Chapter 11

Associative Containers

Contents

Section 11.1 Using an Associative Container 420
Section 11.2 Overview of the Associative Containers . 423
Section 11.3 Operations on Associative Containers . . . 428
Section 11.4 The Unordered Containers 443
Chapter Summary 447
Defined Terms 447

Associative and sequential containers differ from one another in a fundamental way: Elements in an associative container are stored and retrieved by a key. In contrast, elements in a sequential container are stored and accessed sequentially by their position in the container.

Although the associative containers share much of the behavior of the sequential containers, they differ from the sequential containers in ways that reflect the use of keys.

Associative containers support efficient lookup and retrieval by a key. The two primary **associative-container** types are **`map`** and **`set`**. The elements in a `map` are key–value pairs: The key serves as an index into the `map`, and the value represents the data associated with that index. A `set` element contains only a key; a `set` supports efficient queries as to whether a given key is present. We might use a `set` to hold words that we want to ignore during some kind of text processing. A dictionary would be a good use for a `map`: The word would be the key, and its definition would be the value.

The library provides eight associative containers, listed in Table 11.1. These eight differ along three dimensions: Each container is (1) a `set` or a `map`, (2) requires unique keys or allows multiple keys, and (3) stores the elements in order or not. The containers that allow multiple keys include the word `multi`; those that do not keep their keys ordered start with the word `unordered`. Hence an `unordered_multi_set` is a set that allows multiple keys whose elements are not stored in order, whereas a `set` has unique keys that are stored in order. The unordered containers use a hash function to organize their elements. We'll have more to say about the hash function in § 11.4 (p. 444).

The `map` and **`multimap`** types are defined in the `map` header; the `set` and **`multiset`** types are in the `set` header; and the unordered containers are in the `unordered_map` and `unordered_set` headers.

Table 11.1: Associative Container Types

Elements Ordered by Key	
`map`	Associative array; holds key–value pairs
`set`	Container in which the key is the value
`multimap`	`map` in which a key can appear multiple times
`multiset`	`set` in which a key can appear multiple times
Unordered Collections	
`unordered_map`	`map` organized by a hash function
`unordered_set`	`set` organized by a hash function
`unordered_multimap`	Hashed `map`; keys can appear multiple times
`unordered_multiset`	Hashed `set`; keys can appear multiple times

11.1 Using an Associative Container

Although most programmers are familiar with data structures such as `vector`s and `list`s, many have never used an associative data structure. Before we look at the details of how the library supports these types, it will be helpful to start with examples of how we can use these containers.

A `map` is a collection of key–value pairs. For example, each pair might contain a person's name as a key and a phone number as its value. We speak of such a data structure as "mapping names to phone numbers." The `map` type is often referred to as an **associative array**. An associative array is like a "normal" array except that its subscripts don't have to be integers. Values in a `map` are found by a key rather than by their position. Given a map of names to phone numbers, we'd use a person's name as a subscript to fetch that person's phone number.

In contrast, a set is simply a collection of keys. A set is most useful when we simply want to know whether a value is present. For example, a business might define a set named bad_checks to hold the names of individuals who have written bad checks. Before accepting a check, that business would query bad_checks to see whether the customer's name was present.

Using a map

A classic example that relies on associative arrays is a word-counting program:

```
// count the number of times each word occurs in the input
map<string, size_t> word_count; // empty map from string to size_t
string word;
while (cin >> word)
        ++word_count[word];     // fetch and increment the counter for word
for (const auto &w : word_count) // for each element in the map
    // print the results
    cout <<  w.first << " occurs " << w.second
         << ((w.second > 1) ? " times" : " time") << endl;
```

This program reads its input and reports how often each word appears.

Like the sequential containers, the associative containers are templates (§ 3.3, p. 96). To define a map, we must specify both the key and value types. In this program, the map stores elements in which the keys are strings and the values are size_ts (§ 3.5.2, p. 116). When we subscript word_count, we use a string as the subscript, and we get back the size_t counter associated with that string.

The while loop reads the standard input one word at a time. It uses each word to subscript word_count. If word is not already in the map, the subscript operator creates a new element whose key is word and whose value is 0. Regardless of whether the element had to be created, we increment the value.

Once we've read all the input, the range for (§ 3.2.3, p. 91) iterates through the map, printing each word and the corresponding counter. When we fetch an element from a map, we get an object of type pair, which we'll describe in § 11.2.3 (p. 426). Briefly, a pair is a template type that holds two (public) data elements named first and second. The pairs used by map have a first member that is the key and a second member that is the corresponding value. Thus, the effect of the output statement is to print each word and its associated counter.

If we ran this program on the text of the first paragraph in this section, our output would be

```
Although occurs 1 time
Before occurs 1 time
an occurs 1 time
and occurs 1 time
...
```

Using a set

A logical extension to our program is to ignore common words like "the," "and," "or," and so on. We'll use a `set` to hold the words we want to ignore and count only those words that are not in this set:

```
// count the number of times each word occurs in the input
map<string, size_t> word_count; // empty map from string to size_t
set<string> exclude = {"The", "But", "And", "Or", "An", "A",
                       "the", "but", "and", "or", "an", "a"};
string word;
while (cin >> word)
    // count only words that are not in exclude
    if (exclude.find(word) == exclude.end())
        ++word_count[word];   // fetch and increment the counter for word
```

Like the other containers, `set` is a template. To define a `set`, we specify the type of its elements, which in this case are `string`s. As with the sequential containers, we can list initialize (§ 9.2.4, p. 336) the elements of an associative container. Our `exclude` set holds the 12 words we want to ignore.

The important difference between this program and the previous program is that before counting each word, we check whether the word is in the exclusion set. We do this check in the `if`:

```
// count only words that are not in exclude
if (exclude.find(word) == exclude.end())
```

The call to `find` returns an iterator. If the given key is in the `set`, the iterator refers to that key. If the element is not found, `find` returns the off-the-end iterator. In this version, we update the counter for `word` only if `word` is not in `exclude`.

If we run this version on the same input as before, our output would be

```
Although occurs 1 time
Before occurs 1 time
are occurs 1 time
as occurs 1 time
...
```

Exercises Section 11.1

Exercise 11.1: Describe the differences between a `map` and a `vector`.

Exercise 11.2: Give an example of when each of `list`, `vector`, `deque`, `map`, and `set` might be most useful.

Exercise 11.3: Write your own version of the word-counting program.

Exercise 11.4: Extend your program to ignore case and punctuation. For example, "example." "example," and "Example" should all increment the same counter.

11.2 Overview of the Associative Containers

Associative containers (both ordered and unordered) support the general container operations covered in § 9.2 (p. 328) and listed in Table 9.2 (p. 330). The associative containers do *not* support the sequential-container position-specific operations, such as `push_front` or `back`. Because the elements are stored based on their keys, these operations would be meaningless for the associative containers. Moreover, the associative containers do not support the constructors or insert operations that take an element value and a count.

In addition to the operations they share with the sequential containers, the associative containers provide some operations (Table 11.7 (p. 438)) and type aliases (Table 11.3 (p. 429)) that the sequential containers do not. In addition, the unordered containers provide operations for tuning their hash performance, which we'll cover in § 11.4 (p. 444).

The associative container iterators are bidirectional (§ 10.5.1, p. 410).

