7.8. Maps and Multimaps

Maps and multimaps are containers that manage key/value pairs as elements. These containers sort their elements automatically, according to a certain sorting criterion that is used for the key. The difference between the two is that multimaps allow duplicates, whereas maps do not (Figure 7.14).

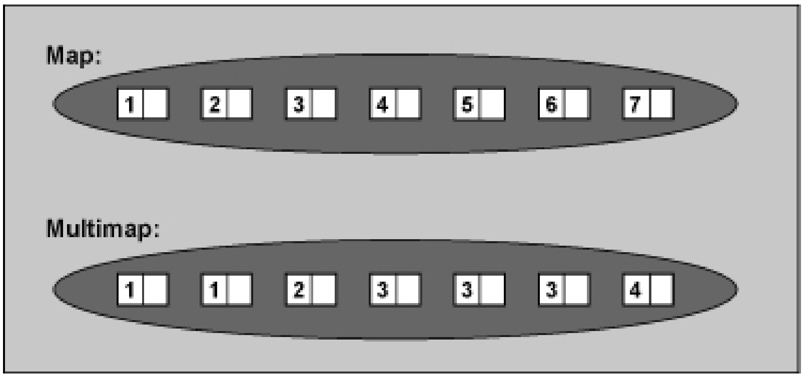

Figure 7.14. Maps and Multimaps

To use a map or a multimap, you must include the header file `<map>`:

```
#include <map>
```

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

Click here to view code image

```
namespace std {
    template <typename Key, typename T,
              typename Compare = less<Key>,
              typename Allocator = allocator<pair<const Key,T> > >
             class map;

    template <typename Key, typename T,
              typename Compare = less<Key>,
              typename Allocator = allocator<pair<const Key,T> > >
             class multimap;
}
```

The first template parameter is the type of the element's key, and the second template parameter is the type of the element's associated value. The elements of a map or a multimap may have any types `Key` and `T` that meet the following two requirements:

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

Note that the element type (`value_type`) is a `pair <const` *Key*, *T* `>`.

The optional third template parameter defines the sorting criterion. As for sets, this sorting criterion must define a "strict weak ordering" (see Section 7.7, page 314). The elements are sorted according to their keys, so the value doesn't matter for the order of the elements. The sorting criterion is also used to check for equivalence; that is, two elements are equal if neither key is less than the other.

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`).

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

The optional fourth 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.

7.8.1. Abilities of Maps and Multimaps

Like all standardized associative container classes, maps and multimaps are usually implemented as balanced binary trees (Figure 7.15). The standard does not specify this, but it follows from the complexity of the map and multimap operations. In fact, sets, multisets, maps, and multimaps typically use the same internal data type. So, you could consider sets and multisets as special maps and multimaps, respectively, for which the value and the key of the elements are the same objects. Thus, maps and multimaps have all the abilities and operations of sets and multisets. Some minor differences exist, however. First, their elements are key/value pairs. In addition, maps can be used as associative arrays.

