Username: Pralay Patoria **Book:** The C++ Standard Library: A Tutorial and Reference, Second Edition. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

7.7. Sets and Multisets

Set and multiset containers sort their elements automatically according to a certain sorting criterion. The difference between the two types of containers is that multisets allow duplicates, whereas sets do not (see Figure 7.12 and the earlier discussion on this topic in Chapter 6).

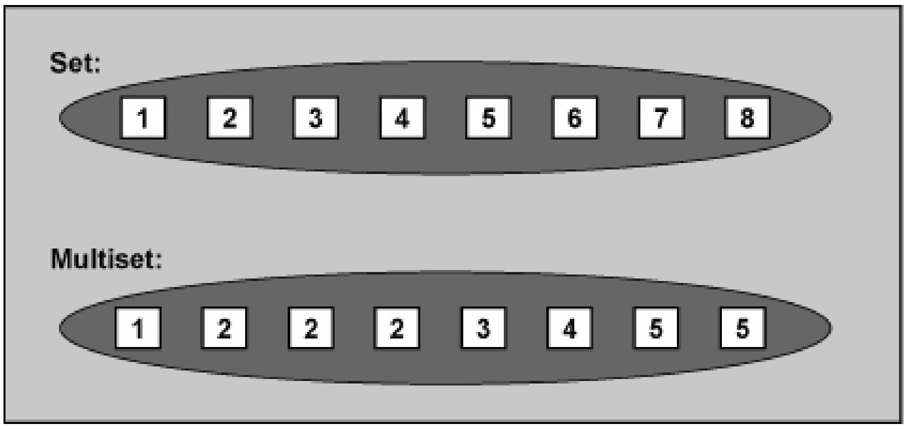


Figure 7.12. Sets and Multisets

To use a set or a multiset, you must include the header file `<set>`:

```
#include <set>
```

There, the types are defined as class templates inside namespace `std`:

```
namespace std {
    template <typename T,
              typename Compare = less<T>,
              typename Allocator = allocator<T> >
             class set;

    template <typename T,
              typename Compare = less<T>,
              typename Allocator = allocator<T> >
             class multiset;
}
```

The elements of a set or a multiset may have any type `T` that is comparable according to the sorting criterion. The optional second template argument defines the sorting criterion. If a special sorting criterion is not passed, the default criterion `less` is used. The function object `less` sorts the elements by comparing them with operator `<` (see Section 10.2.1, page 487, for details about `less`). The optional third template parameter defines the memory model (see Chapter 19). The default memory model is the model `allocator`, which is provided by the C++ standard library.

The sorting criterion must define *strict weak ordering*, which is defined by the following four properties:

1. It has to be **antisymmetric**.

   This means that for operator `<`: If `x < y` is `true`, then `y < x` is `false`.

   This means that for a predicate `op()`: If `op(x,y)` is `true`, then `op(y,x)` is `false`.

2. It has to be **transitive**.

   This means that for operator `<`: If `x < y` is `true` and `y< z` is `true`, then `x < z` is `true`.

   This means that for a predicate `op()`: If `op(x,y)` is `true` and `op(y,z)` is `true`, then `op(x,z)` is `true`.

3. It has to be **irreflexive**.

   This means that for operator `<`: `x < x` is always `false`.

   This means that for a predicate `op()`: `op(x,x)` is always `false`.

4. It has to have **transitivity of equivalence**, which means roughly: If `a` is equivalent to `b` and `b` is equivalent to

`c`, then `a` is equivalent to `c`.

This means that for operator `<`: If `!(a<b) && !(b<a)` is `true` and `!(b<c) && !(c<b)` is `true` then `!(a<c) && !(c<a)` is `true`.

This means that for a predicate `op()`: If `op(a,b)`, `op(b,a)`, `op(b,c)`, and `op(c,b)` all yield `false`, then `op(a,c)` and `op(c,a)` yield `false`.

Note that this means that you have to distinguish between less and equal. A criterion such as operator `<=` does not fulfill this requirement.

Based on these properties, the sorting criterion is also used to check equivalence. That is, two elements are considered to be duplicates if neither is less than the other (or if both `op(x,y)` and `op(y,x)` are `false`).