11.2.1 Defining an Associative Container

As we've just seen, when we define a `map`, we must indicate both the key and value type; when we define a `set`, we specify only a key type, because there is no value type. Each of the associative containers defines a default constructor, which creates an empty container of the specified type. We can also initialize an associative container as a copy of another container of the same type or from a range of values, so long as those values can be converted to the type of the container. Under the new standard, we can also list initialize the elements:

```
map<string, size_t> word_count;  // empty
// list initialization
set<string> exclude = {"the", "but", "and", "or", "an", "a",
                       "The", "But", "And", "Or", "An", "A"};
// three elements; authors maps last name to first
map<string, string> authors = { {"Joyce", "James"},
                                {"Austen", "Jane"},
                                {"Dickens", "Charles"} };
```

As usual, the initializers must be convertible to the type in the container. For `set`, the element type is the key type.

When we initialize a `map`, we have to supply both the key and the value. We wrap each key–value pair inside curly braces:

{*key*, *value*}

to indicate that the items together form one element in the `map`. The key is the first element in each pair, and the value is the second. Thus, `authors` maps last names to first names, and is initialized with three elements.

Initializing a `multimap` or `multiset`

The keys in a `map` or a `set` must be unique; there can be only one element with a given key. The `multimap` and `multiset` containers have no such restriction;

there can be several elements with the same key. For example, the `map` we used to count words must have only one element per given word. On the other hand, a dictionary could have several definitions associated with a particular word.

The following example illustrates the differences between the containers with unique keys and those that have multiple keys. First, we'll create a `vector` of `int`s named `ivec` that has 20 elements: two copies of each of the integers from 0 through 9 inclusive. We'll use that `vector` to initialize a `set` and a `multiset`:

```
// define a vector with 20 elements, holding two copies of each number from 0 to 9
vector<int> ivec;
for (vector<int>::size_type i = 0; i != 10; ++i) {
    ivec.push_back(i);
    ivec.push_back(i);  // duplicate copies of each number
}
// iset holds unique elements from ivec; miset holds all 20 elements
set<int> iset(ivec.cbegin(), ivec.cend());
multiset<int> miset(ivec.cbegin(), ivec.cend());
cout << ivec.size() << endl;    // prints 20
cout << iset.size() << endl;    // prints 10
cout << miset.size() << endl;   // prints 20
```

Even though we initialized `iset` from the entire `ivec` container, `iset` has only ten elements: one for each distinct element in `ivec`. On the other hand, `miset` has 20 elements, the same as the number of elements in `ivec`.

EXERCISES SECTION 11.2.1

Exercise 11.5: Explain the difference between a `map` and a `set`. When might you use one or the other?

Exercise 11.6: Explain the difference between a `set` and a `list`. When might you use one or the other?

Exercise 11.7: Define a `map` for which the key is the family's last name and the value is a `vector` of the children's names. Write code to add new families and to add new children to an existing family.

Exercise 11.8: Write a program that stores the excluded words in a `vector` instead of in a `set`. What are the advantages to using a `set`?

11.2.2 Requirements on Key Type

The associative containers place constraints on the type that is used as a key. We'll cover the requirements for keys in the unordered containers in § 11.4 (p. 445). For the ordered containers—`map`, `multimap`, `set`, and `multiset`—the key type must define a way to compare the elements. By default, the library uses the < operator for the key type to compare the keys. In the set types, the key is the element type;

in the map types, the key is the first type. Thus, the key type for `word_count` in § 11.1 (p. 421) is `string`. Similarly, the key type for `exclude` is `string`.

> **Note:** Callable objects passed to a sort algorithm (§ 10.3.1, p. 386) must meet the same requirements as do the keys in an associative container.

Key Types for Ordered Containers

Just as we can provide our own comparison operation to an algorithm (§ 10.3, p. 385), we can also supply our own operation to use in place of the < operator on keys. The specified operation must define a **strict weak ordering** over the key type. We can think of a strict weak ordering as "less than," although our function might use a more complicated procedure. However we define it, the comparison function must have the following properties:

- Two keys cannot both be "less than" each other; if `k1` is "less than" `k2`, then `k2` must never be "less than" `k1`.
- If `k1` is "less than" `k2` and `k2` is "less than" `k3`, then `k1` must be "less than" `k3`.
- If there are two keys, and neither key is "less than" the other, then we'll say that those keys are "equivalent." If `k1` is "equivalent" to `k2` and `k2` is "equivalent" to `k3`, then `k1` must be "equivalent" to `k3`.

If two keys are equivalent (i.e., if neither is "less than" the other), the container treats them as equal. When used as a key to a `map`, there will be only one element associated with those keys, and either key can be used to access the corresponding value.

> **Note:** In practice, what's important is that a type that defines a < operator that "behaves normally" can be used as a key.

Using a Comparison Function for the Key Type

The type of the operation that a container uses to organize its elements is part of the type of that container. To specify our own operation, we must supply the type of that operation when we define the type of an associative container. The operation type is specified following the element type inside the angle brackets that we use to say which type of container we are defining.

Each type inside the angle brackets is just that, a type. We supply a particular comparison operation (that must have the same type as we specified inside the angle brackets) as a constructor argument when we create a container.

For example, we can't directly define a `multiset` of `Sales_data` because `Sales_data` doesn't have a < operator. However, we can use the `compareIsbn` function from the exercises in § 10.3.1 (p. 387) to define a `multiset`. That function defines a strict weak ordering based on their ISBNs of two given `Sales_data` objects. The `compareIsbn` function should look something like

```
bool compareIsbn(const Sales_data &lhs, const Sales_data &rhs)
{
    return lhs.isbn() < rhs.isbn();
}
```

To use our own operation, we must define the `multiset` with two types: the key type, `Sales_data`, and the comparison type, which is a function pointer type (§ 6.7, p. 247) that can point to `compareIsbn`. When we define objects of this type, we supply a pointer to the operation we intend to use. In this case, we supply a pointer to `compareIsbn`:

```
// bookstore can have several transactions with the same ISBN
// elements in bookstore will be in ISBN order
multiset<Sales_data, decltype(compareIsbn)*>
    bookstore(compareIsbn);
```

Here, we use `decltype` to specify the type of our operation, remembering that when we use `decltype` to form a function pointer, we must add a `*` to indicate that we're using a pointer to the given function type (§ 6.7, p. 250). We initialize `bookstore` from `compareIsbn`, which means that when we add elements to `bookstore`, those elements will be ordered by calling `compareIsbn`. That is, the elements in `bookstore` will be ordered by their `ISBN` members. We can write `compareIsbn` instead of `&compareIsbn` as the constructor argument because when we use the name of a function, it is automatically converted into a pointer if needed (§ 6.7, p. 248). We could have written `&compareIsbn` with the same effect.

EXERCISES SECTION 11.2.2

Exercise 11.9: Define a `map` that associates words with a `list` of line numbers on which the word might occur.

Exercise 11.10: Could we define a `map` from `vector<int>::iterator` to `int`? What about from `list<int>::iterator` to `int`? In each case, if not, why not?

Exercise 11.11: Redefine `bookstore` without using `decltype`.

11.2.3 The `pair` Type

Before we look at the operations on associative containers, we need to know about the library type named **`pair`**, which is defined in the `utility` header.