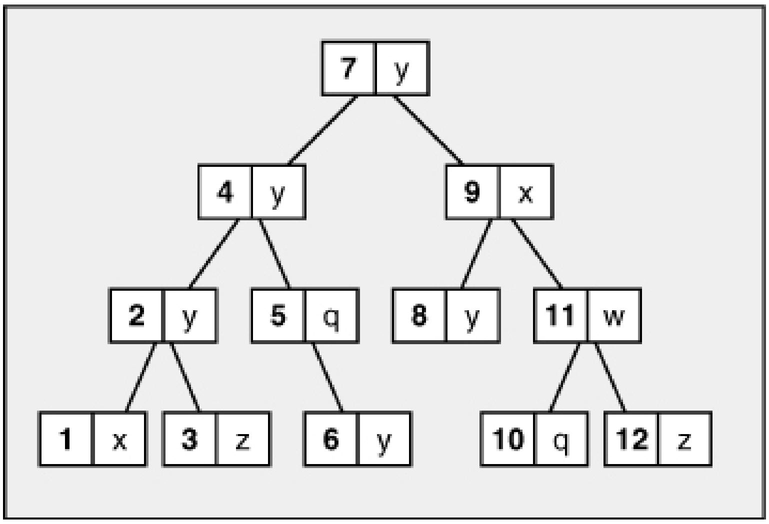

Figure 7.15. Internal Structure of Maps and Multimaps

Maps and multimaps sort their elements automatically, according to the element's keys, and so have good performance when searching for elements that have a certain key. Searching for elements that have a certain value promotes bad performance. Automatic sorting imposes an important constraint on maps and multimaps: You may *not* change the key of an element directly, because doing so might compromise the correct order. To modify the key of an element, you must remove the element that has the old key and insert a new element that has the new key and the old value (see Section 7.8.2, page 339, for details). As a consequence, from the iterator's point of view, the element's key is constant. However, a direct modification of the value of the element is still possible, provided that the type of the value is not constant.

7.8.2. Map and Multimap Operations

Create, Copy, and Destroy

Table 7.40 lists the constructors and destructors of maps and multimaps.

Table 7.40. Constructors and Destructors of Maps and Multimaps

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

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

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

You can define the sorting criterion in two ways:

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

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

   In this case, the sorting criterion is part of the type. Thus, the type system ensures that only containers with the same sorting criterion can be combined. This is the usual way to specify the sorting criterion. To be more precise, the third 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, and the initial value or state of the sorting criteria might differ. This is useful when processing the sorting criterion at runtime or when sorting criteria are needed that are different but of the same data type. A typical example is specifying the sorting criterion for string keys at runtime. See Section 7.8.6, for a complete example.

If no special sorting criterion is passed, the default sorting criterion, function object `less<>`, is used, which sorts the elements according to their key by using operator `<`. Again, the sorting criterion is also used to check for equivalence of two elements in the same container (i.e., to find duplicates). Only to compare two containers is operator `==` required.

You might prefer a type definition to avoid the boring repetition of the type whenever it is used:

```
typedef std::map<std::string,float,std::greater<std::string>>
        StringFloatMap;
...
StringFloatMap coll;
```

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. However, the elements are key/value pairs, so you must ensure that the elements from the source range have or are convertible into type `pair<`*key*`,`*value*`>`.

Nonmodifying and Special Search Operations

Maps and multimaps provide the usual nonmodifying operations: those that query size aspects and make comparisons (Table 7.41).

Table 7.41. Nonmodifying Operations of Maps and Multimaps

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

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

Click here to view code image

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

Checking 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 criterion), you must use the comparing algorithms of Section 11.5.4, page 542.

Special Search Operations

As for sets and multisets, maps and multimaps provide special search member functions that perform better because of their internal tree structure (Table 7.42).

Table 7.42. Special Search Operations of Maps and Multimaps

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

The `find()` member function searches for the first element that has the appropriate key and returns its iterator position. If no such element is found, `find()` returns `end()` of the container. You can't use the `find()` member function to search for an element that has a certain value. Instead, you have to use a general algorithm, such as the `find_if()` algorithm, or program an explicit loop. Here is an example of a simple loop that does something with each element that has a certain value:

```
std::multimap<std::string,float> coll;
...
// do something with all elements having a certain value
std::multimap<std::string,float>::iterator pos;
for (pos = coll.begin(); pos != coll.end(); ++pos) {
    if (pos->second == value) {
        do_something(pos);
    }
}
```

Be careful when you want to use such a loop to remove elements. It might happen that you saw off the branch on which you are sitting. See Section 7.8.2, page 342, for details about this issue.

Using the `find_if()` algorithm to search for an element that has a certain value is even more complicated than writing a loop, because you have to provide a function object that compares the value of an element with a certain value. See Section 7.8.5, page 350, for an example.

