,3 ) returns value of true, "
           << "where kc2 is the function object of m2."
           << endl;
   }
   else
   {
      cout << "kc2( 2,3 ) returns value of false, "
           << "where kc2 is the function object of m2."
           << endl;
   }
}
```

Output

```
kc1( 2,3 ) returns value of true, where kc1 is the function object of m1.
kc2( 2,3 ) returns value of false, where kc2 is the function object of m2.
```

map::key_compare

A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the map.

```
typedef Traits key_compare;
```

Remarks

`key_compare` is a synonym for the template parameter `Traits`.

For more information on `Traits` see the map Class topic.

Example

See example for key_comp for an example of how to declare and use `key_compare`.

map::key_type

A type that describes the sort key stored in each element of the map.

```
typedef Key key_type;
```

Remarks

`key_type` is a synonym for the template parameter `Key`.

For more information on `Key`, see the Remarks section of the map Class topic.

Example

See example for value_type for an example of how to declare and use `key_type`.

map::lower_bound

Returns an iterator to the first element in a map with a key value that is equal to or greater than that of a specified key.

```
iterator lower_bound(const Key& key);

const_iterator lower_bound(const Key& key) const;
```

Parameters

key
The argument key value to be compared with the sort key of an element from the map being searched.

Return Value

An **iterator** or `const_iterator` that addresses the location of an element in a map that with a key that is equal to or greater than the argument key, or that addresses the location succeeding the last element in the map if no match is found for the key.

If the return value of `lower_bound` is assigned to a `const_iterator`, the map object cannot be modified. If the return value of `lower_bound` is assigned to an **iterator**, the map object can be modified.

Example

```
// map_lower_bound.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map <int, int> m1;
   map <int, int> :: const_iterator m1_AcIter, m1_RcIter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 3, 30 ) );

   m1_RcIter = m1.lower_bound( 2 );
   cout << "The first element of map m1 with a key of 2 is: "
        << m1_RcIter -> second << "." << endl;

   // If no match is found for this key, end( ) is returned
   m1_RcIter = m1. lower_bound ( 4 );

   if ( m1_RcIter == m1.end( ) )
      cout << "The map m1 doesn't have an element "
           << "with a key of 4." << endl;
   else
      cout << "The element of map m1 with a key of 4 is: "
           << m1_RcIter -> second << "." << endl;

   // The element at a specific location in the map can be found
   // using a dereferenced iterator addressing the location
   m1_AcIter = m1.end( );
   m1_AcIter--;
   m1_RcIter = m1. lower_bound ( m1_AcIter -> first );
   cout << "The element of m1 with a key matching "
        << "that of the last element is: "
        << m1_RcIter -> second << "." << endl;
}
```

Output

```
The first element of map m1 with a key of 2 is: 20.
The map m1 doesn't have an element with a key of 4.
The element of m1 with a key matching that of the last element is: 30.
```

map::map

Constructs a map that is empty or that is a copy of all or part of some other map.

```
map();

explicit map(
    const Traits& Comp);

map(
    const Traits& Comp,
    const Allocator& Al);

map(
    const map& Right);

map(
    map&& Right);

map(
    initializer_list<value_type> IList);

map(
    initializer_list<value_type> IList,
    const Traits& Comp);

map(
    initializer_list<value_type> IList,
    const Traits& Comp,
    const Allocator& Allocator);

template <class InputIterator>
map(
 InputIterator First,
    InputIterator Last);

template <class InputIterator>
map(
 InputIterator First,
    InputIterator Last,
    const Traits& Comp);

