6.2. Containers

Container classes, or *containers* for short, manage a collection of elements. To meet different needs, the STL provides different kinds of containers, as shown in Figure 6.2.

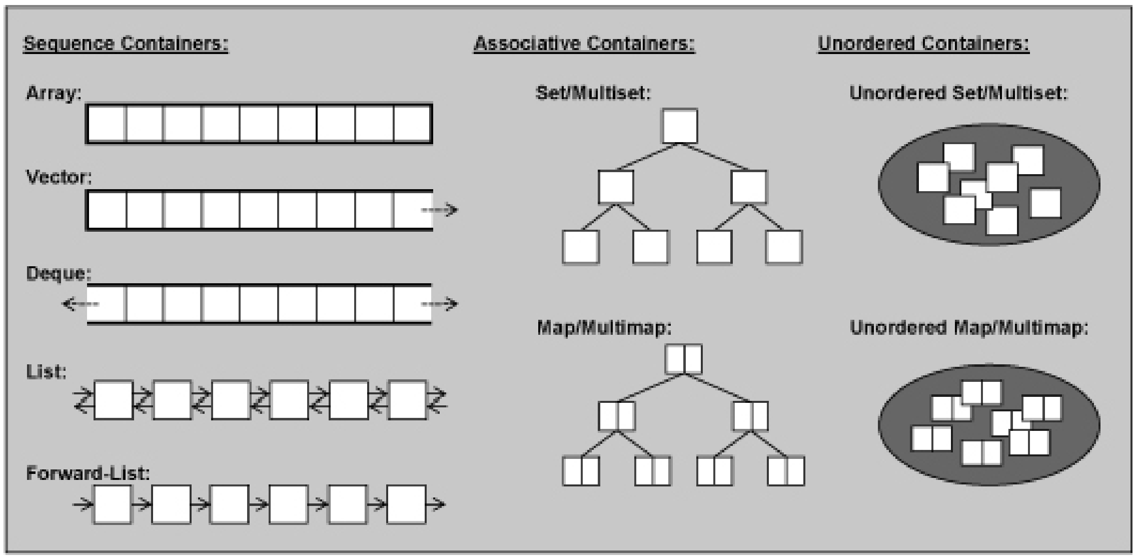

Figure 6.2. STL Container Types

There are three general kinds of containers:

1. **Sequence containers** are *ordered collections* in which every element has a certain position. This position depends on the time and place of the insertion, but it is independent of the value of the element. For example, if you put six elements into an ordered collection by appending each element at the end of the collection, these elements are in the exact order in which you put them. The STL contains five predefined sequence container classes: `array`, `vector`, `deque`, `list`, and `forward_list`.[1]

[1] Class `array` was added with TR1; `forward_list` was added with C++11.

2. **Associative containers** are *sorted collections* in which the position of an element depends on its value (or key, if it's a key/value pair) due to a certain sorting criterion. If you put six elements into a collection, their value determines their order. The order of insertion doesn't matter. The STL contains four predefined associative container classes: `set`, `multiset`, `map`, and `multimap`.
3. **Unordered (associative) containers** are *unordered collections* in which the position of an element doesn't matter. The only important question is whether a specific element is in such a collection. Neither the order of insertion nor the value of the inserted element has an influence on the position of the element, and the position might change over the lifetime of the container. Thus, if you put six elements into a collection, their order is undefined and might change over time. The STL contains four predefined unordered container classes: `unordered_set`, `unordered_multiset`, `unordered_map`, and `unordered_multimap`.

Unordered containers were introduced with TR1 and created a bit of confusion in container terminology. Officially, unordered containers are categorized as "unordered associative containers." For this reason, it's a bit unclear, what is meant by "associative container": Is it a general term of (ordered) associative containers and unordered associative containers, or is it the counterpart of unordered containers? The answer often depends on the context. Throughout this book, I mean the "old" sorted associative containers when I use the term "associative containers" and use the term "unordered containers" without "associative" in the middle.

The three container categories introduced here are just logical categories according to the way the order of elements is defined. According to this point of view, an associative container can be considered a special kind of sequence container because sorted collections have the additional ability to be ordered according to a sorting criterion. You might expect this, especially if you have used other libraries of collection classes, such as those in Smalltalk or the NIHCL,[2] in which sorted collections are derived from ordered collections. However, the STL collection types are completely distinct from one another and have very different implementations that are not derived from one another. As we will see:

[2] The National Institutes of Health's Class Library was one of the first class libraries in C++.