The `lower_bound()`, `upper_bound()`, and `equal_range()` functions behave as they do for sets (see Section 7.7.2, page 319), except that the elements are key/value pairs.

Assignments

As listed in Table 7.43, maps and multimaps provide only the fundamental assignment operations that all containers provide (see Section 7.1.2, page 258).

Table 7.43. Assignment Operations of Maps and Multimaps

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 7.8.6, page 351, for an example of different sorting criteria that have the same type. If the criteria are different, they also get assigned or swapped.

Iterator Functions and Element Access

Maps and multimaps do not provide direct element access, so the usual way to access elements is via range-based `for` loops (see Section 3.1.4, page 17) or iterators. An exception to that rule is that maps provide `at()` and the subscript operator to access elements directly (see Section 7.8.3, page 343). Table 7.44 lists the usual member functions for iterators that maps and multimaps provide.

Table 7.44. Iterator Operations of Maps and Multimaps

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

As for all associative container classes, the iterators are bidirectional (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 the key of all elements inside a map and a multimap is considered to be constant. Thus, the type of the elements is `pair<const Key, T>`. This is necessary to ensure that you can't compromise the order of the elements by changing their keys. However, you can't call any modifying algorithm if the destination is a map or a multimap. 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 maps and multimaps, you can use only member functions provided by the container.

The following is an example of element access via use range-based `for` loops:

Click here to view code image

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

```
...
for (auto& elem : coll) {
    std::cout << "key: "   << elem.first  << "\t"
              << "value: " << elem.second << std::endl;
}
```

Inside the loop, `elem` becomes a reference referring to the actual element of the container `coll` currently processed. Thus, `elem` has type `pair<const std::string,float>`. The expression `elem.first` yields the key of the actual element, whereas the expression `elem.second` yields the value of the actual element.

The corresponding code using iterators, which has to be used before C++11, looks as follows:

Click here to view code image

```
std::map<std::string,float> coll;
...
std::map<std::string,float>::iterator pos;
for (pos = coll.begin(); pos != coll.end(); ++pos) {
    std::cout << "key: "   << pos->first  << "\t"
              << "value: " << pos->second << std::endl;
}
```

Here, the iterator `pos` iterates through the sequence of pairs of `const string` and `float`, and you have to use operator `->` to access key and value of the actual element.[12]

[12] `pos->first` is a shortcut for `(*pos).first`.

Trying to change the value of the key results in an error:

```
elem.first = "hello";    // ERROR at compile time
pos->first = "hello";    // ERROR at compile time
```

However, changing the value of the element is no problem, as long as `elem` is declared as a nonconstant reference and the type of the value is not constant:

```
elem.second = 13.5;      // OK
pos->second = 13.5;      // OK
```

If you use algorithms and lambdas to operate with the elements of a map, you explicitly have to declare the element type:

```
std::map<std::string,float> coll;
...
// add 10 to the value of each element:
std::for_each (coll.begin(), coll.end(),
               [] (std::pair<const std::string,float>& elem) {
                    elem.second += 10;
               });
```

Instead of using the following:

```
std::pair<const std::string,float>
```

you could use

```
std::map<std::string,float>::value_type
```

or

```
decltype(coll)::value_type
```

to declare the type of an element. See Section 7.8.5, page 345, for a complete example.

To change the key of an element, you have only one choice: You must replace the old element with a new element that has the same value. Here is a generic function that does this:

Click here to view code image

```
// cont/newkey.hpp

namespace MyLib {
    template <typename Cont>
    inline
    bool replace_key (Cont& c,
                      const typename Cont::key_type& old_key,
                      const typename Cont::key_type& new_key)
    {
         typename Cont::iterator pos;
```

```
            pos = c.find(old_key);
            if (pos != c.end()) {
                // insert new element with value of old element
                c.insert(typename Cont::value_type(new_key,
                                                   pos->second));
                // remove old element
                c.erase(pos);
                return true;
            }
            else {
                // key not found
                return false;
            }
        }
    }
```

The `insert()` and `erase()` member functions are discussed in the next subsection.