template <class InputIterator>
map(
```

```
   InputIterator First,
      InputIterator Last,
      const Traits& Comp,
      const Allocator& Al);
```

Parameters

Parameter	Description
`Al`	The storage allocator class to be used for this map object, which defaults to `Allocator`.
`Comp`	The comparison function of type `const Traits` used to order the elements in the map, which defaults to `hash_compare`.
`Right`	The map of which the constructed set is to be a copy.
`First`	The position of the first element in the range of elements to be copied.
`Last`	The position of the first element beyond the range of elements to be copied.
`IList`	The initializer_list from which the elements are to be copied.

Remarks

All constructors store a type of allocator object that manages memory storage for the map and that can later be returned by calling get_allocator. The allocator parameter is often omitted in the class declarations and preprocessing macros used to substitute alternative allocators.

All constructors initialize their map.

All constructors store a function object of type Traits that is used to establish an order among the keys of the map and that can later be returned by calling key_comp.

The first three constructors specify an empty initial map, the second specifying the type of comparison function (`Comp`) to be used in establishing the order of the elements and the third explicitly specifying the allocator type (`Al`) to be used. The key word `explicit` suppresses certain kinds of automatic type conversion.

The fourth constructor specifies a copy of the map `Right`.

The fifth constructor specifies a copy of the map by moving `Right`.

The sixth, seventh, and eighth constructors use an initializer_list from which to copy the members.

The next three constructors copy the range `[First, Last)` of a map with increasing explicitness in specifying the type of comparison function of class `Traits` and allocator.

Example

```
// map_map.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main()
{
    using namespace std;
    typedef pair <int, int> Int_Pair;
    map <int, int>::iterator m1_Iter, m3_Iter, m4_Iter, m5_Iter, m6_Iter,
m7_Iter;
    map <int, int, less<int> >::iterator m2_Iter;

    // Create an empty map m0 of key type integer
    map <int, int> m0;

    // Create an empty map m1 with the key comparison
    // function of less than, then insert 4 elements
    map <int, int, less<int> > m1;
    m1.insert(Int_Pair(1, 10));
    m1.insert(Int_Pair(2, 20));
    m1.insert(Int_Pair(3, 30));
    m1.insert(Int_Pair(4, 40));

    // Create an empty map m2 with the key comparison
    // function of geater than, then insert 2 elements
    map <int, int, less<int> > m2;
    m2.insert(Int_Pair(1, 10));
    m2.insert(Int_Pair(2, 20));

    // Create a map m3 with the
    // allocator of map m1
    map <int, int>::allocator_type m1_Alloc;
    m1_Alloc = m1.get_allocator();
    map <int, int> m3(less<int>(), m1_Alloc);
    m3.insert(Int_Pair(3, 30));

    // Create a copy, map m4, of map m1
    map <int, int> m4(m1);

    // Create a map m5 by copying the range m1[ first,  last)
    map <int, int>::const_iterator m1_bcIter, m1_ecIter;
    m1_bcIter = m1.begin();
    m1_ecIter = m1.begin();
    m1_ecIter++;
    m1_ecIter++;
    map <int, int> m5(m1_bcIter, m1_ecIter);

    // Create a map m6 by copying the range m4[ first,  last)
    // and with the allocator of map m2
    map <int, int>::allocator_type m2_Alloc;
    m2_Alloc = m2.get_allocator();
    map <int, int> m6(m4.begin(), ++m4.begin(), less<int>(), m2_Alloc);
```

```
    cout << "m1 =";
    for (auto i : m1)
        cout << i.first << " " << i.second << ", ";
    cout << endl;

    cout << "m2 =";
    for(auto i : m2)
        cout << i.first << " " << i.second << ", ";
    cout << endl;

    cout << "m3 =";
    for (auto i : m3)
        cout << i.first << " " << i.second << ", ";
    cout << endl;

    cout << "m4 =";
    for (auto i : m4)
        cout << i.first << " " << i.second << ", ";
    cout << endl;

    cout << "m5 =";
    for (auto i : m5)
        cout << i.first << " " << i.second << ", ";
    cout << endl;

    cout << "m6 =";
    for (auto i : m6)
        cout << i.first << " " << i.second << ", ";
    cout << endl;

    // Create a map m7 by moving m5
    cout << "m7 =";
    map<int, int> m7(move(m5));
    for (auto i : m7)
        cout << i.first << " " << i.second << ", ";
    cout << endl;

