multiset 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 multiset Class.

The Standard Template Library multiset class is used for the storage and retrieval of data from a collection in which the values of the elements contained need not be unique and in which they serve as the key values according to which the data is automatically ordered. The key value of an element in a multiset may not be changed directly. Instead, old values must be deleted and elements with new values inserted.

Syntax

```
template <class Key, class Compare =less <Key>, class Allocator =allocator <Key>>
class multiset
```

Parameters

Key
The element data type to be stored in the multiset.

Compare
The type that provides a function object that can compare two element values as sort keys to determine their relative order in the multiset. 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 multiset's allocation and deallocation of memory. The default value is **allocator**<*Key*>.

Remarks

The STL multiset class is:

- An associative container, which is 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 in the sense that its elements do not need to have unique keys, so that one key value can have many element values associated with it.
- A simple associative container because its element values are 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. The data type to be used is, instead, specified as a parameter in the class template along with the comparison function and allocator.

The iterator provided by the multiset class is a bidirectional iterator, but the class member functions insert and multiset 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 multiset should be the associative container of choice when the conditions associating the values with their keys are satisfies by the application. The elements of a multiset may be multiple and serve as their own sort keys, so keys are not unique. A model for this type of structure is an ordered list of, say, words in which the words may occur more than once. Had multiple occurrences of the words not been allowed, then a set would have been the appropriate container structure. If unique definitions were attached as values to the list of unique key words, then a map would be an appropriate structure to contain this data. If instead the definitions were not unique, then a multimap would be the container of choice.

The multiset orders the sequence it controls by calling a stored function object of type `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

Constructors

multiset	Constructs a `multiset` that is empty or that is a copy of all or part of a specified `multiset`.

Typedefs

allocator_type	A typedef for the `allocator` class for the `multiset` object.
const_iterator	A typedef for a bidirectional iterator that can read a `const` element in the `multiset`.
const_pointer	A typedef for a pointer to a `const` element in a `multiset`.
const_reference	A typedef for a reference to a `const` element stored in a `multiset` for reading and performing `const` operations.
const_reverse_iterator	A typedef for a bidirectional iterator that can read any `const` element in the `multiset`.
difference_type	A signed integer typedef for the number of elements of a `multiset` 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 `multiset`.
key_compare	A typedef for a function object that can compare two keys to determine the relative order of two elements in the `multiset`.
key_type	A typedef for a function object that can compare two sort keys to determine the relative order of two elements in the `multiset`.
pointer	A typedef for a pointer to an element in a `multiset`.
reference	A typedef for a reference to an element stored in a `multiset`.
reverse_iterator	A typedef for a bidirectional iterator that can read or modify an element in a reversed `multiset`.
size_type	An unsigned integer type that can represent the number of elements in a `multiset`.
value_compare	The typedef for a function object that can compare two elements as sort keys to determine their relative order in the `multiset`.
value_type	A typedef that describes an object stored as an element as a `multiset` in its capacity as a value.

Member Functions

begin	Returns an iterator that points to the first element in the `multiset`.
cbegin	Returns a const iterator that addresses the first element in the `multiset`.
cend	Returns a const iterator that addresses the location succeeding the last element in a `multiset`.
clear	Erases all the elements of a `multiset`.
count	Returns the number of elements in a `multiset` whose key matches the key specified as a parameter.
crbegin	Returns a const iterator addressing the first element in a reversed set.
crend	Returns a const iterator that addresses the location succeeding the last element in a reversed set.
emplace	Inserts an element constructed in place into a `multiset`.
emplace_hint	Inserts an element constructed in place into a `multiset`, with a placement hint.
empty	Tests if a `multiset` is empty.
end	Returns an iterator that points to the location after the last element in a `multiset`.
equal_range	Returns a pair of iterators. The first iterator in the pair points to the first element in a `multiset` with a key that is greater than a specified key. The second iterator in the pair points to first element in the `multiset` with a key that is equal to or greater than the key.
erase	Removes an element or a range of elements in a `multiset` from specified positions or removes elements that match a specified key.
find	Returns an iterator that points to the first location of an element in a `multiset` that has a key equal to a specified key.
get_allocator	Returns a copy of the `allocator` object that is used to construct the `multiset`.
insert	Inserts an element or a range of elements into a `multiset`.
key_comp	Provides a function object that can compare two sort keys to determine the relative order of two elements in the `multiset`.
lower_bound	Returns an iterator to the first element in a `multiset` with a key that is equal to or greater than a specified key.
max_size	Returns the maximum length of the `multiset`.

rbegin	Returns an iterator that points to the first element in a reversed `multiset`.
rend	Returns an iterator that points to the location succeeding the last element in a reversed `multiset`.
size	Returns the number of elements in a `multiset`.
swap	Exchanges the elements of two `multisets`.
upper_bound	Returns an iterator to the first element in a `multiset` with a key that is greater than a specified key.
value_comp	Retrieves a copy of the comparison object that is used to order element values in a `multiset`.