To use this generic function, you simply pass the container, the old key, and the new key. For example:

```
std::map<std::string,float> coll;
...
MyLib::replace_key(coll,"old key","new key");
```

It works the same way for multimaps (replacing the *first* matching key).

Note that maps provide a more convenient way to modify the key of an element. Instead of calling `replace_key()`, you can simply write the following:

```
// insert new element with value of old element
coll["new_key"] = coll["old_key"];
// remove old element
coll.erase("old_key");
```

See Section 7.8.3, page 343, for details about the use of the subscript operator with maps.

Inserting and Removing Elements

Table 7.45 shows the operations provided for maps and multimaps to insert and remove elements. The remarks in Section 7.7.2, page 322, regarding sets and multisets apply here. In particular, the return types of these operations have the same differences as they do for sets and multisets. However, note that the elements here are key/value pairs. So, the use is getting a bit more complicated.

Table 7.45. Insert and Remove Operations of Maps and Multimaps

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

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

To insert a key/value pair, you must keep in mind that inside maps and multimaps, the key is considered to be constant. You must provide either the correct type or you need to provide implicit or explicit type conversions.

Since C++11, the most convenient way to insert elements is to use `emplace()` or to pass them to `insert()` as initializer list, where the first entry is the key and the second entry is the value:

```
std::map<std::string,float> coll;
...
coll.emplace("jim",17.7);   // see below, if key/value need more args for initialization
coll.insert({"otto",22.3});
```

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

1. **Use `value_type`**. To avoid implicit type conversion, you could pass the correct type explicitly by using `value_type`, which is provided as a type definition by the container type. For example:

   [Click here to view code image]

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

   or

   ```
   coll.insert(decltype(coll)::value_type("otto",22.3));
   ```

2. **Use `pair<>`**. Another way is to use `pair<>` directly. For example:

   [Click here to view code image]

   ```
   std::map<std::string,float> coll;
   ...
   // use implicit conversion:
   coll.insert(std::pair<std::string,float>("otto",22.3));
   // use no implicit conversion:
   coll.insert(std::pair<const std::string,float>("otto",22.3));
   ```

   In the first `insert()` statement, the type is not quite right, so it is converted into the real element type. For this to happen, the `insert()` member function is defined as a member template (see Section 3.2, page 34).

3. **Use `make_pair()`**. Probably the most convenient way before C++11 was to use `make_pair()`, which produces a pair object that contains the two values passed as arguments (see Section 5.1.1, page 65):

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

   Again, the necessary type conversions are performed by the `insert()` member template.

Here is a simple example of the insertion of an element into a map that also checks whether the insertion was successful:

[Click here to view code image]

```
std::map<std::string,float> coll;
...
if (coll.insert(std::make_pair("otto",22.3)).second) {
    std::cout << "OK, could insert otto/22.3" << std::endl;
}
else {
    std::cout << "OOPS, could not insert otto/22.3 "
              << "(key otto already exists)" << std::endl;
}
```

See Section 7.7.2, page 322, for a discussion about the return values of the `insert()` functions and more examples that also apply to maps. Note, again, that maps provide operator `[]` and `at()` as another convenient way to insert (and set) elements with the subscript operator (see Section 7.8.3, page 343).

When using `emplace()` and the key and/or the element value require more than one value for initialization, you have to pass two tuples of arguments: one for the key and one for the value. The most convenient way to do this is as follows:

[Click here to view code image]

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

```
m.emplace(std::piecewise_construct,    // pass tuple elements as arguments
          std::make_tuple("hello"),    // elements for the key
          std::make_tuple(3.4,7.8));   // elements for the value
```

See Section 5.1.1, page 63, for details of piecewise construction of pairs.

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

```
std::map<std::string,float> coll;
...
// remove all elements with the passed key
coll.erase(key);
```

This version of `erase()` returns the number of removed elements. When called for maps, the return value of `erase()` can only be `0` or `1` .