    // Create a map m8 by copying in an initializer_list
    map<int, int> m8{ { { 1, 1 }, { 2, 2 }, { 3, 3 }, { 4, 4 } } };
    cout << "m8: = ";
    for (auto i : m8)
        cout << i.first << " " << i.second << ", ";
    cout << endl;

    // Create a map m9 with an initializer_list and a comparator
    map<int, int> m9({ { 5, 5 }, { 6, 6 }, { 7, 7 }, { 8, 8 } }, less<int>());
    cout << "m9: = ";
    for (auto i : m9)
        cout << i.first << " " << i.second << ", ";
    cout << endl;

    // Create a map m10 with an initializer_list, a comparator, and an allocator
    map<int, int> m10({ { 9, 9 }, { 10, 10 }, { 11, 11 }, { 12, 12 } }, less<int>
(), m9.get_allocator());
    cout << "m10: = ";
```

```
    for (auto i : m10)
        cout << i.first << " " << i.second << ", ";
    cout << endl;
}
```

map::mapped_type

A type that represents the data stored in a map.

```
typedef Type mapped_type;
```

Remarks

The type `mapped_type` is a synonym for the class's `Type` template parameter.

For more information on `Type` see the map Class topic.

Example

See example for value_type for an example of how to declare and use `mapped_type`.

map::max_size

Returns the maximum length of the map.

```
size_type max_size() const;
```

Return Value

The maximum possible length of the map.

Example

```
// map_max_size.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
```

```
    using namespace std;
    map <int, int> m1;
    map <int, int> :: size_type i;

    i = m1.max_size( );
    cout << "The maximum possible length "
         << "of the map is " << i << "."
         << endl << "(Magnitude is machine specific.)";
}
```

map::operator[]

Inserts an element into a map with a specified key value.

```
Type& operator[](const Key& key);

Type& operator 0-(Key&& key);
```

Parameters

Parameter	Description
key	The key value of the element that is to be inserted.

Return Value

A reference to the data value of the inserted element.

Remarks

If the argument key value is not found, then it is inserted along with the default value of the data type.

`operator[]` may be used to insert elements into a map `m` using `m[ key] = DataValue;` where `DataValue` is the value of the `mapped_type` of the element with a key value of `key`.

When using `operator[]` to insert elements, the returned reference does not indicate whether an insertion is changing a pre-existing element or creating a new one. The member functions find and insert can be used to determine whether an element with a specified key is already present before an insertion.

Example

```
// map_op_insert.cpp
// compile with: /EHsc
```

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

int main( )
{
   using namespace std;
   typedef pair <const int, int> cInt2Int;
   map <int, int> m1;
   map <int, int> :: iterator pIter;

   // Insert a data value of 10 with a key of 1
   // into a map using the operator[] member function
   m1[ 1 ] = 10;

   // Compare other ways to insert objects into a map
   m1.insert ( map <int, int> :: value_type ( 2, 20 ) );
   m1.insert ( cInt2Int ( 3, 30 ) );

   cout  << "The keys of the mapped elements are:";
   for ( pIter = m1.begin( ) ; pIter != m1.end( ) ; pIter++ )
      cout << " " << pIter -> first;
   cout << "." << endl;

   cout  << "The values of the mapped elements are:";
   for ( pIter = m1.begin( ) ; pIter != m1.end( ) ; pIter++ )
      cout << " " << pIter -> second;
   cout << "." << endl;

   // If the key already exists, operator[]
   // changes the value of the datum in the element
   m1[ 2 ] = 40;

   // operator[] will also insert the value of the data
   // type's default constructor if the value is unspecified
   m1[5];

   cout  << "The keys of the mapped elements are now:";
   for ( pIter = m1.begin( ) ; pIter != m1.end( ) ; pIter++ )
      cout << " " << pIter -> first;
   cout << "." << endl;

   cout  << "The values of the mapped elements are now:";
   for ( pIter = m1.begin( ) ; pIter != m1.end( ) ; pIter++ )
      cout << " " << pIter -> second;
   cout << "." << endl;

// insert by moving key
    map<string, int> c2;
    string str("abc");
    cout << "c2[move(str)] == " << c2[move(str)] << endl;
    cout << "c2["abc"] == " << c2["abc"] << endl;

    return (0);
```

```
}
```