Operators

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

Requirements

Header: <set>

Namespace: std

multiset::allocator_type

A type that represents the allocator class for the multiset object

```
typedef Allocator allocator_type;
```

Remarks

`allocator_type` is a synonym for the template parameter `Allocator`.

For more information on `Allocator`, see the Remarks section of the multiset Class topic.

Example

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

multiset::begin

Returns an iterator addressing the first element in the multiset.

```
const_iterator begin() const;

iterator begin();
```

Return Value

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

Example

```
// multiset_begin.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> ms1;
   multiset <int>::iterator ms1_Iter;
   multiset <int>::const_iterator ms1_cIter;

   ms1.insert( 1 );
   ms1.insert( 2 );
   ms1.insert( 3 );

   ms1_Iter = ms1.begin( );
   cout << "The first element of ms1 is " << *ms1_Iter << endl;

   ms1_Iter = ms1.begin( );
   ms1.erase( ms1_Iter );

   // The following 2 lines would err as the iterator is const
   // ms1_cIter = ms1.begin( );
   // ms1.erase( ms1_cIter );

   ms1_cIter = ms1.begin( );
   cout << "The first element of ms1 is now " << *ms1_cIter << endl;
}
```

Output

```
The first element of ms1 is 1
The first element of ms1 is now 2
```

multiset::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
```

multiset::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.

multiset::clear

Erases all the elements of a multiset.

```
void clear();
```

Example

```
// multiset_clear.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> ms1;

   ms1.insert( 1 );
   ms1.insert( 2 );

   cout << "The size of the multiset is initially "
        << ms1.size( ) << "." << endl;
```

```
    ms1.clear( );
    cout << "The size of the multiset after clearing is "
         << ms1.size( ) << "." << endl;
}
```

Output

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

multiset::const_iterator

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

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

Remarks

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

Example

See the example for begin for an example using `const_iterator`.

multiset::const_pointer

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

```
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 multiset object.

multiset::const_reference

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

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

Example

```
// multiset_const_ref.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> ms1;

   ms1.insert( 10 );
   ms1.insert( 20 );

   // Declare and initialize a const_reference &Ref1
   // to the 1st element
   const int &Ref1 = *ms1.begin( );

   cout << "The first element in the multiset is "
        << Ref1 << "." << endl;

   // The following line would cause an error because the
   // const_reference cannot be used to modify the multiset
   // Ref1 = Ref1 + 5;
}
```

Output

```
The first element in the multiset is 10.
```

multiset::const_reverse_iterator

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

```
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 multiset in reverse.

Example

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

multiset::count

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

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

Parameters

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

Return Value

The number of elements in the multiset whose sort key matches the parameter key.

Remarks

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

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

Example

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

```
// multiset_count.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main()
{
    using namespace std;
    multiset<int> ms1;
    multiset<int>::size_type i;

    ms1.insert(1);
    ms1.insert(1);
```

```
    ms1.insert(2);

    // Elements do not need to be unique in multiset,
    // so duplicates are allowed and counted.
    i = ms1.count(1);
    cout << "The number of elements in ms1 with a sort key of 1 is: "
         << i << "." << endl;

    i = ms1.count(2);
    cout << "The number of elements in ms1 with a sort key of 2 is: "
         << i << "." << endl;

    i = ms1.count(3);
    cout << "The number of elements in ms1 with a sort key of 3 is: "
         << i << "." << endl;
}
```

Output

```
The number of elements in ms1 with a sort key of 1 is: 2.
The number of elements in ms1 with a sort key of 2 is: 1.
The number of elements in ms1 with a sort key of 3 is: 0.
```

multiset::crbegin

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

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

Return Value

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

Remarks

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

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

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

Example

```
// multiset_crbegin.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> ms1;
   multiset <int>::const_reverse_iterator ms1_crIter;

   ms1.insert( 10 );
   ms1.insert( 20 );
   ms1.insert( 30 );

   ms1_crIter = ms1.crbegin( );
   cout << "The first element in the reversed multiset is "
        << *ms1_crIter << "." << endl;
}
```

Output

```
The first element in the reversed multiset is 30.
```

multiset::crend

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

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

Return Value

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

Remarks

`crend` is used with a reversed multiset just as end is used with a multiset.

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

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

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

Example

```
// multiset_crend.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main() {
   using namespace std;
   multiset <int> ms1;
   multiset <int>::const_reverse_iterator ms1_crIter;

   ms1.insert( 10 );
   ms1.insert( 20 );
   ms1.insert( 30 );

   ms1_crIter = ms1.crend( ) ;
   ms1_crIter--;
   cout << "The last element in the reversed multiset is "
        << *ms1_crIter << "." << endl;
}
```

multiset::difference_type

A signed integer type that can be used to represent the number of elements of a multiset 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 like set, subtraction between iterators is only supported by random-access iterators provided by a random-access container like vector.

Example

