multimap 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 multimap Class.

The Standard Template Library multimap class is used for the storage and retrieval of data from a collection in which the each element is a pair that has both a data value and a sort key. The value of the key does not need to be unique and is used to order the data automatically. The value of an element in a multimap, but not its associated key value, may be changed directly. Instead, key values associated with old elements must be deleted and new key values associated with new elements inserted.

Syntax

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

Parameters

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

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

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

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

`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 STL multimap class is

- An associative container, which a variable size container that supports the efficient retrieval of element values based on an associated key value.
- Reversible, because it provides bidirectional iterators to access its elements.
- Sorted, because its elements are ordered by key values within the container in accordance with a specified comparison function.
- Multiple, because its elements do not need to have a unique keys, so that one key value may have many element data values associated with it.
- 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 so independent of the specific type of data contained as elements or keys. The data types to be used for elements and keys are, instead, specified as parameters in the class template along with the comparison function and allocator.

The iterator provided by the map class is a bidirectional iterator, but the class member functions insert and multimap have versions that take as template parameters a weaker input iterator, whose functionality requirements are more minimal than those guaranteed by the class of bidirectional iterators. The different iterator concepts form a family related by refinements in their functionality. Each iterator concept has its own set of requirements and the algorithms that work with them must limit their assumptions to the requirements provided by that type of iterator. It may be assumed that an input iterator may be dereferenced to refer to some object and that it may be incremented to the next iterator in the sequence. This is a minimal set of functionality, but it is enough to be able to talk meaningfully about a range of iterators `[First, Last)` in the context of the class's member functions.

The choice of container type should be based in general on the type of searching and inserting required by the application. Associative containers are optimized for the operations of lookup, insertion and removal. The member functions that explicitly support these operations are efficient, performing them in a time that is on average 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 had specifically pointed at the removed elements.

The multimap should be the associative container of choice when the conditions associating the values with their keys are satisfied by the application. A model for this type of structure is an ordered list of key words with associated string values providing, say, definitions, where the words were not always uniquely defined. If, instead, the key words were uniquely defined so that keys were unique, then a map would be the container of choice. If, on the other hand, just the list of words were being stored, then a set would be the correct container. If multiple occurrences of the words were allowed, then a multiset would be the appropriate container structure.

The multimap orders the sequence it controls by calling a stored function object of type key_compare. This stored object is a comparison function that may be accessed by calling the member function key_comp. In general, the elements need be merely less than comparable to establish this order: so that, given any two elements, it may be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the nonequivalent elements. On a more technical 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, then the ordering becomes total (in the sense that all the elements are ordered with respect to each other) and the keys matched will be indiscernible from each 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

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

Typedefs

allocator_type	A type that represents the `allocator` class for the `multimap` object.
const_iterator	A type that provides a bidirectional iterator that can read a `const` element in the `multimap`.
const_pointer	A type that provides a pointer to a `const` element in a `multimap`.
const_reference	A type that provides a reference to a `const` element stored in a `multimap` for reading and performing `const` operations.
const_reverse_iterator	A type that provides a bidirectional iterator that can read any `const` element in the `multimap`.
difference_type	A signed integer type that can be used to represent the number of elements of a `multimap` in a range between elements pointed to by iterators.
iterator	A type that provides the difference between two iterators that refer to elements within the same `multimap`.
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 `multimap`.
key_type	A type that describes the sort key object that constitutes each element of the `multimap`.
mapped_type	A type that represents the data type stored in a `multimap`.
pointer	A type that provides a pointer to a `const` element in a `multimap`.
reference	A type that provides a reference to an element stored in a `multimap`.

reverse_iterator	A type that provides a bidirectional iterator that can read or modify an element in a reversed `multimap`.
size_type	An unsigned integer type that provides a pointer to a `const` element in a `multimap`.
value_type	A type that provides a function object that can compare two elements as sort keys to determine their relative order in the `multimap`.