Output

```
The keys of the mapped elements are: 1 2 3.
The values of the mapped elements are: 10 20 30.
The keys of the mapped elements are now: 1 2 3 5.
The values of the mapped elements are now: 10 40 30 0.
c2[move(str)] == 0
c2["abc"] == 1
```

map::operator=

Replaces the elements of a map with a copy of another map.

```
map& operator=(const map& right);

map& operator=(map&& right);
```

Parameters

Parameter	Description
`right`	The map being copied into the map.

Remarks

After erasing any existing elements in a `map`, `operator=` either copies or moves the contents of `right` into the map.

Example

```
// map_operator_as.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
   {
   using namespace std;
```

```
    map<int, int> v1, v2, v3;
    map<int, int>::iterator iter;

    v1.insert(pair<int, int>(1, 10));

    cout << "v1 = " ;
    for (iter = v1.begin(); iter != v1.end(); iter++)
       cout << iter->second << " ";
    cout << endl;

    v2 = v1;
    cout << "v2 = ";
    for (iter = v2.begin(); iter != v2.end(); iter++)
       cout << iter->second << " ";
    cout << endl;

// move v1 into v2
    v2.clear();
    v2 = move(v1);
    cout << "v2 = ";
    for (iter = v2.begin(); iter != v2.end(); iter++)
       cout << iter->second << " ";
    cout << endl;
    }
```

map::pointer

A type that provides a pointer to an element in a map.

```
typedef typename allocator_type::pointer pointer;
```

Remarks

A type **pointer** can be used to modify the value of an element.

In most cases, an iterator should be used to access the elements in a map object.

map::rbegin

Returns an iterator addressing the first element in a reversed map.

```
const_reverse_iterator rbegin() const;
```

```
reverse_iterator rbegin();
```

Return Value

A reverse bidirectional iterator addressing the first element in a reversed map or addressing what had been the last element in the unreversed map.

Remarks

`rbegin` is used with a reversed map just as begin is used with a map.

If the return value of `rbegin` is assigned to a `const_reverse_iterator`, then the map object cannot be modified. If the return value of `rbegin` is assigned to a `reverse_iterator`, then the map object can be modified.

`rbegin` can be used to iterate through a map backwards.

Example

```
// map_rbegin.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map <int, int> m1;

   map <int, int> :: iterator m1_Iter;
   map <int, int> :: reverse_iterator m1_rIter;
   map <int, int> :: const_reverse_iterator m1_crIter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 3, 30 ) );

   m1_rIter = m1.rbegin( );
   cout << "The first element of the reversed map m1 is "
        << m1_rIter -> first << "." << endl;

   // begin can be used to start an iteration
   // through a map in a forward order
   cout << "The map is: ";
   for ( m1_Iter = m1.begin( ) ; m1_Iter != m1.end( ); m1_Iter++)
      cout << m1_Iter -> first << " ";
      cout << "." << endl;

   // rbegin can be used to start an iteration
   // through a map in a reverse order
   cout << "The reversed map is: ";
```

```
   for ( m1_rIter = m1.rbegin( ) ; m1_rIter != m1.rend( ); m1_rIter++)
      cout << m1_rIter -> first << " ";
      cout << "." << endl;

   // A map element can be erased by dereferencing to its key
   m1_rIter = m1.rbegin( );
   m1.erase ( m1_rIter -> first );

   m1_rIter = m1.rbegin( );
   cout << "After the erasure, the first element "
        << "in the reversed map is "
        << m1_rIter -> first << "." << endl;
}
```

Output

```
The first element of the reversed map m1 is 3.
The map is: 1 2 3 .
The reversed map is: 3 2 1 .
After the erasure, the first element in the reversed map is 2.
```

map::reference

A type that provides a reference to an element stored in a map.

```
typedef typename allocator_type::reference reference;
```