```
```

```
// multiset_diff_type.cpp
// compile with: /EHsc
#include <iostream>
#include <set>
#include <algorithm>

int main( )
{
   using namespace std;

   multiset <int> ms1;
   multiset <int>::iterator ms1_Iter, ms1_bIter, ms1_eIter;

   ms1.insert( 20 );
   ms1.insert( 10 );
   ms1.insert( 20 );

   ms1_bIter = ms1.begin( );
   ms1_eIter = ms1.end( );

   multiset <int>::difference_type   df_typ5, df_typ10, df_typ20;

   df_typ5 = count( ms1_bIter, ms1_eIter, 5 );
   df_typ10 = count( ms1_bIter, ms1_eIter, 10 );
   df_typ20 = count( ms1_bIter, ms1_eIter, 20 );

   // The keys, and hence the elements, of a multiset are not unique
   cout << "The number '5' occurs " << df_typ5
        << " times in multiset ms1.\n";
   cout << "The number '10' occurs " << df_typ10
        << " times in multiset ms1.\n";
   cout << "The number '20' occurs " << df_typ20
        << " times in multiset ms1.\n";

   // Count the number of elements in a multiset
   multiset <int>::difference_type  df_count = 0;
   ms1_Iter = ms1.begin( );
   while ( ms1_Iter != ms1_eIter)
   {
      df_count++;
      ms1_Iter++;
   }

   cout << "The number of elements in the multiset ms1 is: "
        << df_count << "." << endl;
}
```

Output

```
The number '5' occurs 0 times in multiset ms1.
The number '10' occurs 1 times in multiset ms1.
The number '20' occurs 2 times in multiset ms1.
```

```
The number of elements in the multiset ms1 is: 3.
```

multiset::emplace

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

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

Parameters

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

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.

Example

C++

```
// multiset_emplace.cpp
// compile with: /EHsc
#include <set>
#include <string>
#include <iostream>

using namespace std;

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

    for (const auto& p : s) {
        cout << "(" << p << ") ";
    }
```

```
    cout << endl;
}

int main()
{
    multiset<string> s1;

    s1.emplace("Anna");
    s1.emplace("Bob");
    s1.emplace("Carmine");

    cout << "multiset modified, now contains ";
    print(s1);
    cout << endl;

    s1.emplace("Bob");

    cout << "multiset modified, now contains ";
    print(s1);
    cout << endl;
}
```

multiset::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 multiset.
`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.

For a code example, see set::emplace_hint.

multiset::empty

Tests if a multiset is empty.

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

Return Value

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

Example

```
// multiset_empty.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> ms1, ms2;
   ms1.insert ( 1 );

   if ( ms1.empty( ) )
      cout << "The multiset ms1 is empty." << endl;
   else
      cout << "The multiset ms1 is not empty." << endl;

   if ( ms2.empty( ) )
      cout << "The multiset ms2 is empty." << endl;
   else
      cout << "The multiset ms2 is not empty." << endl;
}
```

Output

```
The multiset ms1 is not empty.
The multiset ms2 is empty.
```

multiset::end

Returns the past-the-end iterator.

```
const_iterator end() const;


iterator end();
```

Return Value

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

Remarks

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

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

For a code example, see multiset::find.

multiset::equal_range

Returns a pair of iterators respectively to the first element in a multiset with a key that is greater than a specified key and to the first element in the multiset with a key that is equal to or greater than 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 to be compared with the sort key of an element from the multiset 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

```
// multiset_equal_range.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   typedef multiset<int, less<int> > IntSet;
   IntSet ms1;
   multiset <int> :: const_iterator ms1_RcIter;

   ms1.insert( 10 );
   ms1.insert( 20 );
   ms1.insert( 30 );

   pair <IntSet::const_iterator, IntSet::const_iterator> p1, p2;
   p1 = ms1.equal_range( 20 );

   cout << "The upper bound of the element with "
        << "a key of 20 in the multiset ms1 is: "
        << *( p1.second ) << "." << endl;

   cout << "The lower bound of the element with "
        << "a key of 20 in the multiset ms1 is: "
        << *( p1.first ) << "." << endl;

   // Compare the upper_bound called directly
   ms1_RcIter = ms1.upper_bound( 20 );
   cout << "A direct call of upper_bound( 20 ) gives "
        << *ms1_RcIter << "," << endl
        << "matching the 2nd element of the pair"
        << " returned by equal_range( 20 )." << endl;

   p2 = ms1.equal_range( 40 );

   // If no match is found for the key,
   // both elements of the pair return end( )
   if ( ( p2.first == ms1.end( ) ) && ( p2.second == ms1.end( ) ) )
      cout << "The multiset ms1 doesn't have an element "
              << "with a key less than 40." << endl;
   else
      cout << "The element of multiset ms1 with a key >= 40 is: "
```