Member Functions

begin	Returns an iterator addressing the first element in the `multimap`.
cbegin	Returns a const iterator addressing the first element in the `multimap`.
cend	Returns a const iterator that addresses the location succeeding the last element in a `multimap`.
clear	Erases all the elements of a `multimap`.
count	Returns the number of elements in a `multimap` whose key matches a parameter-specified key.
crbegin	Returns a const iterator addressing the first element in a reversed `multimap`.
crend	Returns a const iterator that addresses the location succeeding the last element in a reversed `multimap`.
emplace	Inserts an element constructed in place into a `multimap`.
emplace_hint	Inserts an element constructed in place into a `multimap`, with a placement hint
empty	Tests if a `multimap` is empty.
end	Returns an iterator that addresses the location succeeding the last element in a `multimap`.
equal_range	Finds the range of elements where the key of the element matches a specified value.
erase	Removes an element or a range of elements in a `multimap` from specified positions or removes elements that match a specified key.
find	Returns an iterator addressing the first location of an element in a `multimap` that has a key equivalent to a specified key.
get_allocator	Returns a copy of the `allocator` object used to construct the `multimap`.
insert	

	Inserts an element or a range of elements into a `multimap`.
key_comp	Retrieves a copy of the comparison object used to order keys in a `multimap`.
lower_bound	Returns an iterator to the first element in a `multimap` that with a key that is equal to or greater than a specified key.
max_size	Returns the maximum length of the `multimap`.
rbegin	Returns an iterator addressing the first element in a reversed `multimap`.
rend	Returns an iterator that addresses the location succeeding the last element in a reversed `multimap`.
size	Returns the number of elements in the `multimap`.
swap	Exchanges the elements of two `multimaps`.
upper_bound	Returns an iterator to the first element in a `multimap` that with a key that is greater than a specified key.
value_comp	The member function returns a function object that determines the order of elements in a `multimap` by comparing their key values.

Operators

operator=	Replaces the elements of a `multimap` with a copy of another `multimap`.

Requirements

Header: <map>

Namespace: std

The (**key**, **value**) pairs are stored in a multimap as objects of type `pair`. The pair class requires the header <utility>, which is automatically included by <map>.

multimap::allocator_type

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


```
typedef Allocator allocator_type;
```

Example

See the example for get_allocator for an example using `allocator_type`.

multimap::begin

Returns an iterator addressing the first element in the multimap.

```
const_iterator begin() const;

iterator begin();
```

Return Value

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

Example

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

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

   multimap <int, int> :: iterator m1_Iter;
   multimap <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 as the iterator is const
    // m1_cIter = m1.begin ( );
    // m1.erase ( m1_cIter );

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

Output

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

multimap::cbegin

Returns a `const` iterator that addresses the first element in the range.

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

Return Value

A `const` bidirectional-access iterator that points at the first element of 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
```

multimap::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.

multimap::clear

Erases all the elements of a multimap.

```
void clear();
```

Example

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

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

int main( )
{
   using namespace std;
   multimap<int, int> m1;
   multimap<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 multimap is initially "
        << i << "." << endl;

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

Output

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

multimap::const_iterator

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

```
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 multimap points to objects of value_type, which are of type `pair<`**const Key**, **Type**>. The value of the key is available through the first member pair and the value of the mapped element is available through the second member of the pair.