Example

```
// map_reference.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map <int, int> m1;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );

   // Declare and initialize a const_reference &Ref1
```

```
    // to the key of the first element
    const int &Ref1 = ( m1.begin( ) -> first );

    // The following line would cause an error because the
    // non-const_reference cannot be used to access the key
    // int &Ref1 = ( m1.begin( ) -> first );

    cout << "The key of first element in the map is "
         << Ref1 << "." << endl;

    // Declare and initialize a reference &Ref2
    // to the data value of the first element
    int &Ref2 = ( m1.begin( ) -> second );

    cout << "The data value of first element in the map is "
         << Ref2 << "." << endl;

    //The non-const_reference can be used to modify the
    //data value of the first element
    Ref2 = Ref2 + 5;
    cout << "The modified data value of first element is "
         << Ref2 << "." << endl;
}
```

Output

```
The key of first element in the map is 1.
The data value of first element in the map is 10.
The modified data value of first element is 15.
```

map::rend

Returns an iterator that addresses the location succeeding the last element in a reversed map.

```
const_reverse_iterator rend() const;

reverse_iterator rend();
```

Return Value

A reverse bidirectional iterator that addresses the location succeeding the last element in a reversed map (the location that had preceded the first element in the unreversed map).

Remarks

rend is used with a reversed map just as end is used with a map.

If the return value of rend is assigned to a const_reverse_iterator, then the map object cannot be modified. If the return value of rend is assigned to a reverse_iterator, then the map object can be modified.

rend can be used to test to whether a reverse iterator has reached the end of its map.

The value returned by rend should not be dereferenced.

Example

```
// map_rend.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map <int, int> m1;

   map <int, int> :: iterator m1_Iter;
   map <int, int> :: reverse_iterator m1_rIter;
   map <int, int> :: const_reverse_iterator m1_crIter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 3, 30 ) );

   m1_rIter = m1.rend( );
   m1_rIter--;
   cout << "The last element of the reversed map m1 is "
        << m1_rIter -> first << "." << endl;

   // begin can be used to start an iteration
   // through a map in a forward order
   cout << "The map is: ";
   for ( m1_Iter = m1.begin( ) ; m1_Iter != m1.end( ); m1_Iter++)
      cout << m1_Iter -> first << " ";
      cout << "." << endl;

   // rbegin can be used to start an iteration
   // through a map in a reverse order
   cout << "The reversed map is: ";
   for ( m1_rIter = m1.rbegin( ) ; m1_rIter != m1.rend( ); m1_rIter++)
      cout << m1_rIter -> first << " ";
      cout << "." << endl;

   // A map element can be erased by dereferencing to its key
   m1_rIter = --m1.rend( );
   m1.erase ( m1_rIter -> first );

   m1_rIter = m1.rend( );
   m1_rIter--;
```

```
    cout << "After the erasure, the last element "
         << "in the reversed map is "
         << m1_rIter -> first << "." << endl;
}
```

Output

```
The last element of the reversed map m1 is 1.
The map is: 1 2 3 .
The reversed map is: 3 2 1 .
After the erasure, the last element in the reversed map is 2.
```

map::reverse_iterator

A type that provides a bidirectional iterator that can read or modify an element in a reversed map.

```
typedef std::reverse_iterator<iterator> reverse_iterator;
```

Remarks

A type `reverse_iterator` cannot modify the value of an element and is use to iterate through the map in reverse.

The `reverse_iterator` defined by map points to elements that are objects of value_type, that is of type `pair<`**constKey**, **Type**>, whose first member is the key to the element and whose second member is the mapped datum held by the element.

To dereference a `reverse_iterator``rIter` pointing to an element in a map, use the **->** operator.

To access the value of the key for the element, use `rIter` -> **first**, which is equivalent to (* `rIter`). **first**. To access the value of the mapped datum for the element, use `rIter` -> **second**, which is equivalent to (* `rIter`). **first**.