- Sequence containers are usually implemented as arrays or linked lists.
- Associative containers are usually implemented as binary trees.
- Unordered containers are usually implemented as hash tables.

Strictly speaking, the particular implementation of any container is not defined by the C++ standard library. However, the behavior and complexity specified by the standard do not leave much room for variation. So, in practice, the implementations differ only in minor details.

When choosing the right container, abilities other than the order of elements might be taken into account. In fact, the automatic sorting of elements in associative containers does *not* mean that those containers are especially designed for sorting elements. You can also sort the elements of a sequence container. The key advantage of automatic sorting is better performance when you search elements. In particular, you can always use a binary search, which results in logarithmic complexity rather than linear complexity. For example, this means that for a search in a collection of 1,000 elements, you need, on average, only 10 instead of 500 comparisons (see Section 2.2, page 10). Thus, automatic sorting is only a (useful) “side effect” of the implementation of an associative container, designed to enable better performance.

The following subsections discuss the container classes in detail: how containers are typically implemented and the benefits and drawbacks this introduces. Chapter 7 covers the exact behavior of the container classes, describing their common and individual abilities and member functions in detail. Section 7.12, page 392, discusses in detail when to use which container.

6.2.1. Sequence Containers

The following sequence containers are predefined in the STL:

- Arrays (a class called `array`)
- Vectors
- Deques
- Lists (singly and doubly linked)

We start with the discussion of vectors because `array`s came later, with TR1, into the C++ standard and have some special properties that are not common for STL containers in general.

Vectors

A vector manages its elements in a dynamic array. It enables random access, which means that you can access each element directly with the corresponding index. Appending and removing elements at the end of the array is very fast.[3] However, inserting an element in the middle or at the beginning of the array takes time because all the following elements have to be moved to make room for it while maintaining the order.

[3] Strictly speaking, appending elements is *amortized* very fast. An individual append may be slow when a vector has to reallocate new memory and to copy existing elements into the new memory. However, because such reallocations are rather rare, the operation is very fast in the long term. See Section 2.2, page 10, for a discussion of complexity.

The following example defines a vector for integer values, inserts six elements, and prints the elements of the vector:

```
// stl/vector1.cpp

#include <vector>
#include <iostream>
using namespace std;

int main()
{
    vector<int> coll;     // vector container for integer elements

    // append elements with values 1 to 6
    for (int i=1; i<=6; ++i) {
        coll.push_back(i);
    }

    // print all elements followed by a space
    for (int i=0; i<coll.size(); ++i) {
        cout << coll[i] << ' ';
    }
    cout << endl;
}
```

The header file for vectors is included with

```
#include <vector>
```

The following declaration creates a vector for elements of type `int`:

```
vector<int> coll;
```

The vector is not initialized by any value, so the default constructor creates it as an empty collection. The `push_back()` function appends an element to the container:

```
coll.push_back(i);
```

This member function is provided for all sequence containers, where appending an element is possible and reasonably fast.

The `size()` member function returns the number of elements of a container:

```
for (int i=0; i<coll.size(); ++i) {
        ...
    }
```

`size()` is provided for any container class except singly linked lists (class `forward_list`). By using the subscript operator `[]`, you can access a single element of a vector:

```
    cout << coll[i] << ' ';
```

Here, the elements are written to the standard output, so the output of the whole program is as follows:

```
    1 2 3 4 5 6
```

Deques

The term *deque* (it rhymes with "check"[4]) is an abbreviation for "double-ended queue." It is a dynamic array that is implemented so that it can grow in both directions. Thus, inserting elements at the end *and* at the beginning is fast. However, inserting elements in the middle takes time because elements must be moved.

[4] It is only a mere accident that "deque" also sounds like "hack" :-) .