A `pair` holds two data members. Like the containers, `pair` is a template from which we generate specific types. We must supply two type names when we create a `pair`. The data members of the `pair` have the corresponding types. There is no requirement that the two types be the same:

```
pair<string, string> anon;       // holds two strings
pair<string, size_t> word_count; // holds a string and an size_t
pair<string, vector<int>> line;  // holds string and vector<int>
```

The default pair constructor value initializes (§ 3.3.1, p. 98) the data members. Thus, anon is a pair of two empty strings, and line holds an empty string and an empty vector. The size_t value in word_count gets the value 0, and the string member is initialized to the empty string.

We can also provide initializers for each member:

```
pair<string, string> author{"James", "Joyce"};
```

creates a pair named author, initialized with the values "James" and "Joyce".

Table 11.2: Operations on pairs

pair<T1, T2> p;	p is a pair with value initialized (§ 3.3.1, p. 98) members of types T1 and T2, respectively.
pair<T1, T2> p(v1, v2);	p is a pair with types T1 and T2; the first and second members are initialized from v1 and v2, respectively.
pair<T1, T2> p = {v1, v2};	Equivalent to p(v1, v2).
make_pair(v1, v2)	Returns a pair initialized from v1 and v2. The type of the pair is inferred from the types of v1 and v2.
p.first	Returns the (public) data member of p named first.
p.second	Returns the (public) data member of p named second.
p1 *relop* p2	Relational operators (<, >, <=, >=). Relational operators are defined as dictionary ordering: For example, p1 < p2 is true if p1.first < p2.first or if !(p2.first < p1.first) && p1.second < p2.second. Uses the element's < operator.
p1 == p2 p1 != p2	Two pairs are equal if their first and second members are respectively equal. Uses the element's == operator.

Unlike other library types, the data members of pair are public (§ 7.2, p. 268). These members are named first and second, respectively. We access these members using the normal member access notation (§ 1.5.2, p. 23), as, for example, we did in the output statement of our word-counting program on page 421:

```
// print the results
cout <<  w.first << " occurs " << w.second
     << ((w.second > 1) ? " times" : " time") << endl;
```

Here, w is a reference to an element in a map. Elements in a map are pairs. In this statement we print the first member of the element, which is the key, followed by the second member, which is the counter. The library defines only a limited number of operations on pairs, which are listed in Table 11.2.

A Function to Create pair Objects

Imagine we have a function that needs to return a pair. Under the new standard we can list initialize the return value (§ 6.3.2, p. 226): C++ 11

```
pair<string, int>
process(vector<string> &v)
{
    // process v
    if (!v.empty())
        return {v.back(), v.back().size()}; // list initialize
    else
        return pair<string, int>(); // explicitly constructed return value
}
```

If `v` isn't empty, we return a `pair` composed of the last `string` in `v` and the size of that `string`. Otherwise, we explicitly construct and return an empty `pair`.

Under earlier versions of C++, we couldn't use braced initializers to return a type like `pair`. Instead, we might have written both returns to explicitly construct the return value:

```
if (!v.empty())
    return pair<string, int>(v.back(), v.back().size());
```

Alternatively, we could have used `make_pair` to generate a new `pair` of the appropriate type from its two arguments:

```
if (!v.empty())
    return make_pair(v.back(), v.back().size());
```

EXERCISES SECTION 11.2.3

Exercise 11.12: Write a program to read a sequence of `string`s and `int`s, storing each into a `pair`. Store the `pair`s in a `vector`.

Exercise 11.13: There are at least three ways to create the `pair`s in the program for the previous exercise. Write three versions of that program, creating the `pair`s in each way. Explain which form you think is easiest to write and understand, and why.

Exercise 11.14: Extend the `map` of children to their family name that you wrote for the exercises in § 11.2.1 (p. 424) by having the `vector` store a `pair` that holds a child's name and birthday.

11.3 Operations on Associative Containers

In addition to the types listed in Table 9.2 (p. 330), the associative containers define the types listed in Table 11.3. These types represent the container's key and value types.

For the `set` types, the **`key_type`** and the **`value_type`** are the same; the values held in a `set` are the keys. In a `map`, the elements are key–value pairs. That is, each element is a `pair` object containing a key and a associated value. Because we cannot change an element's key, the key part of these `pair`s is `const`:

Table 11.3: Associative Container Additional Type Aliases

`key_type`	Type of the key for this container type
`mapped_type`	Type associated with each key; **`map` types only**
`value_type`	For `set`s, same as the `key_type` For `map`s, `pair<const key_type, mapped_type>`

```
set<string>::value_type v1;       // v1 is a string
set<string>::key_type v2;         // v2 is a string
map<string, int>::value_type v3;  // v3 is a pair<const string, int>
map<string, int>::key_type v4;    // v4 is a string
map<string, int>::mapped_type v5; // v5 is an int
```

As with the sequential containers (§ 9.2.2, p. 332), we use the scope operator to fetch a type member—for example, `map<string, int>::key_type`.

Only the `map` types (`unordered_map`, `unordered_multimap`, `multimap`, and `map`) define **`mapped_type`**.

11.3.1 Associative Container Iterators

When we dereference an iterator, we get a reference to a value of the container's `value_type`. In the case of `map`, the `value_type` is a `pair` in which `first` holds the `const` key and `second` holds the value:

```
// get an iterator to an element in word_count
auto map_it = word_count.begin();
// *map_it is a reference to a pair<const string, size_t> object
cout << map_it->first;          // prints the key for this element
cout << " " << map_it->second;  // prints the value of the element
map_it->first = "new key";      // error: key is const
++map_it->second;      // ok: we can change the value through an iterator
```

Note

It is essential to remember that the `value_type` of a `map` is a `pair` and that we can change the value but not the key member of that `pair`.

Iterators for `set`s Are `const`

Although the `set` types define both the `iterator` and `const_iterator` types, both types of iterators give us read-only access to the elements in the `set`. Just as we cannot change the key part of a `map` element, the keys in a `set` are also `const`. We can use a `set` iterator to read, but not write, an element's value:

```
set<int> iset = {0,1,2,3,4,5,6,7,8,9};
set<int>::iterator set_it = iset.begin();
if (set_it != iset.end()) {
    *set_it = 42;            // error: keys in a set are read-only
    cout << *set_it << endl; // ok: can read the key
}
```

Iterating across an Associative Container

The map and set types provide all the begin and end operations from Table 9.2 (p. 330). As usual, we can use these functions to obtain iterators that we can use to traverse the container. For example, we can rewrite the loop that printed the results in our word-counting program on page 421 as follows:

```
// get an iterator positioned on the first element
auto map_it = word_count.cbegin();
// compare the current iterator to the off-the-end iterator
while (map_it != word_count.cend()) {
    // dereference the iterator to print the element key--value pairs
    cout << map_it->first << " occurs "
         << map_it->second << " times" << endl;
    ++map_it;  // increment the iterator to denote the next element
}
```

The while condition and increment for the iterator in this loop look a lot like the programs we wrote that printed the contents of a vector or a string. We initialize an iterator, map_it, to refer to the first element in word_count. As long as the iterator is not equal to the end value, we print the current element and then increment the iterator. The output statement dereferences map_it to get the members of pair but is otherwise the same as the one in our original program.

Note

The output of this program is in alphabetical order. When we use an iterator to traverse a map, multimap, set, or multiset, the iterators yield elements in ascending key order.