```
                << *( p1.first ) << "." << endl;
}
```

Output

```
The upper bound of the element with a key of 20 in the multiset ms1 is: 30.
The lower bound of the element with a key of 20 in the multiset ms1 is: 20.
A direct call of upper_bound( 20 ) gives 30,
matching the 2nd element of the pair returned by equal_range( 20 ).
The multiset ms1 doesn't have an element with a key less than 40.
```

multiset::erase

Removes an element or a range of elements in a multiset 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 multiset if no such element exists.

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

Remarks

For a code example, see set::erase.

multiset::find

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

Remarks

The member function returns an iterator that refers to an element in the multiset whose 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 multiset object cannot be modified. If the return value of **find** is assigned to an **iterator**, the multiset object can be modified

Example

C++

```
// compile with: /EHsc /W4 /MTd
#include <set>
#include <iostream>
#include <vector>
#include <string>

using namespace std;

template <typename T> void print_elem(const T& t) {
    cout << "(" << t << ") ";
}

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()
{
    multiset<int> s1({ 40, 45 });
    cout << "The starting multiset s1 is: " << endl;
    print_collection(s1);

    vector<int> v;
    v.push_back(43);
    v.push_back(41);
    v.push_back(46);
    v.push_back(42);
    v.push_back(44);
    v.push_back(44); // attempt a duplicate

    cout << "Inserting the following vector data into s1: " << endl;
    print_collection(v);

    s1.insert(v.begin(), v.end());

    cout << "The modified multiset s1 is: " << endl;
    print_collection(s1);
    cout << endl;
    findit(s1, 45);
    findit(s1, 6);
}
```

multiset::get_allocator

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

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

Return Value

The allocator used by the multiset.

Remarks

Allocators for the multiset 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

```
// multiset_get_allocator.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int>::allocator_type ms1_Alloc;
   multiset <int>::allocator_type ms2_Alloc;
   multiset <double>::allocator_type ms3_Alloc;
   multiset <int>::allocator_type ms4_Alloc;

   // The following lines declare objects
   // that use the default allocator.
   multiset <int> ms1;
   multiset <int, allocator<int> > ms2;
   multiset <double, allocator<double> > ms3;

   cout << "The number of integers that can be allocated"
        << endl << "before free memory is exhausted: "
        << ms2.max_size( ) << "." << endl;

   cout << "The number of doubles that can be allocated"
        << endl << "before free memory is exhausted: "
        << ms3.max_size( ) <<  "." << endl;

   // The following lines create a multiset ms4
   // with the allocator of multiset ms1
   ms1_Alloc = ms1.get_allocator( );
   multiset <int> ms4( less<int>( ), ms1_Alloc );
   ms4_Alloc = ms4.get_allocator( );

   // Two allocators are interchangeable if
   // storage allocated from each can be
   // deallocated with the other
   if( ms1_Alloc == ms4_Alloc )
   {
      cout << "Allocators are interchangeable."
           << endl;
   }
```

```
   else
   {
      cout << "Allocators are not interchangeable."
           << endl;
   }
}
```

multiset::insert

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

```
// (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 multiset.
`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 multiset 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 multiset.

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 multiset.

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 set, `multiset<V>::value_type` is type `const V`.

The range member function (5) inserts the sequence of element values into a multiset 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 `s.insert(v.begin(), v.end());` inserts all elements of `v` into `s`.

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

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

Example

C++

```cpp
// multiset_insert.cpp
// compile with: /EHsc
#include <set>
#include <iostream>
#include <string>
#include <vector>

using namespace std;

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

    for (const auto& p : s) {
        cout << "(" << p << ") ";
    }

    cout << endl;
}

int main()
{

    // insert single values
    multiset<int> s1;
    // call insert(const value_type&) version
    s1.insert({ 1, 10 });
    // call insert(ValTy&&) version
    s1.insert(20);

    cout << "The original multiset values of s1 are:" << endl;
    print(s1);

    // intentionally attempt a duplicate, single element
    s1.insert(1);
    cout << "The modified multiset values of s1 are:" << endl;
    print(s1);
    cout << endl;

    // single element, with hint
    s1.insert(s1.end(), 30);
    cout << "The modified multiset values of s1 are:" << endl;
    print(s1);
    cout << endl;

    // The templatized version inserting a jumbled range
    multiset<int> s2;
    vector<int> v;
    v.push_back(43);
    v.push_back(294);
    v.push_back(41);
```

```
    v.push_back(330);
    v.push_back(42);
    v.push_back(45);

    cout << "Inserting the following vector data into s2:" << endl;
    print(v);

    s2.insert(v.begin(), v.end());

    cout << "The modified multiset values of s2 are:" << endl;
    print(s2);
    cout << endl;

    // The templatized versions move-constructing elements
    multiset<string>  s3;
    string str1("blue"), str2("green");

    // single element
    s3.insert(move(str1));
    cout << "After the first move insertion, s3 contains:" << endl;
    print(s3);

    // single element with hint
    s3.insert(s3.end(), move(str2));
    cout << "After the second move insertion, s3 contains:" << endl;
    print(s3);
    cout << endl;

    multiset<int> s4;
    // Insert the elements from an initializer_list
    s4.insert({ 4, 44, 2, 22, 3, 33, 1, 11, 5, 55 });
    cout << "After initializer_list insertion, s4 contains:" << endl;
    print(s4);
    cout << endl;
}
```

multiset::iterator

A type that provides a constant bidirectional iterator that can read any element in a multiset.