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

7.7.1. Abilities of Sets and Multisets

Like all standardized associative container classes, sets and multisets are usually implemented as balanced binary trees (Figure 7.13). The standard does not specify this, but it follows from the complexity of set and multiset operations.[9]

[9] In fact, sets and multisets are typically implemented as *red-black trees*, which are good for both changing the number of elements and searching for elements. They guarantee at most two internal relinks on insertions and that the longest path is at most twice as long as the shortest path to a leaf.

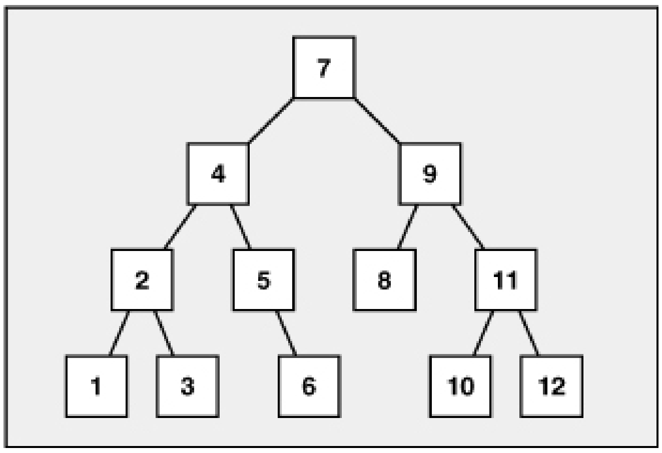


Figure 7.13. Internal Structure of Sets and Multisets

The major advantage of automatic sorting is that a binary tree performs well when searching for elements with a specific value. In fact, search functions have logarithmic complexity. For example, to search for an element in a set or a multiset of 1,000 elements, a tree search performed by a member function needs, on average, one-fiftieth of the comparisons of a linear search (which is performed by a search algorithm that iterates over all elements). See Section 2.2, page 10, for more details about complexity.

However, automatic sorting also imposes an important constraint on sets and multisets: You may *not* change the value of an element directly, because doing so might compromise the correct order.

Therefore, to modify the value of an element, you must remove the element having the old value and insert a new element that has the new value. The interface reflects this behavior:

- Sets and multisets don't provide operations for direct element access.
- Indirect access via iterators has the constraint that, from the iterator's point of view, the element value is constant.

7.7.2. Set and Multiset Operations

Create, Copy, and Destroy

Table 7.34 lists the constructors and destructors of sets and multisets.

Table 7.34. Constructors and Destructors of Sets and Multisets

set	**Effect**
`set<`*`Elem`*`>`	A set that by default sorts with `less<>` (operator `<`)
`set<`*`Elem`*`,`*`Op`*`>`	A set that by default sorts with `Op`
`multiset<`*`Elem`*`>`	A multiset that by default sorts with `less<>` (operator `<`)
`multiset<`*`Elem`*`,`*`Op`*`>`	A multiset that by default sorts with *`Op`*

You can define the sorting criterion in two ways:

1. **As a template parameter.** For example:

```
std::set<int,std::greater<int>> coll;
```