Associative Containers and Algorithms

In general, we do not use the generic algorithms (Chapter 10) with the associative containers. The fact that the keys are const means that we cannot pass associative container iterators to algorithms that write to or reorder container elements. Such algorithms need to write to the elements. The elements in the set types are const, and those in maps are pairs whose first element is const.

Associative containers can be used with the algorithms that read elements. However, many of these algorithms search the sequence. Because elements in an associative container can be found (quickly) by their key, it is almost always a bad idea to use a generic search algorithm. For example, as we'll see in § 11.3.5 (p. 436), the associative containers define a member named find, which directly fetches the element with a given key. We could use the generic find algorithm to look for an element, but that algorithm does a sequential search. It is much faster to use the find member defined by the container than to call the generic version.

In practice, if we do so at all, we use an associative container with the algorithms either as the source sequence or as a destination. For example, we might use the generic copy algorithm to copy the elements from an associative container into another sequence. Similarly, we can call inserter to bind an insert iterator (§ 10.4.1, p. 401) to an associative container. Using inserter, we can use the associative container as a destination for another algorithm.

Exercises Section 11.3.1

Exercise 11.15: What are the mapped_type, key_type, and value_type of a map from int to vector<int>?

Exercise 11.16: Using a map iterator write an expression that assigns a value to an element.

Exercise 11.17: Assuming c is a multiset of strings and v is a vector of strings, explain the following calls. Indicate whether each call is legal:

```
copy(v.begin(), v.end(), inserter(c, c.end()));
copy(v.begin(), v.end(), back_inserter(c));
copy(c.begin(), c.end(), inserter(v, v.end()));
copy(c.begin(), c.end(), back_inserter(v));
```

Exercise 11.18: Write the type of map_it from the loop on page 430 without using auto or decltype.

Exercise 11.19: Define a variable that you initialize by calling begin() on the multiset named bookstore from § 11.2.2 (p. 425). Write the variable's type without using auto or decltype.

11.3.2 Adding Elements

The insert members (Table 11.4 (overleaf)) add one element or a range of elements. Because map and set (and the corresponding unordered types) contain unique keys, inserting an element that is already present has no effect:

```
vector<int> ivec = {2,4,6,8,2,4,6,8};     // ivec has eight elements
set<int> set2;                            // empty set
set2.insert(ivec.cbegin(), ivec.cend()); // set2 has four elements
set2.insert({1,3,5,7,1,3,5,7});        // set2 now has eight elements
```

The versions of insert that take a pair of iterators or an initializer list work similarly to the corresponding constructors (§ 11.2.1, p. 423)—only the first element with a given key is inserted.

Adding Elements to a map

When we insert into a map, we must remember that the element type is a pair. Often, we don't have a pair object that we want to insert. Instead, we create a pair in the argument list to insert:

```
// four ways to add word to word_count
word_count.insert({word, 1});
word_count.insert(make_pair(word, 1));
word_count.insert(pair<string, size_t>(word, 1));
word_count.insert(map<string, size_t>::value_type(word, 1));
```

As we've seen, under the new standard the easiest way to create a pair is to use brace initialization inside the argument list. Alternatively, we can call make_pair C++ 11 or explicitly construct the `pair`. The argument in the last call to `insert`:

```
map<string, size_t>::value_type(s, 1)
```

constructs a new object of the appropriate `pair` type to insert into the `map`.

Table 11.4: Associative Container `insert` Operations

`c.insert(v)` `c.emplace(`*args*`)`	`v` `value_type` object; *args* are used to construct an element. For `map` and `set`, the element is inserted (or constructed) only if an element with the given key is not already in `c`. Returns a `pair` containing an iterator referring to the element with the given key and a `bool` indicating whether the element was inserted. For `multimap` and `multiset`, inserts (or constructs) the given element and returns an iterator to the new element.
`c.insert(b, e)` `c.insert(il)`	`b` and `e` are iterators that denote a range of `c::value_type` values; `il` is a braced list of such values. Returns `void`. For `map` and `set`, inserts the elements with keys that are not already in `c`. For `multimap` and `multiset` inserts, each element in the range.
`c.insert(p, v)` `c.emplace(p, `*args*`)`	Like `insert(v)` (or `emplace(`*args*`)`), but uses iterator `p` as a hint for where to begin the search for where the new element should be stored. Returns an iterator to the element with the given key.

Testing the Return from `insert`

The value returned by `insert` (or `emplace`) depends on the container type and the parameters. For the containers that have unique keys, the versions of `insert` and `emplace` that add a single element return a `pair` that lets us know whether the insertion happened. The `first` member of the `pair` is an iterator to the element with the given key; the `second` is a `bool` indicating whether that element was inserted, or was already there. If the key is already in the container, then `insert` does nothing, and the `bool` portion of the return value is `false`. If the key isn't present, then the element is inserted and the `bool` is `true`.

As an example, we'll rewrite our word-counting program to use `insert`:

```
// more verbose way to count number of times each word occurs in the input
map<string, size_t> word_count; // empty map from string to size_t
string word;
while (cin >> word) {
    // inserts an element with key equal to word and value 1;
    // if word is already in word_count, insert does nothing
    auto ret = word_count.insert({word, 1});
    if (!ret.second)          // word was already in word_count
        ++ret.first->second; // increment the counter
}
```

For each `word`, we attempt to `insert` it with a value `1`. If `word` is already in the `map`, then nothing happens. In particular, the counter associated with `word` is unchanged. If word is not already in the map, then that string is added to the map and its counter value is set to 1.

The if test examines the bool part of the return value. If that value is false, then the insertion didn't happen. In this case, word was already in word_count, so we must increment the value associated with that element.

Unwinding the Syntax

The statement that increments the counter in this version of the word-counting program can be hard to understand. It will be easier to understand that expression by first parenthesizing it to reflect the precedence (§ 4.1.2, p. 136) of the operators:

```
++((ret.first)->second); // equivalent expression
```

Explaining this expression step by step:

ret holds the value returned by insert, which is a pair.

ret.first is the first member of that pair, which is a map iterator referring to the element with the given key.

ret.first-> dereferences that iterator to fetch that element. Elements in the map are also pairs.

ret.first->second is the value part of the map element pair.

++ret.first->second increments that value.

Putting it back together, the increment statement fetches the iterator for the element with the key word and increments the counter associated with the key we tried to insert.

For readers using an older compiler or reading code that predates the new standard, declaring and initializing ret is also somewhat tricky:

```
pair<map<string, size_t>::iterator, bool> ret =
            word_count.insert(make_pair(word, 1));
```

It should be easy to see that we're defining a pair and that the second type of the pair is bool. The first type of that pair is a bit harder to understand. It is the iterator type defined by the map<string, size_t> type.

Adding Elements to multiset or multimap

Our word-counting program depends on the fact that a given key can occur only once. That way, there is only one counter associated with any given word. Sometimes, we want to be able to add additional elements with the same key. For example, we might want to map authors to titles of the books they have written. In this case, there might be multiple entries for each author, so we'd use a multimap rather than a map. Because keys in a multi container need not be unique, insert on these types always inserts an element:

```
multimap<string, string> authors;
// adds the first element with the key Barth, John
authors.insert({"Barth, John", "Sot-Weed Factor"});
// ok: adds the second element with the key Barth, John
authors.insert({"Barth, John", "Lost in the Funhouse"});
```

For the containers that allow multiple keys, the `insert` operation that takes a single element returns an iterator to the new element. There is no need to return a `bool`, because `insert` always adds a new element in these types.