To dereference a `const_iterator``cIter` pointing to an element in a multimap, 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 the example for begin for an example using `const_iterator`.

multimap::const_pointer

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

```
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 multimap object.

multimap::const_reference

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

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

Example

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

int main( )
{
   using namespace std;
   multimap <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 the first element in the multimap 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 the first element in the multimap is "
         << Ref2 << "." << endl;
}
```

Output

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

multimap::const_reverse_iterator

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

```
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 multimap in reverse.

The `const_reverse_iterator` defined by multimap points to objects of value_type, which are of type `pair<`**const Key**, **Type**>. The value of the key is available through the first member pair and the value of the mapped element is available through the second member of the pair.

To dereference a `const_reverse_iterator``crIter` pointing to an element in a multimap, 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`.

multimap::count

Returns the number of elements in a multimap whose keys match a parameter-specified key.

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

Parameters

key
The key of the elements to be matched from the multimap.

Return Value

The number of elements whose sort keys match the parameter key; 0 if the multimap doesn't contain an element with a matching key.

Remarks

The member function returns the number of elements in the range

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

that have a key value `key`.

Example

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

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

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

```
    // Elements do not need to have unique keys in multimap,
    // so duplicates are allowed and counted
    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: 2.
The number of elements in m1 with a sort key of 2 is: 2.
The number of elements in m1 with a sort key of 3 is: 0.
```

multimap::crbegin

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

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

Return Value

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

Remarks

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

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

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

Example

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

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

   multimap <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 multimap m1 is "
        << m1_crIter -> first << "." << endl;
}
```

Output

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

multimap::crend

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

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

Return Value

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

Remarks

`crend` is used with a reversed `multimap` just as multimap::end is used with a `multimap`.

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

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

The value returned by `crend` should not be dereferenced.

Example

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

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

   multimap <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 multimap m1 is "
        << m1_crIter -> first << "." << endl;
}
```

Output

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

multimap::difference_type

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

```
typedef typename 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

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

int main( )
{
   using namespace std;
   multimap <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 ) );

   // The following will insert as multimap keys are not unique
   m1.insert ( Int_Pair ( 2, 30 ) );

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

   // Count the number of elements in a multimap
   multimap <int, int>::difference_type  df_count = 0;
   m1_Iter = m1.begin( );
   while ( m1_Iter != m1_eIter )
   {
      df_count++;
      m1_Iter++;
   }

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

Output

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

multimap::emplace

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

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

Parameters

Parameter	Description
args	The arguments forwarded to construct an element to be inserted into the multimap.

Return Value

An iterator to the newly inserted element.

Remarks

No references to container elements are invalidated by this function, but it may invalidate all iterators to the container.

If an exception is thrown during the insertion, the container is left unaltered and the exception is rethrown.

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++

```
// multimap_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()
{
    multimap<string, string> m1;

    m1.emplace("Anna", "Accounting");
    m1.emplace("Bob", "Accounting");
    m1.emplace("Carmine", "Engineering");

    cout << "multimap modified, now contains ";
    print(m1);
    cout << endl;

    m1.emplace("Bob", "Engineering");

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

multimap::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 multimap.
`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.

Remarks

No references to container elements are invalidated by this function, but it may invalidate all iterators to the container.

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.

For a code example, see map::emplace_hint.

multimap::empty

Tests if a multimap is empty.

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

Return Value

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

Example

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

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

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

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

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

Output

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

multimap::end

Returns the past-the-end iterator.

```
const_iterator end() const;


iterator end();
```

Return Value

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

Remarks

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

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

For a code example, see multimap::find.

multimap::equal_range

Finds the range of elements where the key of the element matches a specified value.

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

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

Parameters

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

Return Value

A pair of iterators such that the first is the lower_bound of the key and the second is the upper_bound of the key.

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

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

int main( )
{
   using namespace std;
   typedef multimap <int, int, less<int> > IntMMap;
   IntMMap m1;
   multimap <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 <IntMMap::const_iterator, IntMMap::const_iterator> p1, p2;
   p1 = m1.equal_range( 2 );

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

   cout << "The upper bound of the element with "
        << "a key of 2 in the multimap 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 multimap m1 doesn't have an element "
           << "with a key less than 4." << endl;
   else
      cout << "The element of multimap m1 with a key >= 40 is: "
```

```
            << p1.first -> first << "." << endl;
}
```

Output

```
The lower bound of the element with a key of 2 in the multimap m1 is: 20.
The upper bound of the element with a key of 2 in the multimap 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 multimap m1 doesn't have an element with a key less than 4.
```

multimap::erase

Removes an element or a range of elements in a multimap 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 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 multimap.