The following example declares a deque for floating-point values, inserts elements from 1.1 to 6.6 at the front of the container, and prints all elements of the deque:

```
    // stl/deque1.cpp

    #include <deque>
    #include <iostream>
    using namespace std;

    int main()
    {
        deque<float> coll;     // deque container for floating-point elements
        // insert elements from 1.1 to 6.6 each at the front
        for (int i=1; i<=6; ++i) {
            coll.push_front(i*1.1);         // insert at the front
        }

        // print all elements followed by a space
        for (int i=0; i<coll.size(); ++i) {
            cout << coll[i] << ' ';
        }
        cout << endl;
    }
```

In this example, the header file for deques is included with

```
    #include <deque>
```

The following declaration creates an empty collection of floating-point values:

```
    deque<float> coll;
```

Here, the `push_front()` member function is used to insert elements:

```
    coll.push_front(i*1.1);
```

`push_front()` inserts an element at the front of the collection. This kind of insertion results in a reverse order of the elements because each element gets inserted in front of the previous inserted elements. Thus, the output of the program is as follows:

```
    6.6 5.5 4.4 3.3 2.2 1.1
```

You could also insert elements in a deque by using the `push_back()` member function. The `push_front()` function, however, is not provided for vectors, because it would have a bad running time for them (if you insert an element at the front of a vector, all elements have to be moved). Usually, the STL containers provide only those special member functions that in general have "good" performance, where "good" normally means constant or logarithmic complexity. This prevents a programmer from calling a function that might cause bad performance.

Nevertheless, it *is* possible to insert an element at the beginning of a vector — as it is possible to insert an element in the middle of both vectors and deques — by using a general insert function we will come to later.

Arrays

An array (an object of class `array<>`)[5] manages its elements in an array of fixed size (sometimes called a "static array" or "C array"). Thus, you can't change the number of elements but only their values. Consequently, you have to specify its size at creation time. An array also enables random access, which means that you can access each element directly with the corresponding index.

[5] Class `array<>` was introduced with TR1.

The following example defines an array for string values:

```
// stl/array1.cpp

#include <array>
#include <string>
#include <iostream>
using namespace std;

int main()
{
        // array container of 5 string elements:
        array<string,5> coll = { "hello", "world" };

        // print each element with its index on a line
        for (int i=0; i<coll.size(); ++i) {
            cout << i << ": " << coll[i] << endl;
        }
}
```

The header file for arrays is included with

```
#include <array>
```

The following declaration creates an array for five elements of type `string` :

```
array<string,5> coll
```

By default, these elements are initialized with the default constructor of the element's type. This means that for fundamental data types, the initial value is undefined.

However, in this program, an initializer list (see Section 3.1.3, page 15) is used, which allows initializing class objects at creation time by a list of values. Since C++11, such a way of initialization is provided by every container, so we could also use it for vectors and deques. In that case, for fundamental data types *zero initialization* is used, which means that fundamental data types are guaranteed to be initialized with 0 (see Section 3.2.1, page 37).

Here, by using `size()` and the subscript operator `[]` , all elements are written with their index line-by-line to the standard output. The output of the whole program is as follows:

```
0: hello
1: world
2:
3:
4:
```

As you can see, the program outputs five lines, because we have an array with five strings defined. According to the initializer list, the first two elements were initialized with `"hello"` and `"world"` , and the remaining elements have their default value, which is the empty string.

Note that the number of elements is a part of the type of an array. Thus, `array<int,5>` and `array<int,10>` are two different types, and you can't assign or compare them as a whole.

Lists

Historically, we had only one list class in C++98. However, since C++11, two different list containers are provided by the STL: class `list<>` and class `forward_list<>` . Thus, the term *list* might refer to the specific class or be a general term for both list classes. However, to some extent, a forward list is just a restricted list and, in practice, this difference is not so important. So, when I use the term *list* I usually mean class `list<>` , which nevertheless often implies that abilities also apply to class `forward_list<>` . For specifics of class `forward_list<>` , I use the term *forward list*. So, this subsection discusses "ordinary" lists, which have been part of the STL since the beginning.

A `list<>` is implemented as a doubly linked list of elements. This means each element in the list has its own segment of memory and refers to its predecessor and its successor.

Lists do not provide random access. For example, to access the tenth element, you must navigate the first nine elements by following the chain of their links. However, a step to the next or previous element is possible in constant time. Thus, the general access to an arbitrary element takes linear time because the average distance is proportional to the number of elements. This is a lot worse than the constant time provided by vectors, deques, and arrays.