Exercises Section 11.3.2

Exercise 11.20: Rewrite the word-counting program from § 11.1 (p. 421) to use `insert` instead of subscripting. Which program do you think is easier to write and read? Explain your reasoning.

Exercise 11.21: Assuming `word_count` is a `map` from `string` to `size_t` and `word` is a `string`, explain the following loop:

```
while (cin >> word)
  ++word_count.insert({word, 0}).first->second;
```

Exercise 11.22: Given a `map<string, vector<int>>`, write the types used as an argument and as the return value for the version of `insert` that inserts one element.

Exercise 11.23: Rewrite the `map` that stored `vector`s of children's names with a key that is the family last name for the exercises in § 11.2.1 (p. 424) to use a `multimap`.

11.3.3 Erasing Elements

The associative containers define three versions of `erase`, which are described in Table 11.5. As with the sequential containers, we can `erase` one element or a range of elements by passing `erase` an iterator or an iterator pair. These versions of `erase` are similar to the corresponding operations on sequential containers: The indicated element(s) are removed and the function returns `void`.

The associative containers supply an additional `erase` operation that takes a `key_type` argument. This version removes all the elements, if any, with the given key and returns a count of how many elements were removed. We can use this version to remove a specific word from `word_count` before printing the results:

```
// erase on a key returns the number of elements removed
if (word_count.erase(removal_word))
     cout << "ok: " << removal_word << " removed\n";
else cout << "oops: " << removal_word << " not found!\n";
```

For the containers with unique keys, the return from `erase` is always either zero or one. If the return value is zero, then the element we wanted to erase was not in the container.

For types that allow multiple keys, the number of elements removed could be greater than one:

```
auto cnt = authors.erase("Barth, John");
```

If `authors` is the `multimap` we created in § 11.3.2 (p. 434), then `cnt` will be 2.

Table 11.5: Removing Elements from an Associative Container

`c.erase(k)`	Removes every element with key `k` from `c`. Returns `size_type` indicating the number of elements removed.
`c.erase(p)`	Removes the element denoted by the iterator `p` from `c`. `p` must refer to an actual element in `c`; it must not be equal to `c.end()`. Returns an iterator to the element after `p` or `c.end()` if `p` denotes the last element in `c`.
`c.erase(b, e)`	Removes the elements in the range denoted by the iterator pair `b`, `e`. Returns `e`.

11.3.4 Subscripting a `map`

The `map` and `unordered_map` containers provide the subscript operator and a corresponding `at` function (§ 9.3.2, p. 348), which are described in Table 11.6 (overleaf). The `set` types do not support subscripting because there is no "value" associated with a key in a `set`. The elements are themselves keys, so the operation of "fetching the value associated with a key" is meaningless. We cannot subscript a `multimap` or an `unordered_multimap` because there may be more than one value associated with a given key.

Like the other subscript operators we've used, the `map` subscript takes an index (that is, a key) and fetches the value associated with that key. However, unlike other subscript operators, if the key is not already present, *a new element is created and inserted* into the `map` for that key. The associated value is value initialized (§ 3.3.1, p. 98).

For example, when we write

```
map <string, size_t> word_count; // empty map
// insert a value-initialized element with key Anna; then assign 1 to its value
word_count["Anna"] = 1;
```

the following steps take place:

- `word_count` is searched for the element whose key is `Anna`. The element is not found.
- A new key–value pair is inserted into `word_count`. The key is a `const string` holding `Anna`. The value is value initialized, meaning in this case that the value is 0.
- The newly inserted element is fetched and is given the value 1.

Because the subscript operator might insert an element, we may use subscript only on a `map` that is not `const`.

> Subscripting a `map` behaves quite differently from subscripting an array or `vector`: Using a key that is not already present *adds* an element with that key to the `map`.

Table 11.6: Subscript Operation for `map` and `unordered_map`

`c[k]`	Returns the element with key `k`; if `k` is not in `c`, adds a new, value-initialized element with key `k`.
`c.at(k)`	Checked access to the element with key `k`; throws an `out_of_range` exception (§ 5.6, p. 193) if `k` is not in `c`.

Using the Value Returned from a Subscript Operation

Another way in which the `map` subscript differs from other subscript operators we've used is its return type. Ordinarily, the type returned by dereferencing an iterator and the type returned by the subscript operator are the same. Not so for `map`s: when we subscript a `map`, we get a `mapped_type` object; when we dereference a `map` iterator, we get a `value_type` object (§ 11.3, p. 428).

In common with other subscripts, the `map` subscript operator returns an lvalue (§ 4.1.1, p. 135). Because the return is an lvalue, we can read or write the element:

```
cout << word_count["Anna"];   // fetch the element indexed by Anna; prints 1
++word_count["Anna"];         // fetch the element and add 1 to it
cout << word_count["Anna"];   // fetch the element and print it; prints 2
```

> Unlike `vector` or `string`, the type returned by the `map` subscript operator differs from the type obtained by dereferencing a `map` iterator.

The fact that the subscript operator adds an element if it is not already in the `map` allows us to write surprisingly succinct programs such as the loop inside our word-counting program (§ 11.1, p. 421). On the other hand, sometimes we only want to know whether an element is present and *do not* want to add the element if it is not. In such cases, we must not use the subscript operator.

11.3.5 Accessing Elements

The associative containers provide various ways to find a given element, which are described in Table 11.7 (p. 438). Which operation to use depends on what problem we are trying to solve. If all we care about is whether a particular element is in the container, it is probably best to use `find`. For the containers that can hold only unique keys, it probably doesn't matter whether we use `find` or `count`. However, for the containers with multiple keys, `count` has to do more work: If the element

EXERCISES SECTION 11.3.4

Exercise 11.24: What does the following program do?

```
map<int, int> m;
m[0] = 1;
```

Exercise 11.25: Contrast the following program with the one in the previous exercise

```
vector<int> v;
v[0] = 1;
```

Exercise 11.26: What type can be used to subscript a `map`? What type does the subscript operator return? Give a concrete example—that is, define a `map` and then write the types that can be used to subscript the `map` and the type that would be returned from the subscript operator.

is present, it still has to count how many elements have the same key. If we don't need the count, it's best to use `find`:

```
set<int> iset = {0,1,2,3,4,5,6,7,8,9};
iset.find(1);   // returns an iterator that refers to the element with key == 1
iset.find(11);  // returns the iterator == iset.end()
iset.count(1);  // returns 1
iset.count(11); // returns 0
```

Using `find` Instead of Subscript for `map`s

For the `map` and `unordered_map` types, the subscript operator provides the simplest method of retrieving a value. However, as we've just seen, using a subscript has an important side effect: If that key is not already in the `map`, then subscript inserts an element with that key. Whether this behavior is correct depends on our expectations. Our word-counting programs relied on the fact that using a nonexistent key as a subscript inserts an element with that key and value 0.