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

Example

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

multiset::key_comp

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

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

Return Value

Returns the function object that a multiset uses to order its elements, which is the template parameter `Compare`.

For more information on `Compare`, see the Remarks section of the multiset Class topic.

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.

Note that both key_compare and value_compare are synonyms for the template parameter `Compare`. Both types are provided for the classes set and multiset, where they are identical, for compatibility with the classes map and multimap, where they are distinct.

Example

```
// multiset_key_comp.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;

   multiset <int, less<int> > ms1;
   multiset <int, less<int> >::key_compare kc1 = ms1.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 s1."
           << endl;
   }
   else
   {
      cout << "kc1( 2,3 ) returns value of false "
           << "where kc1 is the function object of ms1."
```

```
            << endl;
    }

    multiset <int, greater<int> > ms2;
    multiset <int, greater<int> >::key_compare kc2 = ms2.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 ms2."
            << endl;
    }
    else
    {
       cout << "kc2( 2,3 ) returns value of false, "
            << "where kc2 is the function object of ms2."
            << endl;
    }
}
```

Output

```
kc1( 2,3 ) returns value of true, where kc1 is the function object of s1.
kc2( 2,3 ) returns value of false, where kc2 is the function object of ms2.
```

multiset::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 multiset.

```
typedef Compare key_compare;
```

Remarks

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

For more information on `Compare`, see the Remarks section of the multiset Class topic.

Example

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

multiset::key_type

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

```
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 multiset Class topic.

Example

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

multiset::lower_bound

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

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

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

Parameters

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

Return Value

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

Example

```
// multiset_lower_bound.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> ms1;
```

```
    multiset <int> :: const_iterator ms1_AcIter, ms1_RcIter;

    ms1.insert( 10 );
    ms1.insert( 20 );
    ms1.insert( 30 );

    ms1_RcIter = ms1.lower_bound( 20 );
    cout << "The element of multiset ms1 with a key of 20 is: "
         << *ms1_RcIter << "." << endl;

    ms1_RcIter = ms1.lower_bound( 40 );

    // If no match is found for the key, end( ) is returned
    if ( ms1_RcIter == ms1.end( ) )
       cout << "The multiset ms1 doesn't have an element "
            << "with a key of 40." << endl;
    else
       cout << "The element of multiset ms1 with a key of 40 is: "
            << *ms1_RcIter << "." << endl;

    // The element at a specific location in the multiset can be
    // found using a derefenced iterator addressing the location
    ms1_AcIter = ms1.end( );
    ms1_AcIter--;
    ms1_RcIter = ms1.lower_bound( *ms1_AcIter );
    cout << "The element of ms1 with a key matching "
         << "that of the last element is: "
         << *ms1_RcIter << "." << endl;
}
```

Output

```
The element of multiset ms1 with a key of 20 is: 20.
The multiset ms1 doesn't have an element with a key of 40.
The element of ms1 with a key matching that of the last element is: 30.
```

multiset::max_size

Returns the maximum length of the multiset.

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

Return Value

The maximum possible length of the multiset.

Example

```
// multiset_max_size.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> ms1;
   multiset <int>::size_type i;

   i = ms1.max_size( );
   cout << "The maximum possible length "
        << "of the multiset is " << i << "." << endl;
}
```

multiset::multiset

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

```
multiset();

explicit multiset (
    const Compare& Comp);

multiset (
    const Compare& Comp,
    const Allocator& Al);

multiset(
    const multiset& Right);

multiset(
    multiset&& Right);

multiset(
    initializer_list<Type> IList);

multiset(
    initializer_list<Type> IList,
    const Compare& Comp);

multiset(
    initializer_list<Type> IList,
    const Compare& Comp,
```

```
    const Allocator& Al);


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

template <class InputIterator>
multiset (
    InputIterator First,
    InputIterator Last,
    const Compare& Comp);

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

Parameters

Parameter	Description
`Al`	The storage allocator class to be used for this multiset object, which defaults to `Allocator`.
`Comp`	The comparison function of type `const Compare` used to order the elements in the multiset, which defaults to `Compare`.
`Right`	The multiset of which the constructed multiset 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 multiset 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 multiset.

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

The first three constructors specify an empty initial multiset, 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 keyword `explicit` suppresses certain kinds of automatic type conversion.

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

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

The sixth, seventh, and eighth constructors specify an initializer_list from which to copy the elements.