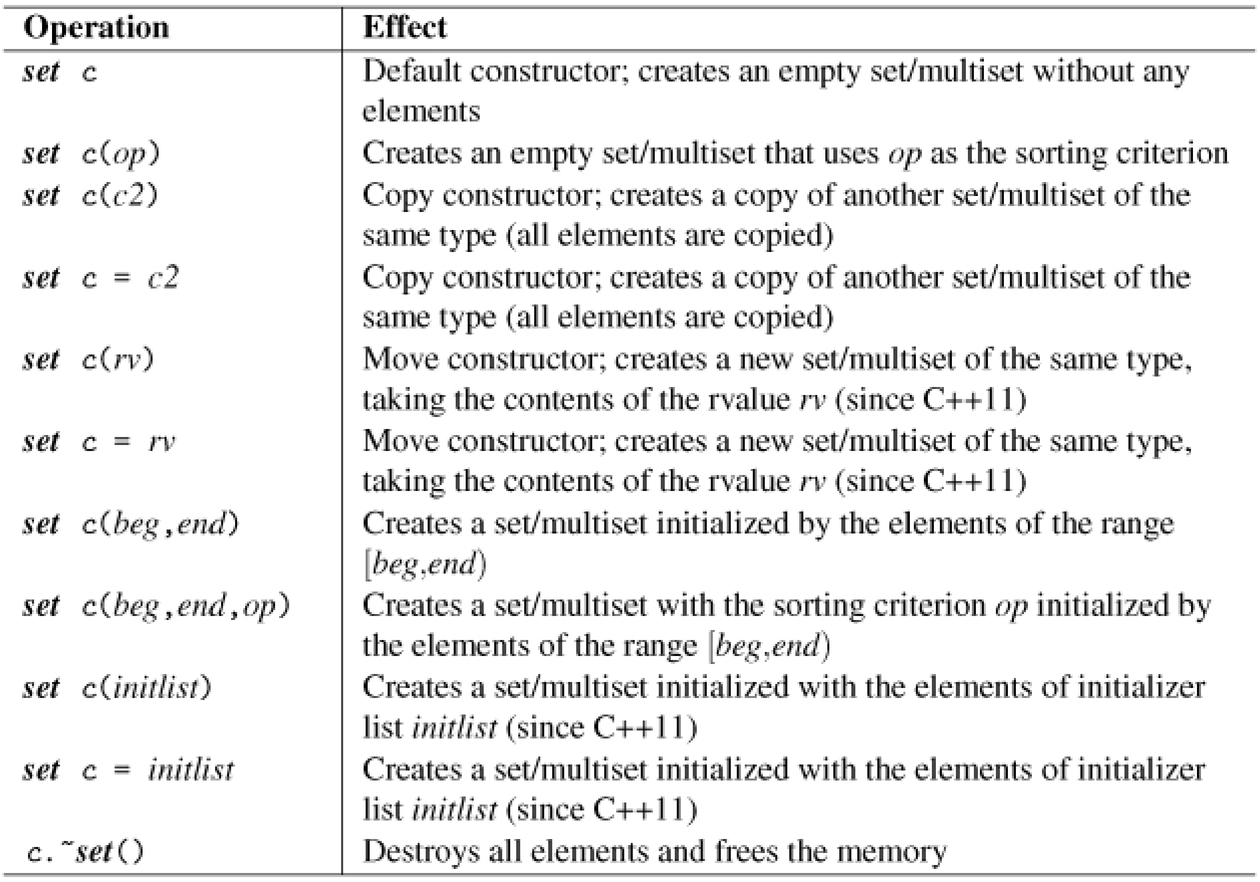
   In this case, the sorting criterion is part of the type. Thus, the type system ensures that only containers with the same sorting criterion can be combined. This is the usual way to specify the sorting criterion. To be more precise, the second parameter is the *type* of the sorting criterion. The concrete sorting criterion is the function object that gets created with the container. To do this, the constructor of the container calls the default constructor of the type of the sorting criterion. See Section 10.1.1, page 476, for an example that uses a user-defined sorting criterion.

2. **As a constructor parameter.** In this case, you might have a type for several sorting criteria that allows having different initial values or states. This is useful when processing the sorting criterion at runtime and when sorting criteria are needed that are different but of the same data type. See Section 7.7.5, page 328, for a complete example.

If no special sorting criterion is passed, the default sorting criterion, function object `less<>`, is used, which sorts the elements by using operator `<`.

Note that the sorting criterion is also used to check for equivalence of two elements in the same container (i.e., to find duplicates). Thus, when the default sorting criterion is used, the check for equivalence of two elements looks like this:

```
if (! (elem1<elem2 || elem2<elem1))
```



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

Here, ***set*** may be one of the following types:

This has three advantages:

1. You need to pass only one argument as the sorting criterion.
2. You don't have to provide operator `==` for the element type.
3. You can have contrary definitions between equivalence and equality (however, this might be a source of confusion).

Checking for equivalence in this way takes a bit more time because two comparisons might be necessary to evaluate the previous expression. Note, however, that if the result of the first comparison yields `true`, the second comparison is not evaluated.

Note also that if two containers are compared by operator `==`, the elements in both containers are compared using their operator `==`, which means that operator `==` has to be provided for the element type.