Sometimes, we want to know if an element with a given key is present without changing the `map`. We cannot use the subscript operator to determine whether an element is present, because the subscript operator inserts a new element if the key is not already there. In such cases, we should use `find`:

```
if (word_count.find("foobar") == word_count.end())
    cout << "foobar is not in the map" << endl;
```

Finding Elements in a `multimap` or `multiset`

Finding an element in an associative container that requires unique keys is a simple matter—the element is or is not in the container. For the containers that allow multiple keys, the process is more complicated: There may be many elements with the given key. When a `multimap` or `multiset` has multiple elements of a given key, those elements will be adjacent within the container.

Table 11.7: Operations to Find Elements in an Associative Container

`lower_bound` and `upper_bound` not valid for the unordered containers.
Subscript and `at` operations only for `map` and `unordered_map` that are not `const`.

`c.find(k)`	Returns an iterator to the (first) element with key `k`, or the off-the-end iterator if `k` is not in the container.
`c.count(k)`	Returns the number of elements with key `k`. For the containers with unique keys, the result is always zero or one.
`c.lower_bound(k)`	Returns an iterator to the first element with key not less than `k`.
`c.upper_bound(k)`	Returns an iterator to the first element with key greater than `k`.
`c.equal_range(k)`	Returns a `pair` of iterators denoting the elements with key `k`. If `k` is not present, both members are `c.end()`.

For example, given our map from author to titles, we might want to print all the books by a particular author. We can solve this problem in three different ways. The most obvious way uses `find` and `count`:

```
string search_item("Alain de Botton"); // author we'll look for
auto entries = authors.count(search_item); // number of elements
auto iter = authors.find(search_item); // first entry for this author
// loop through the number of entries there are for this author
while(entries) {
    cout << iter->second << endl; // print each title
    ++iter;       // advance to the next title
    --entries;    // keep track of how many we've printed
}
```

We start by determining how many entries there are for the author by calling `count` and getting an iterator to the first element with this key by calling `find`. The number of iterations of the `for` loop depends on the number returned from `count`. In particular, if the `count` was zero, then the loop is never executed.

Note

We are guaranteed that iterating across a `multimap` or `multiset` returns all the elements with a given key in sequence.

A Different, Iterator-Oriented Solution

Alternatively, we can solve our problem using `lower_bound` and `upper_bound`. Each of these operations take a key and returns an iterator. If the key is in the container, the iterator returned from `lower_bound` will refer to the first instance of that key and the iterator returned by `upper_bound` will refer just after the last instance of the key. If the element is not in the `multimap`, then `lower_bound` and `upper_bound` will return equal iterators; both will refer to the point at which the key can be inserted without disrupting the order. Thus, calling `lower_bound` and `upper_bound` on the same key yields an iterator range (§ 9.2.1, p. 331) that denotes all the elements with that key.

Of course, the iterator returned from these operations might be the off-the-end iterator for the container itself. If the element we're looking for has the largest key in the container, then `upper_bound` on that key returns the off-the-end iterator. If the key is not present and is larger than any key in the container, then the return from `lower_bound` will also be the off-the-end iterator.

> The iterator returned from `lower_bound` may or may not refer to an element with the given key. If the key is not in the container, then `lower_bound` refers to the first point at which this key can be inserted while preserving the element order within the container.

Using these operations, we can rewrite our program as follows:

```
// definitions of authors and search_item as above
// beg and end denote the range of elements for this author
for (auto beg = authors.lower_bound(search_item),
          end = authors.upper_bound(search_item);
     beg != end; ++beg)
    cout << beg->second << endl; // print each title
```

This program does the same work as the previous one that used `count` and `find` but accomplishes its task more directly. The call to `lower_bound` positions `beg` so that it refers to the first element matching `search_item` if there is one. If there is no such element, then `beg` refers to the first element with a key larger than `search_item`, which could be the off-the-end iterator. The call to `upper_bound` sets `end` to refer to the element just beyond the last element with the given key. These operations say nothing about whether the key is present. The important point is that the return values act like an iterator range (§ 9.2.1, p. 331).

If there is no element for this key, then `lower_bound` and `upper_bound` will be equal. Both will refer to the point at which this key can be inserted while maintaining the container order.

Assuming there are elements with this key, `beg` will refer to the first such element. We can increment `beg` to traverse the elements with this key. The iterator in `end` will signal when we've seen all the elements. When `beg` equals `end`, we have seen every element with this key.

Because these iterators form a range, we can use a `for` loop to traverse that range. The loop is executed zero or more times and prints the entries, if any, for the given author. If there are no elements, then `beg` and `end` are equal and the loop is never executed. Otherwise, we know that the increment to `beg` will eventually reach `end` and that in the process we will print each record associated with this author.

> If `lower_bound` and `upper_bound` return the same iterator, then the given key is not in the container.

The `equal_range` Function

The remaining way to solve this problem is the most direct of the three approaches: Instead of calling `upper_bound` and `lower_bound`, we can call `equal_range`.

This function takes a key and returns a `pair` of iterators. If the key is present, then the first iterator refers to the first instance of the key and the second iterator refers one past the last instance of the key. If no matching element is found, then both the first and second iterators refer to the position where this key can be inserted.

We can use `equal_range` to modify our program once again:

```
// definitions of authors and search_item as above
// pos holds iterators that denote the range of elements for this key
for (auto pos = authors.equal_range(search_item);
     pos.first != pos.second; ++pos.first)
    cout << pos.first->second << endl; // print each title
```

This program is essentially identical to the previous one that used `upper_bound` and `lower_bound`. Instead of using local variables, `beg` and `end`, to hold the iterator range, we use the `pair` returned by `equal_range`. The `first` member of that `pair` holds the same iterator as `lower_bound` would have returned and `second` holds the iterator `upper_bound` would have returned. Thus, in this program `pos.first` is equivalent to `beg`, and `pos.second` is equivalent to `end`.

EXERCISES SECTION 11.3.5

Exercise 11.27: What kinds of problems would you use `count` to solve? When might you use `find` instead?

Exercise 11.28: Define and initialize a variable to hold the result of calling `find` on a `map` from `string` to `vector` of `int`.

Exercise 11.29: What do `upper_bound`, `lower_bound`, and `equal_range` return when you pass them a key that is not in the container?

Exercise 11.30: Explain the meaning of the operand `pos.first->second` used in the output expression of the final program in this section.

Exercise 11.31: Write a program that defines a `multimap` of authors and their works. Use `find` to find an element in the `multimap` and `erase` that element. Be sure your program works correctly if the element you look for is not in the `map`.

Exercise 11.32: Using the `multimap` from the previous exercise, write a program to print the list of authors and their works alphabetically.

11.3.6 A Word Transformation Map

We'll close this section with a program to illustrate creating, searching, and iterating across a `map`. We'll write a program that, given one `string`, transforms it into another. The input to our program is two files. The first file contains rules that we will use to transform the text in the second file. Each rule consists of a word that might be in the input file and a phrase to use in its place. The idea is that whenever the first word appears in the input, we will replace it with the corresponding phrase. The second file contains the text to transform.

If the contents of the word-transformation file are

```
brb be right back
k okay?
y why
r are
u you
pic picture
thk thanks!
l8r later
```

and the text we are given to transform is

```
where r u
y dont u send me a pic
k thk l8r
```

then the program should generate the following output:

```
where are you
why dont you send me a picture
okay? thanks! later
```

The Word Transformation Program

Our solution will use three functions. The `word_transform` function will manage the overall processing. It will take two `ifstream` arguments: The first will be bound to the word-transformation file and the second to the file of text we're to transform. The `buildMap` function will read the file of transformation rules and create a `map` from each word to its transformation. The `transform` function will take a `string` and return the transformation if there is one.