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

Example

```
// multiset_ctor.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main()
{
    using namespace std;
    //multiset <int>::iterator ms1_Iter, ms2_Iter, ms3_Iter;
    multiset <int>::iterator ms4_Iter, ms5_Iter, ms6_Iter, ms7_Iter;

    // Create an empty multiset ms0 of key type integer
    multiset <int> ms0;

    // Create an empty multiset ms1 with the key comparison
    // function of less than, then insert 4 elements
    multiset <int, less<int> > ms1;
    ms1.insert(10);
    ms1.insert(20);
    ms1.insert(20);
    ms1.insert(40);

    // Create an empty multiset ms2 with the key comparison
    // function of geater than, then insert 2 elements
    multiset <int, less<int> > ms2;
    ms2.insert(10);
    ms2.insert(20);

    // Create a multiset ms3 with the
    // allocator of multiset ms1
    multiset <int>::allocator_type ms1_Alloc;
    ms1_Alloc = ms1.get_allocator();
    multiset <int> ms3(less<int>(), ms1_Alloc);
    ms3.insert(30);

    // Create a copy, multiset ms4, of multiset ms1
    multiset <int> ms4(ms1);
```

```
 // Create a multiset ms5 by copying the range ms1[ first,  last)
 multiset <int>::const_iterator ms1_bcIter, ms1_ecIter;
 ms1_bcIter = ms1.begin();
 ms1_ecIter = ms1.begin();
 ms1_ecIter++;
 ms1_ecIter++;
 multiset <int> ms5(ms1_bcIter, ms1_ecIter);

 // Create a multiset ms6 by copying the range ms4[ first,  last)
 // and with the allocator of multiset ms2
 multiset <int>::allocator_type ms2_Alloc;
 ms2_Alloc = ms2.get_allocator();
 multiset <int> ms6(ms4.begin(), ++ms4.begin(), less<int>(), ms2_Alloc);

 cout << "ms1 =";
 for (auto i : ms1)
     cout << " " << i;
 cout << endl;

 cout << "ms2 =";
 for (auto i : ms2)
     cout << " " << i;
cout << endl;

cout << "ms3 =";
for (auto i : ms3)
    cout << " " << i;
 cout << endl;

 cout << "ms4 =";
 for (auto i : ms4)
     cout << " " << i;
 cout << endl;

 cout << "ms5 =";
 for (auto i : ms5)
     cout << " " << i;
 cout << endl;

 cout << "ms6 =";
 for (auto i : ms6)
     cout << " " << i;
 cout << endl;

 // Create a multiset by moving ms5
 multiset<int> ms7(move(ms5));
 cout << "ms7 =";
 for (auto i : ms7)
     cout << " " << i;
 cout << endl;

 // Create a multiset with an initializer_list
 multiset<int> ms8({1, 2, 3, 4});
 cout << "ms8=";
 for (auto i : ms8)
```

```
        cout << " " << i;
    cout << endl;
}
```

multiset::operator=

Replaces the elements of this `multiset` using elements from another `multiset`.

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

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

Parameters

Parameter	Description
`right`	The `multiset` from which elements are copied or moved.

Remarks

`operator=` copies or moves the elements in `right` into this `multiset`, depending on the reference type (lvalue or rvalue) used. Elements that are in this `multiset` before `operator=` executes are discarded.

Example

```
// multiset_operator_as.cpp
// compile with: /EHsc
#include <multiset>
#include <iostream>

int main( )
   {
   using namespace std;
   multiset<int> v1, v2, v3;
   multiset<int>::iterator iter;

   v1.insert(10);

   cout << "v1 = " ;
   for (iter = v1.begin(); iter != v1.end(); iter++)
      cout << *iter << " ";
   cout << endl;
```

```
    v2 = v1;
    cout << "v2 = ";
    for (iter = v2.begin(); iter != v2.end(); iter++)
       cout << *iter << " ";
    cout << endl;

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

multiset::pointer

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

```
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 multiset object.

multiset::rbegin

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

```
const_reverse_iterator rbegin() const;

reverse_iterator rbegin();
```

Return Value

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

Remarks

`rbegin` is used with a reversed multiset just as rbegin is used with a multiset.

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

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

Example

```
// multiset_rbegin.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> ms1;
   multiset <int>::iterator ms1_Iter;
   multiset <int>::reverse_iterator ms1_rIter;

   ms1.insert( 10 );
   ms1.insert( 20 );
   ms1.insert( 30 );

   ms1_rIter = ms1.rbegin( );
   cout << "The first element in the reversed multiset is "
        << *ms1_rIter << "." << endl;

   // begin can be used to start an interation
   // throught a multiset in a forward order
   cout << "The multiset is:";
   for ( ms1_Iter = ms1.begin( ) ; ms1_Iter != ms1.end( ); ms1_Iter++ )
      cout << " " << *ms1_Iter;
   cout << endl;

   // rbegin can be used to start an interation
   // throught a multiset in a reverse order
   cout << "The reversed multiset is:";
   for ( ms1_rIter = ms1.rbegin( ) ; ms1_rIter != ms1.rend( ); ms1_rIter++ )
      cout << " " << *ms1_rIter;
   cout << endl;

   // A multiset element can be erased by dereferencing to its key
   ms1_rIter = ms1.rbegin( );
   ms1.erase ( *ms1_rIter );

   ms1_rIter = ms1.rbegin( );
   cout << "After the erasure, the first element "
        << "in the reversed multiset is "<< *ms1_rIter << "."
```

```
        << endl;
}
```