Remarks

For a code example, see map::erase.

multimap::find

Returns an iterator that refers to the first location of an element in a multimap 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 multimap 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 multimap (`multimap::end()`) if no match is found for the key.

Remarks

The member function returns an iterator that refers to an element in the multimap 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 multimap object cannot be modified. If the return value of **find** is assigned to an **iterator**, the multimap 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()
{
    multimap<int, string> m1({ { 40, "Zr" }, { 45, "Rh" } });
    cout << "The starting multimap 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 multimap m1 is (key, value):" << endl;
    print_collection(m1);
    cout << endl;
    findit(m1, 45);
    findit(m1, 6);
}
```

multimap::get_allocator

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

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

Return Value

The allocator used by the multimap.

Remarks

Allocators for the multimap 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

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

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

   // The following lines declare objects
   // that use the default allocator.
   multimap <int, int> m1;
   multimap <int, int, allocator<int> > m2;
   multimap <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"
        << endl << "before free memory is exhausted: "
        << m2.max_size( ) << ".\n" << endl;

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

   // The following line creates a multimap m4
   // with the allocator of multimap 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 via the other
   if( m1_Alloc == m4_Alloc )
   {
      cout << "The allocators are interchangeable."
           << endl;
   }
   else
   {
      cout << "The allocators are not interchangeable."
           << endl;
   }
}
```

multimap::insert

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

```
// (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 multimap.
`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-insert member functions, (1) and (2), return an iterator to the position where the new element was inserted into the multimap.

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 multimap.

Remarks

No pointers or references are invalidated by this function, but it may invalidate all iterators to the container.

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.

The value_type of a container is a typedef that belongs to the container, and for map, `multimap<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 multimap 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());` inserts all elements of `v` into `m`.

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 multimap::emplace and multimap::emplace_hint.

Example

C++

```
// multimap_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
    multimap<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
    m1.insert(make_pair(1, 111));

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

    // 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
    multimap<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
    multimap<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;

    multimap<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;
}
```

multimap::iterator

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

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

Remarks

The **iterator** defined by multimap points to objects of value_type, which are of type `pair<`**const Key**, **Type**`>`. The value of the key is available through the first member pair and the value of the mapped element is available through the second member of the pair.

To dereference an **iterator**`Iter` pointing to an element in a multimap, 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**.

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

Example

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

multimap::key_comp

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

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

Return Value

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

Remarks

The stored object defines the member function

bool operator(**const Key&** *x*, **const Key&** *y*);

which returns true if *x* strictly precedes *y* in the sort order.

Example

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

int main( )
{
```

```
   using namespace std;

   multimap <int, int, less<int> > m1;
   multimap <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;
   }

   multimap <int, int, greater<int> > m2;
   multimap <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.
```

multimap::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 multimap.

```
typedef Traits key_compare;
```

Remarks

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

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

Example

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

multimap::key_type

A type that describes the sort key object that constitutes each element of the multimap.

```
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 multimap Class topic.

Example

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

multimap::lower_bound

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

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

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

Parameters

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

Return Value

An iterator or `const_iterator` that addresses the location of an element in a multimap 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 multimap if no match is found for the key.

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

Example

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

int main( )
{
   using namespace std;
   multimap <int, int> m1;
   multimap <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, 20 ) );
   m1.insert ( Int_Pair ( 3, 30 ) );

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

   m1_RcIter = m1.lower_bound( 3 );
   cout << "The first element of multimap m1 with a key of 3 is: "
        << m1_RcIter -> second << "." << endl;

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

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

   // The element at a specific location in the multimap 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 first element of m1 with a key matching\n"
        << "that of the last element is: "
```

```
         << m1_RcIter -> second << "." << endl;

   // Note that the first element with a key equal to
   // the key of the last element is not the last element
   if ( m1_RcIter == --m1.end( ) )
      cout << "This is the last element of multimap m1."
           << endl;
   else
      cout << "This is not the last element of multimap m1."
           << endl;
}
```