We'll start by defining the `word_transform` function. The important parts are the calls to `buildMap` and `transform`:

```
void word_transform(ifstream &map_file, ifstream &input)
{
    auto trans_map = buildMap(map_file); // store the transformations
    string text;                        // hold each line from the input
    while (getline(input, text)) {      // read a line of input
        istringstream stream(text);     // read each word
        string word;
        bool firstword = true;          // controls whether a space is printed
        while (stream >> word) {
           if (firstword)
               firstword = false;
           else
               cout << " ";  // print a space between words
           // transform returns its first argument or its transformation
           cout << transform(word, trans_map); // print the output
        }
        cout << endl;        // done with this line of input
    }
}
```

The function starts by calling `buildMap` to generate the word-transformation `map`. We store the result in `trans_map`. The rest of the function processes the `input` file. The `while` loop uses `getline` to read the input file a line at a time. We read by line so that our output will have line breaks at the same position as in the input file. To get the words from each line, we use a nested `while` loop that uses an `istringstream` (§ 8.3, p. 321) to process each word in the current line.

The inner `while` prints the output using the `bool firstword` to determine whether to print a space. The call to `transform` obtains the word to print. The value returned from `transform` is either the original `string` in `word` or its corresponding transformation from `trans_map`.

Building the Transformation Map

The `buildMap` function reads its given file and builds the transformation `map`.

```
map<string, string> buildMap(ifstream &map_file)
{
    map<string, string> trans_map;    // holds the transformations
    string key;     // a word to transform
    string value;   // phrase to use instead
    // read the first word into key and the rest of the line into value
    while (map_file >> key && getline(map_file, value))
        if (value.size() > 1) // check that there is a transformation
            trans_map[key] = value.substr(1); // skip leading space
        else
            throw runtime_error("no rule for " + key);
    return trans_map;
}
```

Each line in `map_file` corresponds to a rule. Each rule is a word followed by a phrase, which might contain multiple words. We use `>>` to read the word that we will transform into `key` and call `getline` to read the rest of the line into `value`. Because `getline` does not skip leading spaces (§ 3.2.2, p. 87), we need to skip the space between the word and its corresponding rule. Before we store the transformation, we check that we got more than one character. If so, we call `substr` (§ 9.5.1, p. 361) to skip the space that separated the transformation phrase from its corresponding word and store that substring in `trans_map`,

Note that we use the subscript operator to add the key–value pairs. Implicitly, we are ignoring what should happen if a word appears more than once in our transformation file. If a word does appear multiple times, our loops will put the last corresponding phrase into `trans_map`. When the `while` concludes, `trans_map` contains the data that we need to transform the input.

Generating a Transformation

The `transform` function does the actual transformation. Its parameters are references to the `string` to transform and to the transformation `map`. If the given `string` is in the `map`, `transform` returns the corresponding transformation. If the given `string` is not in the `map`, `transform` returns its argument:

```
const string &
transform(const string &s, const map<string, string> &m)
{
    // the actual map work; this part is the heart of the program
    auto map_it = m.find(s);
    // if this word is in the transformation map
    if (map_it != m.cend())
        return map_it->second;   // use the replacement word
    else
        return s;                // otherwise return the original unchanged
}
```

We start by calling `find` to determine whether the given `string` is in the `map`. If it is, then `find` returns an iterator to the corresponding element. Otherwise, `find` returns the off-the-end iterator. If the element is found, we dereference the iterator, obtaining a `pair` that holds the key and value for that element (§ 11.3, p. 428). We return the `second` member, which is the transformation to use in place of `s`.

Exercises Section 11.3.6

Exercise 11.33: Implement your own version of the word-transformation program.

Exercise 11.34: What would happen if we used the subscript operator instead of `find` in the `transform` function?

Exercise 11.35: In `buildMap`, what effect, if any, would there be from rewriting

```
trans_map[key] = value.substr(1);
```

as `trans_map.insert({key, value.substr(1)})`?

Exercise 11.36: Our program does no checking on the validity of either input file. In particular, it assumes that the rules in the transformation file are all sensible. What would happen if a line in that file has a key, one space, and then the end of the line? Predict the behavior and then check it against your version of the program.

11.4 The Unordered Containers

The new standard defines four **unordered associative containers**. Rather than using a comparison operation to organize their elements, these containers use a ***hash function*** and the key type's `==` operator. An unordered container is most useful when we have a key type for which there is no obvious ordering relationship among the elements. These containers are also useful for applications in which the cost of maintaining the elements in order is prohibitive.

Although hashing gives better average case performance in principle, achieving good results in practice often requires a fair bit of performance testing and tweaking. As a result, it is usually easier (and often yields better performance) to use an ordered container.

Use an unordered container if the key type is inherently unordered or if performance testing reveals problems that hashing might solve.

Using an Unordered Container

Aside from operations that manage the hashing, the unordered containers provide the same operations (`find`, `insert`, and so on) as the ordered containers. That means that the operations we've used on `map` and `set` apply to `unordered_map` and `unordered_set` as well. Similarly for the unordered versions of the containers that allow multiple keys.

As a result, we can usually use an unordered container in place of the corresponding ordered container, and vice versa. However, because the elements are not stored in order, the output of a program that uses an unordered container will (ordinarily) differ from the same program using an ordered container.

For example, we can rewrite our original word-counting program from § 11.1 (p. 421) to use an `unordered_map`:

```
// count occurrences, but the words won't be in alphabetical order
unordered_map<string, size_t> word_count;
string word;
while (cin >> word)
    ++word_count[word]; // fetch and increment the counter for word
for (const auto &w : word_count) // for each element in the map
    // print the results
    cout << w.first << " occurs " << w.second
         << ((w.second > 1) ? " times" : " time") << endl;
```

The type of `word_count` is the only difference between this program and our original. If we run this version on the same input as our original program,

```
containers. occurs 1 time
use occurs 1 time
can occurs 1 time
examples occurs 1 time
...
```

we'll obtain the same count for each word in the input. However, the output is unlikely to be in alphabetical order.

Managing the Buckets

The unordered containers are organized as a collection of buckets, each of which holds zero or more elements. These containers use a hash function to map elements to buckets. To access an element, the container first computes the element's hash code, which tells which bucket to search. The container puts all of its elements with a given hash value into the same bucket. If the container allows multiple elements with a given key, all the elements with the same key will be in the same bucket. As a result, the performance of an unordered container depends on the quality of its hash function and on the number and size of its buckets.

The hash function must always yield the same result when called with the same argument. Ideally, the hash function also maps each particular value to a unique bucket. However, a hash function is allowed to map elements with differing keys to the same bucket. When a bucket holds several elements, those elements are searched sequentially to find the one we want. Typically, computing an element's hash code and finding its bucket is a fast operation. However, if the bucket has many elements, many comparisons may be needed to find a particular element.

The unordered containers provide a set of functions, listed in Table 11.8, that let us manage the buckets. These members let us inquire about the state of the container and force the container to reorganize itself as needed.