The advantage of a list is that the insertion or removal of an element is fast at any position. Only the links must be changed. This implies that moving an element in the middle of a list is very fast compared to moving an element in a vector or a deque.

The following example creates an empty list of characters, inserts all characters from `'a'` to `'z'` , and prints all elements:

```
// stl/list1.cpp

#include <list>
#include <iostream>
using namespace std;
```

```
int main()
{
    list<char> coll;          // list container for character elements

    // append elements from 'a' to 'z'
    for (char c='a'; c<='z'; ++c) {
        coll.push_back(c);
    }

    // print all elements:
    // - use range-based for loop
    for (auto elem : coll) {
        cout << elem << ' ';
    }
    cout << endl;
}
```

As usual, the header file for lists, `<list>`, is used to define a collection of type `list` for character values:

```
list<char> coll;
```

To print all elements, a range-based `for` loop is used, which is available since C++11 and allows performing statements with each element (see Section 3.1.4, page 17). A direct element access by using operator `[]` is not provided for lists. This is because lists don't provide random access, and so an operator `[]` would have bad performance.

Inside the loop, `auto` is used to declare the type of the `coll` element currently being processed. Thus, the type of `elem` is automatically deduced as `char` because `coll` is a collection of `char` s(see Section 3.1.2, page 14, for details of type deduction with `auto`). Instead, you could also explicitly declare the type of `elem`:

```
for (char elem : coll) {
    ...
}
```

Note that `elem` is always a copy of the element currently processed. Thus, you can modify it, but this would have an effect only for the statements called for this element. Inside `coll`, nothing gets modified. To modify the elements in the passed collection, you have to declare `elem` to be a nonconstant reference:

```
for (auto& elem : coll) {
    ...    // any modification of elem modifies the current element in coll
}
```

As for function parameters, you should generally use a constant reference to avoid a copy operation. Thus, the following function template outputs all elements of a passed container:

```
template <typename T>
void printElements (const T& coll)
{
    for (const auto& elem : coll) {
        std::cout << elem << std::endl;
    }
}
```

Before C++11, you had to use iterators to access all elements. Iterators are introduced later, so you will find a corresponding example in Section 6.3, page 189.

However, another way to "print" all elements before C++11 (without using iterators) is to print and remove the first element while there are elements in the list:

```
// stl/list2.cpp

#include <list>
#include <iostream>
using namespace std;

int main()
{
    list<char> coll;          // list container for character elements

    // append elements from 'a' to 'z'
    for (char c='a'; c<='z'; ++c) {
        coll.push_back(c);
    }

    // print all elements
```

```
        // - while there are elements
        // - print and remove the first element
        while (! coll.empty()) {
            cout << coll.front() << ' ';
            coll.pop_front();
        }
        cout << endl;
    }
```

The `empty()` member function returns whether the container has no elements. The loop continues as long as it returns `false` (that is, the container contains elements):

```
    while (! coll.empty()) {
        ...
    }
```

Inside the loop, the `front()` member function returns the first element:

```
    cout << coll.front() << ' ';
```

The `pop_front()` function removes the first element:

```
    coll.pop_front();
```

Note that `pop_front()` does not return the element it removed. Thus, you can't combine the previous two statements into one.

The output of the program depends on the character set in use. For the ASCII character set, it is as follows:[6]

[6] For other character sets, the output may contain characters that aren't letters, or it may even be empty (if `'z'` is not greater than `'a'`).

```
    a b c d e f g h i j k l m n o p q r s t u v w x y z
```

Forward Lists

Since C++11, the C++ standard library provides an additional list container: a forward list. A `forward_list<>` is implemented as a *singly* linked list of elements. As in an ordinary `list`, each element has its own segment of memory, but to save memory the element refers only to its successor.

As a consequence, a forward list is in principle just a limited list, where all operations that move backward or that would cause a performance penalty are not supported. For this reason, member functions such as `push_back()` and even `size()` are not provided.

In practice, this limitation is even more awkward than it sounds. One problem is that you can't search for an element and then delete it or insert another element in front of it. The reason is that to delete an element, you have to be at the position of the preceding element, because that is the element that gets manipulated to get a new successor. As a consequence, forward lists provide special member functions, discussed in Section 7.6.2, page 305.