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

```
std::multimap<std::string,float> coll;
...
// remove first element with passed key
auto pos = coll.find(key);
if (pos != coll.end()) {
    coll.erase(pos);
}
```

You should use the member function `find()` instead of the `find()` algorithm here because it is faster (see an example with the `find()` algorithm in Section 7.3.2, page 277). However, you can't use the `find()` member functions to remove elements that have a certain value instead of a certain key. See Section 7.8.2, page 335, for a detailed discussion of this topic.

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

Click here to view code image

```
std::map<std::string,float> coll;
...
for (auto pos = coll.begin(); pos != coll.end(); ++pos) {
    if (pos->second == value) {
        coll.erase(pos);                    // RUNTIME ERROR !!!
    }
}
```

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

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

Click here to view code image

```
std::map<std::string,float> coll;
...
for (auto pos = coll.begin(); pos != coll.end(); ) {
    if (pos->second == value) {
        pos = coll.erase(pos);    // possible only since C++11
    }
    else {
        ++pos;
    }
}
```

Unfortunately, before C++11, it was a design decision not to return the following position, because if not needed, it costs unnecessary time. However, this made programming tasks like this error prone and complicated and even more costly in terms of time. Here is an example of the correct way to remove elements to which an iterator refers before C++11:

```
typedef std::map<std::string,float> StringFloatMap;
StringFloatMap coll;
StringFloatMap::iterator pos;
...
// remove all elements having a certain value
for (pos = coll.begin(); pos != coll.end(); ) {
    if (pos->second == value) {
        coll.erase(pos++);
    }
```

```
        else {
            ++pos;
        }
    }
```

Note that `pos++` increments `pos` so that it refers to the next element but yields a copy of its original value. Thus, `pos` doesn't refer to the element that is removed when `erase()` is called.

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

7.8.3. Using Maps as Associative Arrays

Associative containers don't typically provide abilities for direct element access. Instead, you must use iterators. For maps, as well as for unordered maps (see Section 7.9, page 355), however, there is an exception to this rule. Nonconstant maps provide a subscript operator for direct element access. In addition, since C++11, a corresponding member function `at()` is provided for constant and nonconstant maps (see Table 7.46).

Table 7.46. Direct Element Access of Maps

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

`at()` yields the value of the element with the passed key and throws an exception object of type `out_of_range` if no such element is present.

For operator `[ ]`, the index also is the key that is used to identify the element. This means that for operator `[ ]`, the index may have any type rather than only an integral type. Such an interface is the interface of a so-called *associative array*.

For operator `[ ]`, the type of the index is not the only difference from ordinary arrays. In addition, you can't have a wrong index. If you use a key as the index for which no element yet exists, a new element gets inserted into the map automatically. The value of the new element is initialized by the default constructor of its type. Thus, to use this feature, you can't use a value type that has no default constructor. Note that the fundamental data types provide a default constructor that initializes their values to zero (see Section 3.2.1, page 37).

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

• The advantage is that you can insert new elements into a map with a more convenient interface. For example:

Click here to view code image

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

// insert "otto"/7.7 as key/value pair
// - first it inserts "otto"/float()
// - then it assigns 7.7
coll["otto"] = 7.7;
```

The statement