Table 11.8: Unordered Container Management Operations

Bucket Interface	
`c.bucket_count()`	Number of buckets in use.
`c.max_bucket_count()`	Largest number of buckets this container can hold.
`c.bucket_size(n)`	Number of elements in the `n`th bucket.
`c.bucket(k)`	Bucket in which elements with key `k` would be found.
Bucket Iteration	
`local_iterator`	Iterator type that can access elements in a bucket.
`const_local_iterator`	`const` version of the bucket iterator.
`c.begin(n), c.end(n)`	Iterator to the first, one past the last element in bucket `n`.
`c.cbegin(n), c.cend(n)`	Returns `const_local_iterator`.
Hash Policy	
`c.load_factor()`	Average number of elements per bucket. Returns `float`.
`c.max_load_factor()`	Average bucket size that `c` tries to maintain. `c` adds buckets to keep `load_factor` <= `max_load_factor`. Returns `float`.
`c.rehash(n)`	Reorganize storage so that `bucket_count` >= `n` and and `bucket_count` > `size/max_load_factor`.
`c.reserve(n)`	Reorganize so that `c` can hold `n` elements without a `rehash`.

Requirements on Key Type for Unordered Containers

By default, the unordered containers use the `==` operator on the key type to compare elements. They also use an object of type `hash<key_type>` to generate the hash code for each element. The library supplies versions of the **`hash`** template for the built-in types, including pointers. It also defines `hash` for some of the library types, including `string`s and the smart pointer types that we will describe in Chapter 12. Thus, we can directly define unordered containers whose key is one of the built-in types (including pointer types), or a `string`, or a smart pointer.

However, we cannot directly define an unordered container that uses a our own class types for its key type. Unlike the containers, we cannot use the hash template directly. Instead, we must supply our own version of the `hash` template. We'll see how to do so in § 16.5 (p. 709).

Instead of using the default `hash`, we can use a strategy similar to the one we used to override the default comparison operation on keys for the ordered containers (§ 11.2.2, p. 425). To use `Sales_data` as the key, we'll need to supply functions to replace both the `==` operator and to calculate a hash code. We'll start by defining these functions:

```
size_t hasher(const Sales_data &sd)
{
    return hash<string>()(sd.isbn());
}
bool eqOp(const Sales_data &lhs, const Sales_data &rhs)
{
    return lhs.isbn() == rhs.isbn();
}
```

Our `hasher` function uses an object of the library `hash` of `string` type to generate a hash code from the ISBN member. Similarly, the `eqOp` funciton compares two `Sales_data` objects by comparing their ISBNs.

We can use these functions to define an `unordered_multiset` as follows

```
using SD_multiset = unordered_multiset<Sales_data,
                    decltype(hasher)*, decltype(eqOp)*>;
// arguments are the bucket size and pointers to the hash function and equality operator
SD_multiset bookstore(42, hasher, eqOp);
```

To simplify the declaration of `bookstore` we first define a type alias (§ 2.5.1, p. 67) for an `unordered_multiset` whose hash and equality operations have the same types as our `hasher` and `eqOp` functions. Using that type, we define `bookstore` passing pointers to the functions we want `bookstore` to use.

If our class has its own `==` operator we can override just the hash function:

```
// use FooHash to generate the hash code; Foo must have an == operator
unordered_set<Foo, decltype(FooHash)*> fooSet(10, FooHash);
```

Exercises Section 11.4

Exercise 11.37: What are the advantages of an unordered container as compared to the ordered version of that container? What are the advantages of the ordered version?

Exercise 11.38: Rewrite the word-counting (§ 11.1, p. 421) and word-transformation (§ 11.3.6, p. 440) programs to use an `unordered_map`.

Chapter Summary

The associative containers support efficient lookup and retrieval of elements by key. The use of a key distinguishes the associative containers from the sequential containers, in which elements are accessed positionally.

There are eight associative containers, each of which

- Is a `map` or a `set`. a `map` stores key–value pairs; a `set` stores only keys.
- Requires unique keys or not.
- Keeps keys in order or not.

Ordered containers use a comparison function to order the elements by key. By default, the comparison is the `<` operator on the keys. Unordered containers use the key type's `==` operator and an object of type `hash<key_type>` to organize their elements.

Containers with nonunique keys include the word `multi` in their names; those that use hashing start with the word `unordered`. A `set` is an ordered collection in which each key may appear only once; an `unordered_multiset` is an unordered collection of keys in which the keys can appear multiple times.

The associative containers share many operations with the sequential containers. However, the associative containers define some new operations and redefine the meaning or return types of some operations common to both the sequential and associative containers. The differences in the operations reflect the use of keys in associative containers.

Iterators for the ordered containers access elements in order by key. Elements with the same key are stored adjacent to one another in both the ordered and unordered containers.

Defined Terms

associative array Array whose elements are indexed by key rather than positionally. We say that the array maps a key to its associated value.

associative container Type that holds a collection of objects that supports efficient lookup by key.

hash Special library template that the unordered containers use to manage the position of their elements.

hash function Function that maps values of a given type to integral (`size_t`) values. Equal values must map to equal integers; unequal values should map to unequal integers where possible.

key_type Type defined by the associative containers that is the type for the keys used to store and retrieve values. For a `map`, `key_type` is the type used to index the `map`. For `set`, `key_type` and `value_type` are the same.

map Associative container type that defines an associative array. Like `vector`, `map` is a class template. A `map`, however, is defined with two types: the type of the key and the type of the associated value. In a `map`, a given key may appear only once. Each key is associated with a particular value. Dereferencing a `map` iterator yields a `pair` that holds a `const` key and its associated value.

mapped_type Type defined by map types that is the type of the values associated with the keys in the map.

multimap Associative container similar to `map` except that in a `multimap`, a given key may appear more than once. `multimap` does not support subscripting.

multiset Associative container type that holds keys. In a `multiset`, a given key may appear more than once.

pair Type that holds two `public` data members named `first` and `second`. The `pair` type is a template type that takes two type parameters that are used as the types of these members.

set Associative container that holds keys. In a `set`, a given key may appear only once.

strict weak ordering Relationship among the keys used in an associative container. In a strict weak ordering, it is possible to compare any two values and determine which of the two is less than the other. If neither value is less than the other, then the two values are considered equal.

unordered container Associative containers that use hashing rather than a comparison operation on keys to store and access elements. The performance of these containers depends on the quality of the hash function.

unordered_map Container with elements that are key–value pairs, permits only one element per key.

unordered_multimap Container with elements that are key–value pairs, allows multiple elements per key.

unordered_multiset Container that stores keys, allows multiple elements per key.

unordered_set Container that stores keys, permits only one element per key.

value_type Type of the element stored in a container. For `set` and `multiset`, `value_type` and `key_type` are the same. For `map` and `multimap`, this type is a `pair` whose `first` member has type `const key_type` and whose `second` member has type `mapped_type`.

`*` operator Dereference operator. When applied to a `map`, `set`, `multimap`, or `multiset` iterator `*` yields a `value_type`. Note, that for `map` and `multimap`, the `value_type` is a `pair`.

`[ ]` operator Subscript operator. Defined only for non`const` obejcts of type `map` and `unordered_map`. For the map types, `[]` takes an index that must be a `key_type` (or type that can be converted to `key_type`). Yields a `mapped_type` value.