The constructor for the beginning and the end of a range could be used to initialize the container with elements from containers that have other types, from arrays, or from the standard input. See Section 7.1.2, page 254, for details.

Nonmodifying Operations

Sets and multisets provide the usual nonmodifying operations to query the size and to make comparisons (Table 7.35).

Table 7.35. Nonmodifying Operations of Sets and Multisets

Operation	Effect
`c.key_comp()`	Returns the comparison criterion
`c.value_comp()`	Returns the comparison criterion for values as a whole (same as `key_comp()`)
`c.empty()`	Returns whether the container is empty (equivalent to `size()==0` but might be faster)
`c.size()`	Returns the current number of elements
`c.max_size()`	Returns the maximum number of elements possible
c1 `==` *c2*	Returns whether *c1* is equal to *c2* (calls `==` for the elements)
c1 `!=` *c2*	Returns whether *c1* is not equal to *c2* (equivalent to `!(`*c1*`==`*c2*`)`)
c1 `<` *c2*	Returns whether *c1* is less than *c2*
c1 `>` *c2*	Returns whether *c1* is greater than *c2* (equivalent to *c2*`<`*c1*)
c1 `<=` *c2*	Returns whether *c1* is less than or equal to *c2* (equivalent to `!(`*c2*`<`*c1*`)`)
c1 `>=` *c2*	Returns whether *c1* is greater than or equal to *c2* (equivalent to `!(`*c1*`<`*c2*`)`)

Comparisons are provided only for containers of the same type. Thus, the elements *and* the sorting criterion must have the same types; otherwise, a type error occurs at compile time. For example:

```
std::set<float> c1;          // sorting criterion: std::less<>
std::set<float,std::greater<float> > c2;
...
if (c1 == c2) {              // ERROR: different types
    ...
}
```

The check whether a container is less than another container is done by a lexicographical comparison (see Section 11.5.4, page 548). To compare containers of different types (different sorting criteria), you must use the comparing algorithms in Section 11.5.4, page 542.

Special Search Operations

Because they are optimized for fast searching of elements, sets and multisets provide special search functions (Table 7.36). These functions are special versions of general algorithms that have the same name. You should always prefer the optimized versions for sets and multisets to achieve logarithmic complexity instead of the linear complexity of the general algorithms. For example, a search of a collection of 1,000 elements requires on average only 10 comparisons instead of 500 (see Section 2.2, page 10).

Table 7.36. Special Search Operations of Sets and Multisets

Operation	Effect
`c.count(`*val*`)`	Returns the number of elements with value *val*
`c.find(`*val*`)`	Returns the position of the first element with value *val* (or `end()` if none found)
`c.lower_bound(`*val*`)`	Returns the first position, where *val* would get inserted (the first element >= *val*)
`c.upper_bound(`*val*`)`	Returns the last position, where *val* would get inserted (the first element > *val*)
`c.equal_range(`*val*`)`	Returns a range with all elements with a value equal to *val* (i.e., the first and last position, where *val* would get inserted)

The `find()` member function searches for the first element that has the value that was passed as the argument and returns its iterator position. If no such element is found, `find()` returns `end()` of the container.

`lower_bound()` and `upper_bound()` return the first and last position, respectively, at which an element with the passed value would be inserted. In other words, `lower_bound()` returns the position of the first element that has the same or a greater value than the argument, whereas `upper_bound()` returns the position of the first element with a greater value. `equal_range()` returns both return values of `lower_bound()` and `upper_bound()` as a `pair` (type `pair` is introduced in Section 5.1.1, page 60). Thus, `equal_range()` returns the range of elements that have the same value as the argument. If `lower_bound()` or the first value of `equal_range()` is equal to `upper_bound()` or the second value of `equal_range()`, no elements with the same value exist in the set or multiset. Naturally, the range of elements having the same values could contain at most one element in a set.