Example

See example for rbegin for an example of how to declare and use `reverse_iterator`.

map::size

Returns the number of elements in the map.

```
```

```
size_type size() const;
```

Return Value

The current length of the map.

Example

The following example demonstrates the use of the map::size member function.

```
// map_size.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main()
{
    using namespace std;
    map<int, int> m1, m2;
    map<int, int>::size_type i;
    typedef pair<int, int> Int_Pair;

    m1.insert(Int_Pair(1, 1));
    i = m1.size();
    cout << "The map length is " << i << "." << endl;

    m1.insert(Int_Pair(2, 4));
    i = m1.size();
    cout << "The map length is now " << i << "." << endl;
}
```

Output

```
The map length is 1.
The map length is now 2.
```

map::size_type

An unsigned integer type that can represent the number of elements in a map.

```
typedef typename allocator_type::size_type size_type;
```

Example

See the example for size for an example of how to declare and use `size_type`.

map::swap

Exchanges the elements of two maps.

```
void swap(
    map<Key, Type, Traits, Allocator>& right);
```

Parameters

`right`
The argument map providing the elements to be swapped with the target map.

Remarks

The member function invalidates no references, pointers, or iterators that designate elements in the two maps whose elements are being exchanged.

Example

```
// map_swap.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map <int, int> m1, m2, m3;
   map <int, int>::iterator m1_Iter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 3, 30 ) );
   m2.insert ( Int_Pair ( 10, 100 ) );
   m2.insert ( Int_Pair ( 20, 200 ) );
   m3.insert ( Int_Pair ( 30, 300 ) );

   cout << "The original map m1 is:";
   for ( m1_Iter = m1.begin( ); m1_Iter != m1.end( ); m1_Iter++ )
      cout << " " << m1_Iter -> second;
   cout   << "." << endl;

   // This is the member function version of swap
```

```
   //m2 is said to be the argument map; m1 the target map
   m1.swap( m2 );

   cout << "After swapping with m2, map m1 is:";
   for ( m1_Iter = m1.begin( ); m1_Iter != m1.end( ); m1_Iter++ )
      cout << " " << m1_Iter -> second;
   cout  << "." << endl;

   // This is the specialized template version of swap
   swap( m1, m3 );

   cout << "After swapping with m3, map m1 is:";
   for ( m1_Iter = m1.begin( ); m1_Iter != m1.end( ); m1_Iter++ )
      cout << " " << m1_Iter -> second;
   cout   << "." << endl;
}
```

Output

```
The original map m1 is: 10 20 30.
After swapping with m2, map m1 is: 100 200.
After swapping with m3, map m1 is: 300.
```

map::upper_bound

Returns an iterator to the first element in a map that with a key having a value that is greater than that of a specified key.

```
iterator upper_bound(const Key& key);

const_iterator upper_bound(const Key& key) const;
```

Parameters

key
The argument key value to be compared with the sort key value of an element from the map being searched.

Return Value

An **iterator** or `const_iterator` that addresses the location of an element in a map that with a key that is greater than the argument key, or that addresses the location succeeding the last element in the map if no match is found for the key.

If the return value is assigned to a `const_iterator`, the map object cannot be modified. If the return value is assigned to a **iterator**, the map object can be modified.

Example

```
// map_upper_bound.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map <int, int> m1;
   map <int, int> :: const_iterator m1_AcIter, m1_RcIter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 3, 30 ) );

   m1_RcIter = m1.upper_bound( 2 );
   cout << "The first element of map m1 with a key "
        << "greater than 2 is: "
        << m1_RcIter -> second << "." << endl;

   // If no match is found for the key, end is returned
   m1_RcIter = m1. upper_bound ( 4 );

   if ( m1_RcIter == m1.end( ) )
      cout << "The map m1 doesn't have an element "
           << "with a key greater than 4." << endl;
   else
      cout << "The element of map m1 with a key > 4 is: "
           << m1_RcIter -> second << "." << endl;

   // The element at a specific location in the map can be found
   // using a dereferenced iterator addressing the location
   m1_AcIter = m1.begin( );
   m1_RcIter = m1. upper_bound ( m1_AcIter -> first );
   cout << "The 1st element of m1 with a key greater than\n"
        << "that of the initial element of m1 is: "
        << m1_RcIter -> second << "." << endl;
}
```