Output

```
The first element in the reversed multiset is 30.
The multiset is: 10 20 30
The reversed multiset is: 30 20 10
After the erasure, the first element in the reversed multiset is 20.
```

multiset::reference

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

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

Example

```
// multiset_ref.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> ms1;

   ms1.insert( 10 );
   ms1.insert( 20 );

   // Declare and initialize a reference &Ref1 to the 1st element
   const int &Ref1 = *ms1.begin( );

   cout << "The first element in the multiset is "
        << Ref1 << "." << endl;
}
```

Output

```
The first element in the multiset is 10.
```

multiset::rend

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

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

Remarks

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

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

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

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

Example

```
// multiset_rend.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main() {
   using namespace std;
   multiset <int> ms1;
   multiset <int>::iterator ms1_Iter;
   multiset <int>::reverse_iterator ms1_rIter;
   multiset <int>::const_reverse_iterator ms1_crIter;

   ms1.insert( 10 );
   ms1.insert( 20 );
   ms1.insert( 30 );

   ms1_rIter = ms1.rend( ) ;
   ms1_rIter--;
   cout << "The last element in the reversed multiset is "
        << *ms1_rIter << "." << endl;
```

```
    // end can be used to terminate an interation
    // throught a multiset in a forward order
    cout << "The multiset is: ";
    for ( ms1_Iter = ms1.begin( ) ; ms1_Iter != ms1.end( ); ms1_Iter++ )
       cout << *ms1_Iter << " ";
    cout << "." << endl;

    // rend can be used to terminate an interation
    // throught a multiset in a reverse order
    cout << "The reversed multiset is: ";
    for ( ms1_rIter = ms1.rbegin( ) ; ms1_rIter != ms1.rend( ); ms1_rIter++ )
       cout << *ms1_rIter << " ";
    cout << "." << endl;

    ms1_rIter = ms1.rend( );
    ms1_rIter--;
    ms1.erase ( *ms1_rIter );

    ms1_rIter = ms1.rend( );
    --ms1_rIter;
    cout << "After the erasure, the last element in the "
         << "reversed multiset is " << *ms1_rIter << "." << endl;
}
```

multiset::reverse_iterator

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

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

Remarks

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

Example

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

multiset::size

Returns the number of elements in the multiset.

```
```

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

Return Value

The current length of the multiset.

Example

```
// multiset_size.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> ms1;
   multiset <int> :: size_type i;

   ms1.insert( 1 );
   i = ms1.size( );
   cout << "The multiset length is " << i << "." << endl;

   ms1.insert( 2 );
   i = ms1.size( );
   cout << "The multiset length is now " << i << "." << endl;
}
```

Output

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

multiset::size_type

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

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

Example

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

multiset::swap

Exchanges the elements of two multisets.

```
void swap(
    multiset<Key, Compare, Allocator>& right);
```

Parameters

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

Remarks

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

Example

```
// multiset_swap.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> ms1, ms2, ms3;
   multiset <int>::iterator ms1_Iter;

   ms1.insert( 10 );
   ms1.insert( 20 );
   ms1.insert( 30 );
   ms2.insert( 100 );
   ms2.insert( 200 );
   ms3.insert( 300 );

   cout << "The original multiset ms1 is:";
   for ( ms1_Iter = ms1.begin( ); ms1_Iter != ms1.end( ); ms1_Iter++ )
      cout << " " << *ms1_Iter;
   cout << "." << endl;

   // This is the member function version of swap
   ms1.swap( ms2 );

   cout << "After swapping with ms2, list ms1 is:";
   for ( ms1_Iter = ms1.begin( ); ms1_Iter != ms1.end( ); ms1_Iter++ )
```

```
      cout << " " << *ms1_Iter;
   cout << "." << endl;

   // This is the specialized template version of swap
   swap( ms1, ms3 );

   cout << "After swapping with ms3, list ms1 is:";
   for ( ms1_Iter = ms1.begin( ); ms1_Iter != ms1.end( ); ms1_Iter++ )
      cout << " " << *ms1_Iter;
   cout   << "." << endl;
}
```