The following example shows how to use `lower_bound()`, `upper_bound()`, and `equal_range()`:

Click here to view code image

```
// cont/setrange1.cpp

#include <iostream>
#include <set>
using namespace std;
int main ()
{
    set<int> c;

    c.insert(1);
    c.insert(2);
    c.insert(4);
    c.insert(5);
    c.insert(6);

    cout << "lower_bound(3): " << *c.lower_bound(3) << endl;
    cout << "upper_bound(3): " << *c.upper_bound(3) << endl;
    cout << "equal_range(3): " << *c.equal_range(3).first << " "
                               << *c.equal_range(3).second << endl;
    cout << endl;
    cout << "lower_bound(5): " << *c.lower_bound(5) << endl;
    cout << "upper_bound(5): " << *c.upper_bound(5) << endl;
    cout << "equal_range(5): " << *c.equal_range(5).first << " "
                               << *c.equal_range(5).second << endl;
}
```

The output of the program is as follows:

```
lower_bound(3): 4
upper_bound(3): 4
equal_range(3): 4 4

lower_bound(5): 5
upper_bound(5): 6
equal_range(5): 5 6
```

If you use a multiset instead of a set, the program has the same output.

Assignments

As listed in Table 7.37, Sets and multisets provide only the fundamental assignment operations that all containers provide (see Section 7.1.2, page 258).

Table 7.37. Assignment Operations of Sets and Multisets

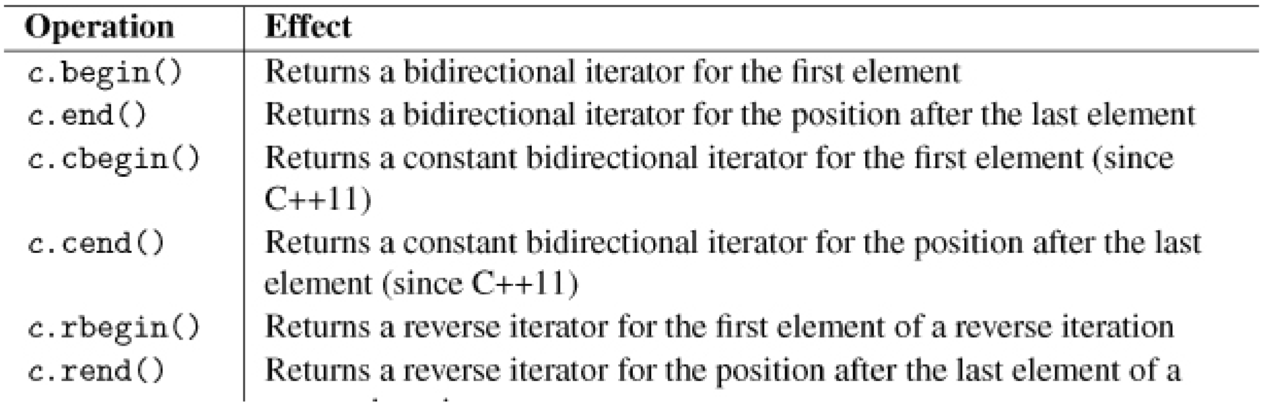
Operation	Effect
`c = c2`	Assigns all elements of *c2* to `c`
`c = rv`	Move assigns all elements of the rvalue *rv* to `c` (since C++11)
`c = initlist`	Assigns all elements of the initializer list *initlist* to `c` (since C++11)
`c1.swap(c2)`	Swaps the data of `c1` and `c2`
`swap(c1,c2)`	Swaps the data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

For these operations, both containers must have the same type. In particular, the type of the comparison criteria must be the same, although the comparison criteria themselves may be different. See Section 7.7.5, page 328, for an example of different sorting criteria that have the same type. If the criteria are different, they will also get assigned or swapped.

Iterator Functions

Sets and multisets do not provide direct element access, so you have to use range-based `for` loops (see Section 3.1.4, page 17) or iterators. Sets and multisets provide the usual member functions for iterators (Table 7.38).

Table 7.38. Iterator Operations of Sets and Multisets



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

As with all associative container classes, the iterators are bidirectional iterators (see Section 9.2.4, page 437). Thus, you can't use them in algorithms that are provided only for random-access iterators, such as algorithms for sorting or random shuffling.

More important is the constraint that, from an iterator's point of view, all elements are considered constant. This is necessary to ensure that you can't compromise the order of the elements by changing their values. However, as a result, you can't call any modifying algorithm on the elements of a set or a multiset. For example, you can't call the `remove()` algorithm, because it "removes" by overwriting "removed" elements with the following elements (see Section 6.7.2, page 221, for a detailed discussion of this problem). To remove elements in sets and multisets, you can use only member functions provided by the container.