```
coll["otto"] = 7.7;
```

is processed here as follows:

1. Process `coll["otto"]` expression:

– If an element with key `"otto"` exists, the expression returns the value of the element by reference.

– If, as in this example, no element with key `"otto"` exists, the expression inserts a new element automatically, with `"otto"` as key and the value of the default constructor of the value type as the element value. It then returns a reference to that new value of the new element.

2. Assign value `7.7`:

– The second part of the statement assigns `7.7` to the value of the new or existing element.

The map then contains an element with key `"otto"` and value `7.7`.

• The disadvantage is that you might insert new elements by accident or mistake. For example, the following statement does something you probably hadn't intended or expected:

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

It inserts a new element with key `"ottto"` and prints its value, which is `0` by default. However, it should have generated an error message telling you that you wrote `"otto"` incorrectly.

Note, too, that this way of inserting elements is slower than the usual way for maps, which is described in Section 7.8.2, page 340. The reason is that the new value is first initialized by the default value of its type, which is then overwritten by the correct value.

See Section 6.2.4, page 185, and Section 7.8.5, page 346, for some example code.

7.8.4. Exception Handling

Maps and multimaps provide the same behavior as sets and multisets with respect to exception safety. This behavior is mentioned in Section 7.7.3, page 325.

7.8.5. Examples of Using Maps and Multimaps

Using Algorithms and Lambdas with a Map/Multimap

Section 6.2.3, page 183, introduced an example for an unordered multimap, which could also be used with an ordinary (sorting) map or multimap. Here is a corresponding example using a map. This program also demonstrates how to use algorithms and lambdas instead of range-based `for` loops:

Click here to view code image

```
// cont/map1.cpp

#include <map>
#include <string>
#include <iostream>
#include <algorithm>
using namespace std;

int main()
{
    map<string,double> coll { { "tim", 9.9 },
                              { "struppi", 11.77 }
                            } ;

    // square the value of each element:
    for_each (coll.begin(), coll.end(),
              [] (pair<const string,double>& elem) {
                     elem.second *= elem.second;
              });

    // print each element:
    for_each (coll.begin(), coll.end(),
              [] (const map<string,double>::value_type& elem) {
                     cout << elem.first << ": " << elem.second << endl;
              });
}
```

As you can see, for a map, `for_each()` is called twice: once to square each element and once to print each element. In the first call, the type of an element is declared explicitly; in the second call, `value_type` is used. In the first call, the element is passed by reference to be able to modify its value; in the second call, a constant reference is used to avoid unnecessary copies.

The program has the following output:

```
struppi: 138.533
tim: 98.01
```

Using a Map as an Associative Array

The following example shows the use of a map as an associative array. The map is used as a stock chart. The elements of the map are pairs in which the key is the name of the stock and the value is its price:

Click here to view code image

```
// cont/map2.cpp

#include <map>
#include <string>
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
```

```
        // create map / associative array
        // - keys are strings
        // - values are floats
        typedef map<string,float> StringFloatMap;

        StringFloatMap stocks;      // create empty container

        // insert some elements
        stocks["BASF"] = 369.50;
        stocks["VW"] = 413.50;
        stocks["Daimler"] = 819.00;
        stocks["BMW"] = 834.00;
        stocks["Siemens"] = 842.20;

        // print all elements
        StringFloatMap::iterator pos;
        cout << left;   // left-adjust values
        for (pos = stocks.begin(); pos != stocks.end(); ++pos) {
            cout << "stock: " << setw(12) << pos->first
                 << "price: " << pos->second << endl;
        }
        cout << endl;

        // boom (all prices doubled)
        for (pos = stocks.begin(); pos != stocks.end(); ++pos) {
            pos->second *= 2;
        }

        // print all elements
        for (pos = stocks.begin(); pos != stocks.end(); ++pos) {
            cout << "stock: " << setw(12) << pos->first
                 << "price: " << pos->second << endl;
        }
        cout << endl;

        // rename key from "VW" to "Volkswagen"
        // - provided only by exchanging element
        stocks["Volkswagen"] = stocks["VW"];
        stocks.erase("VW");

        // print all elements
        for (pos = stocks.begin(); pos != stocks.end(); ++pos) {
            cout << "stock: " << setw(12) << pos->first
                 << "price: " << pos->second << endl;
        }
    }
```

The program has the following output:

```
stock: BASF        price: 369.5
stock: BMW         price: 834
stock: Daimler     price: 819
stock: Siemens     price: 842.2
stock: VW          price: 413.5

stock: BASF        price: 739
stock: BMW         price: 1668
stock: Daimler     price: 1638
stock: Siemens     price: 1684.4
stock: VW          price: 827

stock: BASF        price: 739
stock: BMW         price: 1668
stock: Daimler     price: 1638
stock: Siemens     price: 1684.4
stock: Volkswagen price: 827
```

Using a Multimap as a Dictionary

The following example shows how to use a multimap as a dictionary:

Click here to view code image

```
// cont/multimap1.cpp

#include <map>
#include <string>
#include <iostream>
#include <iomanip>
using namespace std;
```

```
int main()
{
    // create multimap as string/string dictionary
    multimap<string,string> dict;

    // insert some elements in random order
    dict.insert ( { {"day","Tag"}, {"strange","fremd"},
                    {"car","Auto"}, {"smart","elegant"},
                    {"trait","Merkmal"}, {"strange","seltsam"},
                    {"smart","raffiniert"}, {"smart","klug"},
                    {"clever","raffiniert"} } );

    // print all elements
    cout.setf (ios::left, ios::adjustfield);
    cout << ' ' << setw(10) << "english "
         << "german " << endl;
    cout << setfill('-') << setw(20) << ""
         << setfill(' ') << endl;
    for ( const auto& elem : dict ) {
        cout << ' ' << setw(10) << elem.first
             << elem.second << endl;
    }
    cout << endl;

    // print all values for key "smart"
    string word("smart");
    cout << word << ": " << endl;
    for (auto pos = dict.lower_bound(word);
         pos != dict.upper_bound(word);
         ++pos) {
            cout << "    " << pos->second << endl;
    }

    // print all keys for value "raffiniert"
    word = ("raffiniert");
    cout << word << ": " << endl;
    for (const auto& elem : dict) {
        if (elem.second == word) {
            cout << "    " << elem.first << endl;
        }
    }
}
```

The program has the following output:

```
  english   german
--------------------
  car       Auto
  clever    raffiniert
  day       Tag
  smart     elegant
  smart     raffiniert
  smart     klug
  strange   fremd
  strange   seltsam
  trait     Merkmal

smart:
     elegant
     raffiniert
     klug
raffiniert:
     clever
     smart
```

See a corresponding example that uses an unordered multimap as a dictionary in Section 7.9.7, page 383.

Finding Elements with Certain Values

The following example shows how to use the global `find_if()` algorithm to find an element with a certain value (in contrast to finding an element with a certain key):

Click here to view code image

```
// cont/mapfind1.cpp

#include <map>
#include <iostream>
#include <algorithm>
```

```
#include <utility>
using namespace std;

int main()
{
    // map with floats as key and value
    // - initializing keys and values are automatically converted to float
    map<float,float> coll = { {1,7}, {2,4}, {3,2}, {4,3},
                              {5,6}, {6,1}, {7,3} };

    // search an element with key 3.0  (logarithmic complexity)
    auto posKey = coll.find(3.0);
    if (posKey != coll.end()) {
        cout << "key 3.0 found ("
             << posKey->first << ":"
             << posKey->second << ")" << endl;
    }

    // search an element with value 3.0 (linear complexity)
    auto posVal = find_if(coll.begin(),coll.end(),
                          [] (const pair<float,float>& elem) {
                              return elem.second == 3.0;
                          });
    if (posVal != coll.end()) {
        cout << "value 3.0 found ("
             << posVal->first << ":"
             << posVal->second << ")" << endl;
    }
}
```

The output of the program is as follows:

```
key 3.0 found (3:2)
value 3.0 found (4:3)
```

7.8.6. Example with Maps, Strings, and Sorting Criterion at Runtime

The example here is for advanced programmers rather than STL beginners. You can take it as an example of both the power and the problems of the STL. In particular, this example demonstrates the following techniques:

- How to use maps, including the associative array interface
- How to write and use function objects
- How to define a sorting criterion at runtime
- How to compare strings in a case-insensitive way

Click here to view code image

