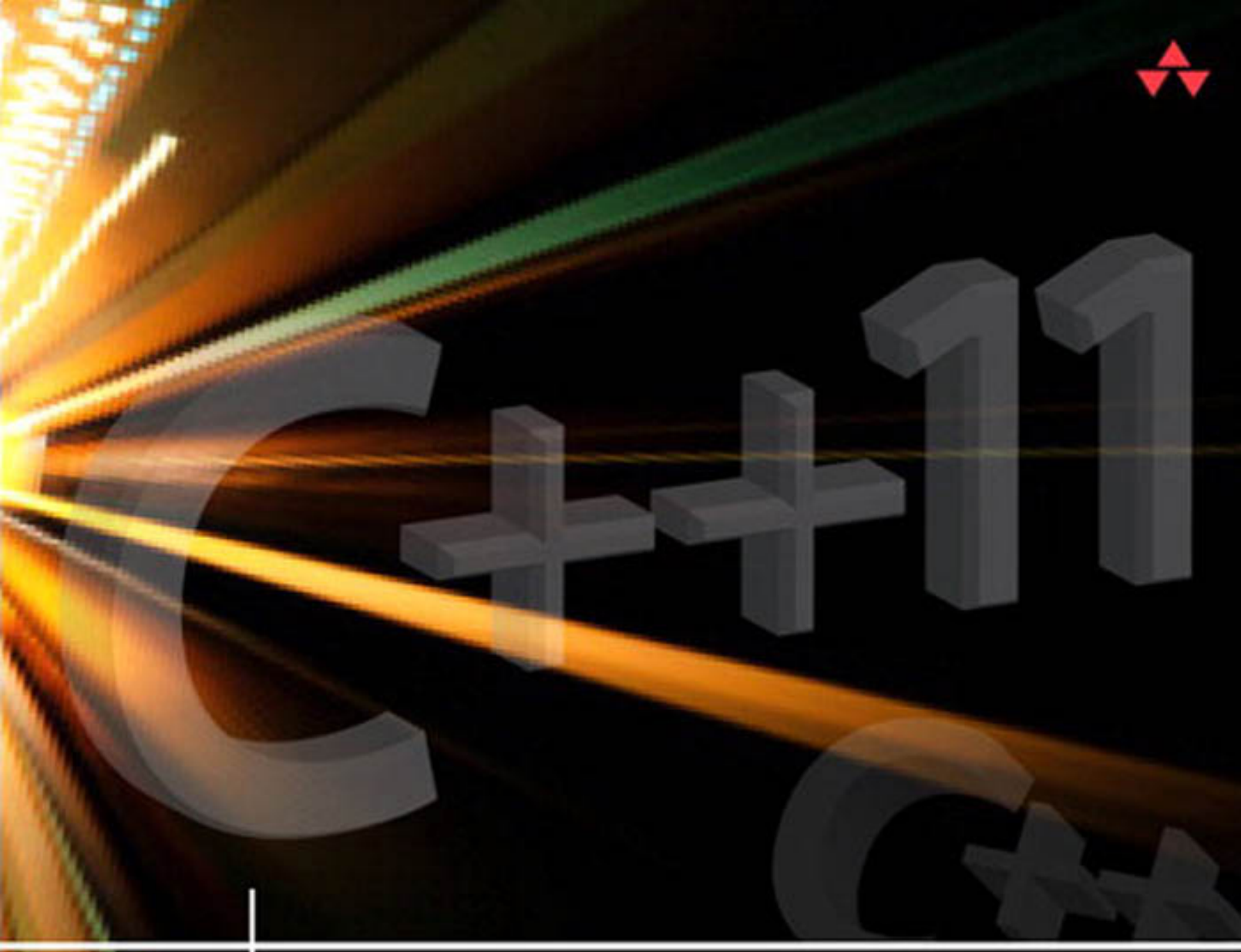

THE C++ STANDARD LIBRARY

SECOND EDITION

A Tutorial and Reference

NICOLAI M. JOSUTTIS

The C++ Standard Library

Second Edition

```
  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.

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).

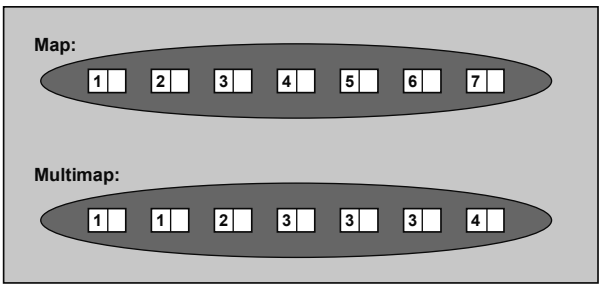

Figure 7.14. Maps and Multimaps

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`:

```
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.

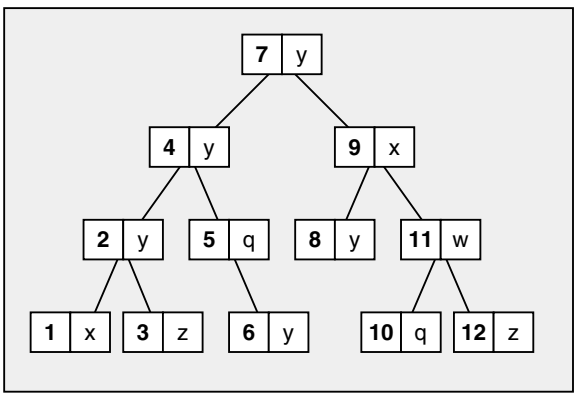

Figure 7.15. Internal Structure of Maps and Multimaps

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.

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:

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*

Table 7.40. Constructors and Destructors of Maps and Multimaps

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).

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:

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