Inserting and Removing Elements

Table 7.39 shows the operations provided for sets and multisets to insert and remove elements.

Table 7.39. Insert and Remove Operations of Sets and Multisets

As usual when using the STL, you must ensure that the arguments are valid. Iterators must refer to valid positions, and the beginning of a range must have a position that is not behind the end.

Inserting and removing is faster if, when working with multiple elements, you use a single call for all elements rather than multiple calls.

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

Note that the return types of the inserting functions `insert()` and `emplace()` differ as follows:

- **Sets** provide the following interface:[10]

[10] Before C++11, only `insert()` was provided, and *posHint* had type `iterator` instead of `const_iterator`.

Click here to view code image

```
pair<iterator,bool>   insert  (const value_type& val);
iterator              insert  (const_iterator posHint,
                               const value_type& val);
template <typename... Args>
 pair<iterator, bool> emplace (Args&&... args);
template <typename... Args>
 iterator             emplace_hint (const_iterator posHint,
                                    Args&&... args);
```

- **Multisets** provide the following interface:[10]

Click here to view code image

```
iterator              insert  (const value_type& val);
iterator              insert  (const_iterator posHint,
                               const value_type& val);
template <typename... Args>
 iterator             emplace (Args&&... args);
template <typename... Args>
 iterator             emplace_hint (const_iterator posHint,
                                    Args&&... args);
```

The difference in return types results because multisets allow duplicates, whereas sets do not. Thus, the insertion of an element might fail for a set if it already contains an element with the same value. Therefore, the return type for a set returns two values by using a `pair` structure (`pair` is discussed in Section 5.1.1, page 60):

1. The member `second` of the `pair` structure returns whether the insertion was successful.
2. The member `first` of the `pair` structure returns the position of the newly inserted element or the position of the still existing element.

In all other cases, the functions return the position of the new element or of the existing element if the set already contains an element with the same value.

The following example shows how to use this interface to insert a new element into a set. It tries to insert the element with value `3.3` into the set `c` :

```
std::set<double> c;
...
if (c.insert(3.3).second) {
    std::cout << "3.3 inserted" << std::endl;
}
else {
    std::cout << "3.3 already exists" << std::endl;
}
```

If you also want to process the new or old positions, the code gets more complicated:

Click here to view code image

```
// insert value and process return value
auto status = c.insert(value);
if (status.second) {
    std::cout << value << " inserted as element "
}
else {
    std::cout << value << " already exists as element "
}
std::cout << std::distance(c.begin(),status.first) + 1 << std::endl;
```

The output of two calls of this sequence might be as follows:

```
8.9 inserted as element 4
7.7 already exists as element 3
```

In this example, the type of `status` is as follows:

```
std::pair<std::set<float>::iterator,bool>
```

Note that the return types of the insert functions with an additional position parameter don't differ. These functions return a single iterator for both sets and multisets. However, these functions have the same effect as the functions without the position parameter. They differ only in their performance. You can pass an iterator position, but this position is processed as a hint to optimize performance. In fact, if the element gets inserted right after the position that is passed as the first argument, the time complexity changes from logarithmic to amortized constant (complexity is discussed in Section 2.2, page 10). The fact that all the insert functions with the additional position hint share the same return type has a good reason. It enables generic code that can insert elements to any container (except for `arrays` and forward lists). In fact, this interface is used by general inserters. See Section 9.4.2, especially page 458, for details about inserters.

To remove an element that has a certain value, you simply call `erase()` :

```
std::multiset<Elem> coll;
...
// remove all elements with passed value
coll.erase(value);
```

Note that this member function has a different name than `remove()` provided for lists (see Section 7.5.2, page 294, for a discussion of `remove()`). It behaves differently in that it returns the number of removed elements. When called for sets, it returns only `0` or `1` .

If a multiset contains duplicates, you can't use `erase()` to remove only the first element of these duplicates. Instead, you can code as follows:

```
std::multiset<Elem> coll;
...
// remove first element with passed value
std::multiset<Elem>::iterator pos;
pos = coll.find(value);
if (pos != coll.end()) {
    coll.erase(pos);
}
```

Because it is faster, you should use the member function `find()` instead of the `find()` algorithm here.