Here is a small example of forward lists:

```
    // stl/forwardlist1.cpp

    #include <forward_list>
    #include <iostream>
    using namespace std;

    int main()
    {
        // create forward-list container for some prime numbers
        forward_list<long> coll = { 2, 3, 5, 7, 11, 13, 17 };

        // resize two times
        // - note: poor performance
        coll.resize(9);
        coll.resize(10,99);

        // print all elements:
        for (auto elem : coll) {
            cout << elem << ' ';
        }
        cout << endl;
    }
```

As usual, the header file for forward lists, `<forward_list>`, is used to be able to define a collection of type `forward_list` for long integer values, initialized by some prime numbers:

```
    forward_list<long> coll = { 2, 3, 5, 7, 11, 13, 17 };
```

Then `resize()` is used to change the number of elements. If the size grows, you can pass an additional parameter to specify the value of the new elements. Otherwise, the default value (zero for fundamental types) is used. Note that calling `resize()` is really an expensive operation here. It has linear complexity because to reach the end, you have to go element-by-element through the whole list. But this is one of the operations almost all sequence containers provide, ignoring possible bad performance (only `array` s do not provide `resize()` , because their size is constant).

As for lists, we use a range-based `for` loop to print all elements. The output is as follows:

```
2 3 5 7 11 13 17 0 0 99
```

6.2.2. Associative Containers

Associative containers sort their elements automatically according to a certain ordering criterion. The elements can be either values of any type or key/value pairs. For key/value pairs, each key, which might be of any type, maps to an associated value, which might be of any type. The criterion to sort the elements takes the form of a function that compares either the value or, if it's a key/value pair, the key. By default, the containers compare the elements or the keys with operator `<` . However, you can supply your own comparison function to define another ordering criterion.

Associative containers are typically implemented as binary trees. Thus, every element (every node) has one parent and two children. All ancestors to the left have lesser values; all ancestors to the right have greater values. The associative containers differ in the kinds of elements they support and how they handle duplicates.

The major advantage of associative containers is that finding an element with a specific value is rather fast because it has logarithmic complexity (in all sequence containers, you have linear complexity). Thus, when using associative containers, with 1,000 elements you have 10 instead of 500 comparisons on average. However, a drawback is that you can't modify values directly, because doing so would corrupt the automatic sorting of the elements.

The following associative containers are predefined in the STL:

- A **set** is a collection in which elements are sorted according to their own values. Each element may occur only once, so duplicates are not allowed.
- A **multiset** is the same as a set except that duplicates are allowed. Thus, a multiset may contain multiple elements that have the same value.
- A **map** contains elements that are key/value pairs. Each element has a key that is the basis for the sorting criterion and a value. Each key may occur only once, so duplicate keys are not allowed. A map can also be used as an *associative array*, an array that has an arbitrary index type (see Section 6.2.4, page 185, for details).
- A **multimap** is the same as a map except that duplicates are allowed. Thus, a multimap may contain multiple elements that have the same key. A multimap can also be used as *dictionary* (See Section 7.8.5, page 348, for an example).

All these associative container classes have an optional template argument for the sorting criterion. The default sorting criterion is the operator `<` . The sorting criterion is also used as the test for equivalence;[7] that is, two elements are duplicates if neither of their values/keys is less than the other.

[7] Note that I use the term *equivalent* here, not *equal*, which usually implies using operator `==` for the element as a whole.

You can consider a set as a special kind of map, in which the value is identical to the key. In fact, all these associative container types are usually implemented by using the same basic implementation of a binary tree.

Examples of Using Sets and Multisets

Here is a first example, using a multiset:

```
// stl/multiset1.cpp

#include <set>
#include <string>
#include <iostream>
using namespace std;

int main()
{
    multiset<string> cities {
        "Braunschweig", "Hanover", "Frankfurt", "New York",
        "Chicago", "Toronto", "Paris", "Frankfurt"
    };

    // print each element:
    for (const auto& elem : cities) {
        cout << elem << "   ";
    }
    cout << endl;

    // insert additional values:
    cities.insert( {"London", "Munich", "Hanover", "Braunschweig"} );

    // print each element:
    for (const auto& elem : cities) {
        cout << elem << "  ";
    }
    cout << endl;
}
```

After declaring the set types in the header file `<set>`, we can declare `cities` being a multiset of strings:

```
multiset<string> cities
```

With the declaration, a couple of elements are passed for initialization and later inserted using an initializer list (see Section 3.1.3, page 15). To print all the elements, we use a range-based `for` loop (see Section 3.1.4, page 17). Note that we declare the elements to be `const auto&`, which means that we derive the type of the elements from the container (see Section 3.1.2, page 14) and avoid having to create a copy for each element the body of the loop is called for.