Output

```
The element of multimap m1 with a key of 2 is: 20.
The first element of multimap m1 with a key of 3 is: 20.
The multimap m1 doesn't have an element with a key of 4.
The first element of m1 with a key matching
that of the last element is: 20.
This is not the last element of multimap m1.
```

multimap::mapped_type

A type that represents the data type stored in a multimap.

```
typedef Type mapped_type;
```

Remarks

`mapped_type` is a synonym for the template parameter `Type`.

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

Example

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

multimap::max_size

Returns the maximum length of the multimap.

```
```

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

Return Value

The maximum possible length of the multimap.

Example

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

int main( )
{
   using namespace std;
   multimap <int, int> m1;
   multimap <int, int> :: size_type i;

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

multimap::multimap

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

```
multimap();

explicit multimap(
    const Traits& Comp);

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

map(
    const multimap& Right);

multimap(
    multimap&& Right);

multimap(
    initializer_list<value_type> IList);
```

```
multimap(
    initializer_list<value_type> IList,
    const Compare& Comp);

multimap(
    initializer_list<value_type> IList,
    const Compare& Comp,
    const Allocator& Al);

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

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

template <class InputIterator>
multimap(
 InputIterator First,
    InputIterator Last,
    const Traits& Comp,
    const Allocator& Al);
```

Parameters

Parameter	Description
`Al`	The storage allocator class to be used for this multimap object, which defaults to Allocator.
`Comp`	The comparison function of type **constTraits** used to order the elements in the map, which defaults to **Traits**.
`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 to copy the elements.

Remarks

All constructors store a type of allocator object that manages memory storage for the multimap 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 multimap.

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

The first three constructors specify an empty initial multimap, 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 multimap `Right`.

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

The sixth, seventh, and eighth constructors copy the members of an initializer_list.

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

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

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

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

    // Create an empty multimap m1 with the key comparison
    // function of less than, then insert 4 elements
    multimap <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 multimap m2 with the key comparison
    // function of geater than, then insert 2 elements
    multimap <int, int, less<int> > m2;
    m2.insert(Int_Pair(1, 10));
    m2.insert(Int_Pair(2, 20));

    // Create a multimap m3 with the
```

```
// allocator of multimap m1
multimap <int, int>::allocator_type m1_Alloc;
m1_Alloc = m1.get_allocator();
multimap <int, int> m3(less<int>(), m1_Alloc);
m3.insert(Int_Pair(3, 30));

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

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

// Create a multimap m6 by copying the range m4[ first,  last)
// and with the allocator of multimap m2
multimap <int, int>::allocator_type m2_Alloc;
m2_Alloc = m2.get_allocator();
multimap <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 multimap m8 by copying in an initializer_list
multimap<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 multimap m9 with an initializer_list and a comparator
    multimap<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 multimap m10 with an initializer_list, a comparator, and an
allocator
    multimap<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;

}
```

multimap::operator=

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

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

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

Parameters

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

Remarks

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

Example

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

int main( )
   {
   using namespace std;
   multimap<int, int> v1, v2, v3;
   multimap<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;
   }
```

multimap::pointer

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

```
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 multimap object.

multimap::rbegin

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

```
const_reverse_iterator rbegin() const;

reverse_iterator rbegin();
```

Return Value

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

Remarks

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

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

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

Example

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

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

   multimap <int, int> :: iterator m1_Iter;
   multimap <int, int> :: reverse_iterator m1_rIter;
   multimap <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 multimap m1 is "
        << m1_rIter -> first << "." << endl;
```

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

   // A multimap element can be erased by dereferencing 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 multimap is "
        << m1_rIter -> first << "." << endl;
}
```

Output

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

multimap::reference

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

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

Example

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

```
int main( )
{
   using namespace std;
   multimap <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 multimap 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 multimap 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 multimap is 1.
The data value of first element in the multimap is 10.
The modified data value of first element is 15.
```

multimap::rend

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

```
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 multimap (the location that had preceded the first element in the unreversed multimap).