Note that before C++11, the `erase()` functions of associative containers returned nothing (had return type `void`). The reason was performance. It might cost time to find and return the successor in an associative container, because the container is implemented as a binary tree. However, this greatly complicated code where you erase elements while iterating over them (see Section 7.8.2, page 342).

Note also that for sets that use iterators as elements, calling `erase()` might be ambiguous now. For this reason, C++11 gets fixed to provide overloads for both `erase(iterator)` and `erase(const_iterator)` .

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

7.7.3. Exception Handling

Sets and multisets are node-based containers, so any failure to construct a node simply leaves the container as it was. Furthermore, because destructors in general don't throw, removing a node can't fail.

However, for multiple-element insert operations, the need to keep elements sorted makes full recovery from throws impractical. Thus, all single-element insert operations support commit-or-rollback behavior. That is, they either succeed or have no effect. In addition, it is guaranteed that all multiple-element delete operations always succeed or have no effect, provided that the comparison criterion does not throw. If copying/assigning the comparison criterion may throw, `swap()` may throw.

See Section 6.12.2, page 248, for a general discussion of exception handling in the STL.

7.7.4. Examples of Using Sets and Multisets

The following program demonstrates some abilities of sets:

Click here to view code image

```
// cont/set1.cpp

#include <iostream>
#include <set>
#include <algorithm>
#include <iterator>
using namespace std;

int main()
{
    // type of the collection:
    // - no duplicates
    // - elements are integral values
    // - descending order
    set<int,greater<int>> coll1;

    // insert elements in random order using different member functions
    coll1.insert({4,3,5,1,6,2});
    coll1.insert(5);

    // print all elements
    for (int elem : coll1) {
        cout << elem << ' ';
    }
    cout << endl;

    // insert 4 again and process return value
    auto status = coll1.insert(4);
    if (status.second) {
        cout << "4 inserted as element "
             << distance(coll1.begin(),status.first) + 1 << endl;
    }
    else {
        cout << "4 already exists" << endl;
    }

    // assign elements to another set with ascending order
    set<int> coll2(coll1.cbegin(),coll1.cend());

    // print all elements of the copy using stream iterators
    copy (coll2.cbegin(), coll2.cend(),
          ostream_iterator<int>(cout," "));
    cout << endl;

    // remove all elements up to element with value 3
    coll2.erase (coll2.begin(), coll2.find(3));
```

```
        // remove all elements with value 5
        int num;
        num = coll2.erase (5);
        cout << num << " element(s) removed" << endl;

        // print all elements
        copy (coll2.cbegin(), coll2.cend(),
              ostream_iterator<int>(cout," "));
        cout << endl;
    }
```

At first, an empty set is created and several elements are inserted by using different overloads of `insert()` :

```
set<int,greater<int>> coll1;

coll1.insert({4,3,5,1,6,2});
coll1.insert(5);
```

Note that the element with value `5` is inserted twice. However, the second insertion is ignored because sets do not allow duplicates.

After printing all elements, the program tries again to insert the element `4` . This time, it processes the return values of `insert()` as discussed in Section 7.7.2, page 323.

The statement

```
set<int> coll2(coll1.cbegin(),coll1.cend());
```

creates a new set of `int` s with ascending order and initializes it with the elements of the old set.

Both containers have different sorting criteria, so their types differ, and you can't assign or compare them directly. However, you can use algorithms, which in general are able to handle different container types as long as the element types are equal or convertible.

The following statement removes all elements up to the element with value `3` :

```
coll2.erase (coll2.begin(), coll2.find(3));
```

Note that the element with value `3` is the end of the range, so it is not removed.