Output

```
The original multiset ms1 is: 10 20 30.
After swapping with ms2, list ms1 is: 100 200.
After swapping with ms3, list ms1 is: 300.
```

multiset::upper_bound

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

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

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

Parameters

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

Return Value

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

Example

```
// multiset_upper_bound.cpp
// compile with: /EHsc
#include <set>
#include <iostream>
```

```
int main( )
{
   using namespace std;
   multiset <int> ms1;
   multiset <int> :: const_iterator ms1_AcIter, ms1_RcIter;

   ms1.insert( 10 );
   ms1.insert( 20 );
   ms1.insert( 30 );

   ms1_RcIter = ms1.upper_bound( 20 );
   cout << "The first element of multiset ms1 with a key greater "
        << "than 20 is: " << *ms1_RcIter << "." << endl;

   ms1_RcIter = ms1.upper_bound( 30 );

   // If no match is found for the key, end( ) is returned
   if ( ms1_RcIter == ms1.end( ) )
      cout << "The multiset ms1 doesn't have an element "
           << "with a key greater than 30." << endl;
   else
      cout << "The element of multiset ms1 with a key > 40 is: "
           << *ms1_RcIter << "." << endl;

   // The element at a specific location in the multiset can be
   // found using a dereferenced iterator addressing the location
   ms1_AcIter = ms1.begin( );
   ms1_RcIter = ms1.upper_bound( *ms1_AcIter );
   cout << "The first element of ms1 with a key greater than"
        << endl << "that of the initial element of ms1 is: "
        << *ms1_RcIter << "." << endl;
}
```

Output

```
The first element of multiset ms1 with a key greater than 20 is: 30.
The multiset ms1 doesn't have an element with a key greater than 30.
The first element of ms1 with a key greater than
that of the initial element of ms1 is: 20.
```

multiset::value_comp

Retrieves a copy of the comparison object used to order element values in a multiset.

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

Return Value

Returns the function object that a multiset uses to order its elements, which is the template parameter `Compare`.

For more information on `Compare`, see the Remarks section of the multiset Class topic.

Remarks

The stored object defines the member function:

bool operator(**const Key&**_xVal, **const Key&**_yVal);

which returns true if _xVal precedes and is not equal to _yVal in the sort order.

Note that both key_compare and value_compare are synonyms for the template parameter `Compare`. Both types are provided for the classes set and multiset, where they are identical, for compatibility with the classes map and multimap, where they are distinct.

Example

```
// multiset_value_comp.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;

   multiset <int, less<int> > ms1;
   multiset <int, less<int> >::value_compare vc1 = ms1.value_comp( );
   bool result1 = vc1( 2, 3 );
   if( result1 == true )
   {
      cout << "vc1( 2,3 ) returns value of true, "
           << "where vc1 is the function object of ms1."
           << endl;
   }
   else
   {
      cout << "vc1( 2,3 ) returns value of false, "
           << "where vc1 is the function object of ms1."
           << endl;
   }

   set <int, greater<int> > ms2;
   set<int, greater<int> >::value_compare vc2 = ms2.value_comp( );
   bool result2 = vc2( 2, 3 );
   if( result2 == true )
   {
      cout << "vc2( 2,3 ) returns value of true, "
           << "where vc2 is the function object of ms2."
           << endl;
   }
```

```
    else
    {
       cout << "vc2( 2,3 ) returns value of false, "
            << "where vc2 is the function object of ms2."
            << endl;
    }
}
```

Output

```
vc1( 2,3 ) returns value of true, where vc1 is the function object of ms1.
vc2( 2,3 ) returns value of false, where vc2 is the function object of ms2.
```

multiset::value_compare

The type that provides a function object that can compare two sort keys to determine their relative order in the multiset.

```
typedef key_compare value_compare;
```

Remarks

`value_compare` is a synonym for the template parameter `Compare`.

Note that both key_compare and **value_compare** are synonyms for the template parameter `Compare`. Both types are provided for the classes set and multiset, where they are identical, for compatibility with the classes map and multimap, where they are distinct.

For more information on `Compare`, see the Remarks section of the multiset Class topic.

Example

See the example for value_comp for an example of how to declare and use `value_compare`.

multiset::value_type

A type that describes an object stored as an element as a multiset in its capacity as a value.

```
typedef Key value_type;
```

Remarks

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

Note that both key_type and `value_type` are synonyms for the template parameter **Key**. Both types are provided for the classes set and multiset, where they are identical, for compatibility with the classes map and multimap, where they are distinct.

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

Example

```
// multiset_value_type.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> ms1;
   multiset <int>::iterator ms1_Iter;

   multiset <int> :: value_type svt_Int;   // Declare value_type
   svt_Int = 10;             // Initialize value_type

   multiset <int> :: key_type skt_Int;   // Declare key_type
   skt_Int = 20;             // Initialize key_type

   ms1.insert( svt_Int );         // Insert value into s1
   ms1.insert( skt_Int );         // Insert key into s1

   // A multiset accepts key_types or value_types as elements
   cout << "The multiset has elements:";
   for ( ms1_Iter = ms1.begin( ) ; ms1_Iter != ms1.end( ); ms1_Iter++ )
      cout << " " << *ms1_Iter;
   cout << "." << endl;
}
```

Output

```
The multiset has elements: 10 20.
```

See Also

<set> Members
Containers

Thread Safety in the C++ Standard Library
Standard Template Library