Remarks

`rend` is used with a reversed multimap just as `end` is used with a multimap.

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

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

The value returned by `rend` should not be dereferenced.

Example

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

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

   multimap <int, int> :: iterator m1_Iter;
   multimap <int, int> :: reverse_iterator m1_rIter;
   multimap <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 multimap m1 is "
        << m1_rIter -> first << "." << endl;

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

    // A multimap 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 multimap is "
         << m1_rIter -> first << "." << endl;
}
```

Output

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

multimap::reverse_iterator

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

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

Remarks

A type `reverse_iterator` is use to iterate through the multimap in reverse.

The `reverse_iterator` defined by multimap points to objects of value_type, which are of type `pair<`**const Key**, **Type**>. The value of the key is available through the first member pair and the value of the mapped element is available through the second member of the pair.

To dereference a `reverse_iterator``rIter` pointing to an element in a multimap, 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 the example for rbegin for an example of how to declare and use `reverse_iterator`.

multimap::size

Returns the number of elements in the multimap.

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

Return Value

The current length of the multimap.

Example

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

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

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

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

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

Output

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

multimap::size_type

An unsigned integer type that counts the number of elements in a multimap.

```
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`

multimap::swap

Exchanges the elements of two multimaps.

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

Parameters

`right`
The multimap providing the elements to be swapped, or the multimap whose elements are to be exchanged with those of the multimap `left`.

Remarks

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

Example

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

int main( )
{
   using namespace std;
   multimap <int, int> m1, m2, m3;
   multimap <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 multimap 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
   m1.swap( m2 );

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

Output

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

multimap::upper_bound

Returns an iterator to the first element in a multimap that with a key that is greater than a specified key.

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

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

Parameters

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

Return Value

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

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

Example

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

int main( )
{
   using namespace std;
   multimap <int, int> m1;
   multimap <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.insert ( Int_Pair ( 3, 40 ) );

   m1_RcIter = m1.upper_bound( 1 );
   cout << "The 1st element of multimap m1 with "
        << "a key greater than 1 is: "
        << m1_RcIter -> second << "." << endl;

   m1_RcIter = m1.upper_bound( 2 );
   cout << "The first element of multimap 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.lower_bound( 4 );

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

   // The element at a specific location in the multimap can be
```

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

Output

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

multimap::value_comp

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

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

Return Value

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

Remarks

For a multimap *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).

Example

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

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

```
    multimap <int, int, less<int> > m1;
    multimap <int, int, less<int> >::value_compare vc1 = m1.value_comp( );
    multimap<int,int>::iterator Iter1, Iter2;

    Iter1= m1.insert ( multimap <int, int> :: value_type ( 1, 10 ) );
    Iter2= m1.insert ( multimap <int, int> :: value_type ( 2, 5 ) );

    if( vc1( *Iter1, *Iter2 ) == 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( *Iter2, *Iter1 ) == 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 ).
```

multimap::value_type

A type that represents the type of object stored as an element in a map.

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

Example

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

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

   // value_type can be used to pass the correct type
   // explicitely to avoid implicit type conversion
   m1.insert ( multimap <int, int> :: value_type ( 1, 10 ) );

   // Compare another way to insert objects into a hash_multimap
   m1.insert ( cInt2Int ( 2, 20 ) );

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

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

   cout << "The data value of first element in the multimap 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;
}
```

Output

```
The key of first element in the multimap is 1.
The data value of first element in the multimap is 10.
The keys of the mapped elements are: 1 2.
```

```
The values of the mapped elements are: 10 20.
```

See Also

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