```
// cont/mapcmp1.cpp

#include <iostream>
#include <iomanip>
#include <map>
#include <string>
#include <algorithm>
#include <cctype>
using namespace std;

// function object to compare strings
// - allows you to set the comparison criterion at runtime
// - allows you to compare case insensitive
class RuntimeStringCmp {
  public:
    // constants for the comparison criterion
    enum cmp_mode {normal, nocase};
  private:
    // actual comparison mode
    const cmp_mode mode;

    // auxiliary function to compare case insensitive
    static bool nocase_compare (char c1, char c2) {
        return toupper(c1) < toupper(c2);
    }
  public:
    // constructor: initializes the comparison criterion
    RuntimeStringCmp (cmp_mode m=normal) : mode(m) {
    }

    // the comparison
```

```
        bool operator() (const string& s1, const string& s2) const {
            if (mode == normal) {
               return s1<s2;
            }
            else {
                return lexicographical_compare (s1.begin(), s1.end(),
                                                s2.begin(), s2.end(),
                                                nocase_compare);
            }
        }
    };

    // container type:
    // - map with
    //    - string keys
    //    - string values
    //    - the special comparison object type
    typedef map<string,string,RuntimeStringCmp> StringStringMap;

    // function that fills and prints such containers
    void fillAndPrint(StringStringMap& coll);

    int main()
    {
        // create a container with the default comparison criterion
        StringStringMap coll1;
        fillAndPrint(coll1);

        // create an object for case-insensitive comparisons
        RuntimeStringCmp ignorecase(RuntimeStringCmp::nocase);

        // create a container with the case-insensitive comparisons criterion
        StringStringMap coll2(ignorecase);
        fillAndPrint(coll2);
    }

    void fillAndPrint(StringStringMap& coll)
    {
        // insert elements in random order
        coll["Deutschland"] = "Germany";
        coll["deutsch"] = "German";
        coll["Haken"] = "snag";
        coll["arbeiten"] = "work";
        coll["Hund"] = "dog";

        coll["gehen"] = "go";
        coll["Unternehmen"] = "enterprise";
        coll["unternehmen"] = "undertake";
        coll["gehen"] = "walk";
        coll["Bestatter"] = "undertaker";

        // print elements
        cout.setf(ios::left, ios::adjustfield);
        for (const auto& elem : coll) {
            cout << setw(15) << elem.first << " "
                 << elem.second << endl;
        }
        cout << endl;
    }
```

In the program, `main()` creates two containers and calls `fillAndPrint()` for them, which fills these containers with the same elements and prints their contents. However, the containers have two different sorting criteria:

1. `coll1` uses the default function object of type `RuntimeStringCmp`, which compares the elements by using operator `<`.
2. `coll2` uses a function object of type `RuntimeStringCmp`, which is initialized by value `nocase` of class `RuntimeStringCmp`. `nocase` forces this function object to sort strings in a case-insensitive way.

The program has the following output:

```
Bestatter       undertaker
Deutschland     Germany
Haken           snag
Hund            dog
Unternehmen     enterprise
arbeiten        work
deutsch         German
gehen           walk
```

```
unternehmen      undertake

arbeiten         work
Bestatter        undertaker
deutsch          German
Deutschland      Germany
gehen            walk
Haken            snag
Hund             dog
Unternehmen      undertake
```

The first block of the output prints the contents of the first container that compares with operator `<`. The output starts with all uppercase keys, followed by all lowercase keys.

The second block prints all case-insensitive items, so the order changed. But note that the second block has one item less because the uppercase word "`Unternehmen`" is, from a case-insensitive point of view, equal to the lowercase word "`unternehmen`,"[13] and we use a map that does not allow duplicates according to its comparison criterion. Unfortunately the result is a mess because the German key, initialized by is the translation for "enterprise," got the value "undertake." So a multimap should probably be used here. Doing so makes sense because a multimap is the typical container for dictionaries.

[13] In German, all nouns are written with an initial capital letter, whereas all verbs are written in lowercase letters.