Output

```
The first element of map m1 with a key greater than 2 is: 30.
The map m1 doesn't have an element with a key greater than 4.
The 1st element of m1 with a key greater than
that of the initial element of m1 is: 20.
```

map::value_comp

The member function returns a function object that determines the order of elements in a map by comparing their key values.

```
value_compare value_comp() const;
```

Return Value

Returns the comparison function object that a map uses to order its elements.

Remarks

For a map *m*, if two elements *e*1(*k*1, *d*1) and *e*2(*k*2, d2) are objects of type `value_type`, where *k*1 and *k*2 are their keys of type `key_type` and d1 and d2 are their data of type `mapped_type`, then *m*.`value_comp`(*e*1, *e*2) is equivalent to *m*.`key_comp`*(k*1, *k*2). A stored object defines the member function

bool operator(**value_type&**`left`, **value_type&**`right`);

which returns **true** if the key value of `left` precedes and is not equal to the key value of `right` in the sort order.

Example

```
// map_value_comp.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;

   map <int, int, less<int> > m1;
   map <int, int, less<int> >::value_compare vc1 = m1.value_comp( );
   pair< map<int,int>::iterator, bool > pr1, pr2;

   pr1= m1.insert ( map <int, int> :: value_type ( 1, 10 ) );
   pr2= m1.insert ( map <int, int> :: value_type ( 2, 5 ) );

   if( vc1( *pr1.first, *pr2.first ) == true )
   {
      cout << "The element ( 1,10 ) precedes the element ( 2,5 )."
           << endl;
   }
   else
   {
      cout << "The element ( 1,10 ) does not precede the element ( 2,5 )."
           << endl;
```

```
   }

   if(vc1( *pr2.first, *pr1.first ) == true )
   {
      cout << "The element ( 2,5 ) precedes the element ( 1,10 )."
           << endl;
   }
   else
   {
      cout << "The element ( 2,5 ) does not precede the element ( 1,10 )."
           << endl;
   }
}
```

Output

```
The element ( 1,10 ) precedes the element ( 2,5 ).
The element ( 2,5 ) does not precede the element ( 1,10 ).
```

map::value_type

The type of object stored as an element in a map.

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

Example

```
// map_value_type.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   typedef pair <const int, int> cInt2Int;
   map <int, int> m1;
   map <int, int> :: key_type key1;
   map <int, int> :: mapped_type mapped1;
   map <int, int> :: value_type value1;
   map <int, int> :: iterator pIter;

   // value_type can be used to pass the correct type
   // explicitly to avoid implicit type conversion
```

```
    m1.insert ( map <int, int> :: value_type ( 1, 10 ) );

    // Compare other ways to insert objects into a map
    m1.insert ( cInt2Int ( 2, 20 ) );
    m1[ 3 ] = 30;

    // Initializing key1 and mapped1
    key1 = ( m1.begin( ) -> first );
    mapped1 = ( m1.begin( ) -> second );

    cout << "The key of first element in the map is "
         << key1 << "." << endl;

    cout << "The data value of first element in the map is "
         << mapped1 << "." << endl;

    // The following line would cause an error because
    // the value_type is not assignable
    // value1 = cInt2Int ( 4, 40 );

    cout  << "The keys of the mapped elements are:";
    for ( pIter = m1.begin( ) ; pIter != m1.end( ) ; pIter++ )
       cout << " " << pIter -> first;
    cout << "." << endl;

    cout  << "The values of the mapped elements are:";
    for ( pIter = m1.begin( ) ; pIter != m1.end( ) ; pIter++ )
       cout << " " << pIter -> second;
    cout << "." << endl;
}
```

See Also

<map> Members
Containers
Thread Safety in the C++ Standard Library
Standard Template Library