Let the template library do the work.

Concordance with maps
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <string>
#include "GetNextWord.h"
#include <map>

char buffer[2048];

int main() {
    int len;
    char fname[1024];
    std::string word;
    std::map<std::string, unsigned> WordList;
    std::cout << "File name please matey: ";
    std::cin.get(fname, 1024);
    std::ifstream is(fname);
    if(!is) {
        std::cerr << "file " << fname << " doesn't seem to be there matey\n";
        exit(1);
    }
    do {
        word = getNextWord(is, len);
        if(len > 0) {
            //std::cout << "word is " << word << std::endl;
            auto WLP = WordList.find(word);
            if(WLP == WordList.end())
                WordList.insert({word, 1});
            else
                WLP->second++;
        }
    } while(len > 0);
    is.close();
    printf("About to list the words\n");
    for(auto WLP = WordList.cbegin(); WLP != WordList.cend(); WLP++)
        std::cout << WLP->first << " occurred " << WLP->second << " times\n";
    return 0;
}
About to list the words
"GetNextWord.h"occurred 1 times
"nullptr"occurred 1 times
#includeoccurred 2 times
%occurred 1 times
&is,occurred 1 times
&l}occurred 1 times
'\0';occurred 1 times
//initialiseoccurred 1 times
//printf("C_stringoccurred 1 times
//thisoccurred 1 times
0;occurred 1 times
<cstdio>occurred 1 times
=occurred 4 times
B[MaxCharCount];occurred 1 times
B[l++]occurred 2 times
B[l]occurred 1 times
Ioccurred 1 times
S("");occurred 1 times
S(B);occurred 1 times
S;occurred 2 times
\n",B);occurred 1 times
beoccurred 1 times
break;occurred 2 times
butoccurred 1 times
c;occurred 3 times
can'toccurred 1 times
charoccurred 2 times
findoccurred 1 times
getNextWord(std::ifstream);occurred 1 times
if(!isspace(c));occurred 1 times
if(is.eof());occurred 1 times
if(isspace(c));occurred 1 times
intoccurred 1 times
isoccurred 1 times
loccurred 1 times
necessaryoccurred 1 times
returnoccurred 3 times
shouldn'toccurred 1 times
std::stringoccurred 3 times
stringoccurred 1 times
theoccurred 1 times
while(is.get(c));occurred 2 times
{occurred 3 times
}occurred 4 times
```cpp
#include "GetNextWord.h"
#include <cstddef>
std::string getNextWord(std::istream &is, int &l)
{
    char B[MaxCharCount];
    char c;
    l = 0;
    while(is.get(c)) {
        if(!isspace(c))
            break;
    }
    if(is.eof()) {
        std::string S(""); //this shouldn't be necessary but I can't find
        "nullptr"
            return S;
    }
    B[l++] = c;
    while(is.get(c)) {
        if(isspace(c))
            break;
        B[l++] = c;
    }
    B[l] = '\0';
    //printf("C string is %s \n",B);
    std::string S(B); //initialise the return string
    return S;
}
```
The C++ Standard Library
A Tutorial and Reference
Second Edition

Nicolai M. Josuttis
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).

![Maps and Multimaps](image)

*Figure 7.14. Maps and Multimaps*

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

There, the types are defined as class templates inside namespace `std`:
```cpp
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 `T` 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 \texttt{less<} is used. The function object \texttt{less<} sorts the elements by comparing them with operator \texttt{<} (see Section 10.2.1, page 487, for details about \texttt{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 \texttt{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](image)

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 \textit{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>
<thead>
<tr>
<th>Operation</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>map c</code></td>
<td>Default constructor; creates an empty map/multimap without any elements</td>
</tr>
<tr>
<td><code>map c(op)</code></td>
<td>Creates an empty map/multimap that uses <code>op</code> as the sorting criterion</td>
</tr>
<tr>
<td><code>map c(c2)</code></td>
<td>Copy constructor; creates a copy of another map/multimap of the same type (all elements are copied)</td>
</tr>
<tr>
<td><code>map c = c2</code></td>
<td>Copy constructor; creates a copy of another map/multimap of the same type (all elements are copied)</td>
</tr>
<tr>
<td><code>map c(rv)</code></td>
<td>Move constructor; creates a new map/multimap of the same type, taking the contents of the rvalue <code>rv</code> (since C++11)</td>
</tr>
<tr>
<td><code>map c = rv</code></td>
<td>Move constructor; creates a new map/multimap of the same type, taking the contents of the rvalue <code>rv</code> (since C++11)</td>
</tr>
<tr>
<td><code>map c(beg, end)</code></td>
<td>Creates a map/multimap initialized by the elements of the range <code>[beg, end)</code></td>
</tr>
<tr>
<td><code>map c(beg, end, op)</code></td>
<td>Creates a map/multimap with the sorting criterion <code>op</code> initialized by the elements of the range <code>[beg, end)</code></td>
</tr>
<tr>
<td><code>map c(initlist)</code></td>
<td>Creates a map/multimap initialized with the elements of initializer list <code>initlist</code> (since C++11)</td>
</tr>
<tr>
<td><code>map c = initlist</code></td>
<td>Creates a map/multimap initialized with the elements of initializer list <code>initlist</code> (since C++11)</td>
</tr>
<tr>
<td><code>c -= map()</code></td>
<td>Destroys all elements and frees the memory</td>
</tr>
</tbody>
</table>

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

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

   ```cpp
   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:

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

   ```cpp
   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:

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

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

Table 7.41. Nonmodifying Operations of Maps and Multimaps

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>
<thead>
<tr>
<th>Operation</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>c.count(val)</td>
<td>Returns the number of elements with key val</td>
</tr>
<tr>
<td>c.find(val)</td>
<td>Returns the position of the first element with key val (or end() if none found)</td>
</tr>
<tr>
<td>c.lower_bound(val)</td>
<td>Returns the first position where an element with key val would get inserted (the first element with a key &gt;= val)</td>
</tr>
<tr>
<td>c.upper_bound(val)</td>
<td>Returns the last position where an element with key val would get inserted (the first element with a key &gt; val)</td>
</tr>
<tr>
<td>c.equal_range(val)</td>
<td>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)</td>
</tr>
</tbody>
</table>

Table 7.42. Special Search Operations of Maps and Multimaps

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:

```cpp
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();
    }
}
```

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 multimeaps provide only the fundamental assignment operations
that all containers provide (see Section 7.1.2, page 258).

<table>
<thead>
<tr>
<th>Operation</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>c = c2</code></td>
<td>Assigns all elements of <code>c2</code> to <code>c</code></td>
</tr>
<tr>
<td><code>c = rv</code></td>
<td>Move assigns all elements of the rvalue <code>rv</code> to <code>c</code> (since C++11)</td>
</tr>
<tr>
<td><code>c = initlist</code></td>
<td>Assigns all elements of the initializer list <code>initlist</code> to <code>c</code> (since C++11)</td>
</tr>
<tr>
<td><code>c1.swap(c2)</code></td>
<td>Swaps the data of <code>c1</code> and <code>c2</code></td>
</tr>
<tr>
<td><code>swap(c1, c2)</code></td>
<td>Swaps the data of <code>c1</code> and <code>c2</code></td>
</tr>
</tbody>
</table>

Table 7.43. Assignment Operations of Maps and Multimeaps

For these operations, both containers must have the same type. In particular, the type of the compar-
ison 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.
7.8 Maps and Multimaps

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>
<thead>
<tr>
<th>Operation</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>c.begin()</td>
<td>Returns a bidirectional iterator for the first element</td>
</tr>
<tr>
<td>c.end()</td>
<td>Returns a bidirectional iterator for the position after the last element</td>
</tr>
<tr>
<td>c.cbegin()</td>
<td>Returns a constant bidirectional iterator for the first element (since C++11)</td>
</tr>
<tr>
<td>c.cend()</td>
<td>Returns a constant bidirectional iterator for the position after the last element (since C++11)</td>
</tr>
<tr>
<td>c.rbegin()</td>
<td>Returns a reverse iterator for the first element of a reverse iteration</td>
</tr>
<tr>
<td>c.rend()</td>
<td>Returns a reverse iterator for the position after the last element of a reverse iteration</td>
</tr>
<tr>
<td>c.crbegin()</td>
<td>Returns a constant reverse iterator for the first element of a reverse iteration (since C++11)</td>
</tr>
<tr>
<td>c.crend()</td>
<td>Returns a constant reverse iterator for the position after the last element of a reverse iteration (since C++11)</td>
</tr>
</tbody>
</table>

Table 7.44. Iterator Operations of Maps and Multimaps

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:

```cpp
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:

```cpp
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.\(^{12}\)

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

```cpp
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:

```cpp
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:

```cpp
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:

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

you could use

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

or

```cpp
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:

\(^{12}\) pos->first is a shortcut for (*pos).first.
// 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:

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

It works the same way for multimaps.

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:

```cpp
// 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>
<thead>
<tr>
<th>Operation</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>c.insert(val)</td>
<td>Inserts a copy of val and returns the position of the new element and, for maps, whether it succeeded</td>
</tr>
<tr>
<td>c.insert(pos,val)</td>
<td>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)</td>
</tr>
<tr>
<td>c.insert(beg,end)</td>
<td>Inserts a copy of all elements of the range [beg,end] (returns nothing)</td>
</tr>
<tr>
<td>c.insert(initialist)</td>
<td>Inserts a copy of all elements in the initializer list initialist (returns nothing; since C++11)</td>
</tr>
<tr>
<td>c.emplace(args...)</td>
<td>Inserts a copy of an element initialized with args and returns the position of the new element and, for maps, whether it succeeded (since C++11)</td>
</tr>
<tr>
<td>c.emplace_hint(pos,args...)</td>
<td>Inserts a copy of an 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)</td>
</tr>
<tr>
<td>c.erase(val)</td>
<td>Removes all elements equal to val and returns the number of removed elements</td>
</tr>
<tr>
<td>c.erase(pos)</td>
<td>Removes the element at iterator position pos and returns the following position (returned nothing before C++11)</td>
</tr>
<tr>
<td>c.erase(beg,end)</td>
<td>Removes all elements of the range [beg,end] and returns the following position (returned nothing before C++11)</td>
</tr>
<tr>
<td>c.clear()</td>
<td>Removes all elements (empties the container)</td>
</tr>
</tbody>
</table>

Table 7.45. Insert and Remove Operations of Maps and Multimaps

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.

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 pass them as an initializer list, where the first entry is the key and the second entry is the value:

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

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:

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

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

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

   ```cpp
   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):

   ```cpp
   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:

```cpp
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 << "DUPS, 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()` to insert a new element by passing the values for its construction, you have to pass two lists of arguments: one for the key and one for the element. The most convenient way to do this is as follows:

```cpp
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()`:

```cpp
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:

```cpp
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:

```cpp
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:

```cpp
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:

```cpp
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