Finally, all elements with value `5` are removed:

```
int num;
num = coll2.erase (5);
cout << num << " element(s) removed" << endl;
```

The output of the whole program is as follows:

```
6 5 4 3 2 1
4 already exists
1 2 3 4 5 6
1 element(s) removed
3 4 6
```

For multisets, the same program (provided in *cont/multiset1.cpp*) looks a bit different and produces different results. First, in all cases type `set` has to get replaced by `multiset` (the header file remains the same):

```
multiset<int,greater<int>> coll1;
...
multiset<int> coll2(coll1.cbegin(),coll1.cend());
```

In addition, the processing of the return value of `insert()` looks different. Sets allow no duplicates, so `insert()` returns both the new position of the inserted element and whether the insertion was successful:

```
auto status = coll1.insert(4);
if (status.second) {
    cout << "4 inserted as element "
         << distance(coll1.begin(),status.first) + 1 << endl;
}
else {
    cout << "4 already exists" << endl;
}
```

For multisets, `insert()` only returns the new position (because multisets may contain duplicates, the insertion can fail only if an exception gets thrown):

```
auto ipos = coll1.insert(4);
```

```
    cout << "4 inserted as element "
         << distance(coll1.begin(),ipos) + 1 << endl;
```

The output of the program changes as follows:

```
    6 5 4 3 2 1
    4 already exists
    1 2 3 4 5 6
    1 element(s) removed
    3 4 6
```

7.7.5. Example of Specifying the Sorting Criterion at Runtime

Normally, you define the sorting criterion as part of the type, by either passing it as a second template argument or using the default sorting criterion `less<>`. Sometimes, however, you must process the sorting criterion at runtime, or you may need different sorting criteria with the same data type. In such cases, you need a special type for the sorting criterion: one that lets you pass your sorting details at runtime. The following example program demonstrates how to do this:[11]

[11] Thanks to Daniel Krügler for details of this example.

Click here to view code image

```
    // cont/setcmp1.cpp

    #include <iostream>
    #include <set>
    #include "print.hpp"
    using namespace std;

    // type for runtime sorting criterion
    class RuntimeCmp {
      public:
        enum cmp_mode {normal, reverse};
      private:
        cmp_mode mode;
      public:
        // constructor for sorting criterion
        // - default criterion uses value normal
        RuntimeCmp (cmp_mode m=normal) : mode(m) {
        }
        // comparison of elements
        // - member function for any element type
        template <typename T>
        bool operator() (const T& t1, const T& t2) const {
            return mode==normal ?  t1<t2
                                :  t2<t1;
        }
        // comparison of sorting criteria
        bool operator== (const RuntimeCmp& rc) const {
            return mode == rc.mode;
        }
    };

    // type of a set that uses this sorting criterion
    typedef set<int,RuntimeCmp> IntSet;

    int main()
    {
        // create, fill, and print set with normal element order
        // - uses default sorting criterion
        IntSet coll1 = { 4, 7, 5, 1, 6, 2, 5 };
        PRINT_ELEMENTS (coll1, "coll1: ");

        // create sorting criterion with reverse element order
        RuntimeCmp reverse_order(RuntimeCmp::reverse);

        // create, fill, and print set with reverse element order
        IntSet coll2(reverse_order);
        coll2 = { 4, 7, 5, 1, 6, 2, 5 };
        PRINT_ELEMENTS (coll2, "coll2: ");

        // assign elements AND sorting criterion
        coll1 = coll2;
        coll1.insert(3);
        PRINT_ELEMENTS (coll1, "coll1: ");

        // just to make sure...
        if (coll1.value_comp() == coll2.value_comp()) {
            cout << "coll1 and coll2 have the same sorting criterion"
                 << endl;
        }
        else {
            cout << "coll1 and coll2 have a different sorting criterion"
                 << endl;
        }
    }
```

In this program, the class `RuntimeCmp` provides the general ability to specify, at runtime, the sorting criterion for any type. Its default constructor sorts in ascending order, using the default value `normal`. It also is possible to pass `RuntimeCmp::reverse` to sort in descending order.

The output of the program is as follows:

```
coll1: 1 2 4 5 6 7
coll2: 7 6 5 4 2 1
coll1: 7 6 5 4 3 2 1
coll1 and coll2 have the same sorting criterion
```

Note that `coll1` and `coll2` have the same type, which is not the case when passing `less<>` and `greater<>` as sorting criteria. Note also that the assignment operator assigns the elements *and* the sorting criterion; otherwise, an assignment would be an easy way to compromise the sorting criterion.