Internally, all the elements are sorted, so the first output is as follows:

```
Braunschweig  Chicago  Frankfurt  Frankfurt  Hanover  New York
  Paris  Toronto
```

The second output is:

```
Braunschweig  Braunschweig  Chicago  Frankfurt  Frankfurt  Hanover
  Hanover  London  Munich  New York  Paris  Toronto
```

As you can see, because we use a multiset rather than a set, duplicates are allowed. If we had declared a set instead of a multiset, each value would be printed only once. If we were to use an unordered multiset, the order of the elements would be undefined (see Section 6.2.3, page 182).

Examples of Using Maps and Multimaps

The following example demonstrates the use of maps and multimaps:

```
// stl/multimap1.cpp

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

int main()
{
    multimap<int,string> coll;          // container for int/string values

    // assign some elements in arbitrary order
    // - a value with key 1 gets inserted twice
    coll = { {5,"tagged"},
             {2,"a"},
             {1,"this"},
             {4,"of"},
             {6,"strings"},
             {1,"is"},
             {3,"multimap"} };

    // print all element values
    // - element member second is the value
    for (auto elem : coll)  {
        cout << elem.second  << ' ';
    }
    cout << endl;
}
```

After including `<map>`, a map with elements that have an `int` as the key and a string as value gets declared:

```
multimap<int,string> coll;
```

Because the elements of maps and multimaps are key/value pairs, the declaration, the insertion, and the access to elements are a bit different:

- First, to initialize (or assign or insert) elements, you have to pass key/value pairs, which is done here by assigning nested initializer lists. The inner lists define the key and the value of each element; the outer list groups all these elements. Thus, `{ 5,"tagged" }` specifies the first element inserted.
- When processing the elements, you again have to deal with key/value pairs. In fact, the type of an element is `pair<const` *key* `,` *value* `>` (type `pair` is introduced in Section 5.1.1, page 60). The key is constant because any modification of its value would break the order of the elements, which are automatically sorted by the container. Because `pair`s don't have an output operator, you can't print them as a whole. Instead, you must access the members of the `pair` structure, which are

called `first` and `second` .

Thus, the following expression yields the second part of the key/value pair, which is the value of the multimap element:

```
elem.second
```

Similarly, the following expression yields the first part of the key/value pair, which is the key of the multimap element:

```
elem.first
```

As a result, the program has the following output:

```
this is a multimap of tagged strings
```

Before C++11, there was no clear guarantee for the order of equivalent elements (elements having an equal key). So, until C++11, the order of `"this"` and `"is"` might be the other way around. C++11 guarantees that newly inserted elements are inserted at the end of equivalent elements that multisets and multimaps already contain. In addition, the order of equivalent elements is guaranteed to remain stable if `insert()` , `emplace()` , or `erase()` is called.

Other Examples for Associative Containers

Section 6.2.4, page 185, gives an example for using a map, which can be used as a so-called *associative array*.

Section 7.7 discusses sets and multisets in detail, with additional examples. Section 7.8 discusses maps and multimaps in detail, with additional examples.

Multimaps can also be used as *dictionaries*. See Section 7.8.5, page 348, for an example.

6.2.3. Unordered Containers

In unordered containers, elements have no defined order. Thus, if you insert three elements, they might have any order when you iterate over all the elements in the container. If you insert a fourth element, the order of the elements previously inserted might change. The only important fact is that a specific element is *somewhere* in the container. Even when you have two containers with equal elements inside, the order might be different. Think of it as like a bag.

Unordered containers are typically implemented as a hash table (Figure 6.3). Thus, internally, the container is an array of linked lists. Using a *hash function*, the position of an element in the array gets processed. The goal is that each element has its own position so that you have fast access to each element, provided that the hash function is fast. But because such a fast perfect hash function is not always possible or might require that the array consumes a huge amount of memory, multiple elements might have the same position. For this reason, the elements in the array are linked lists so that you can store more than one element at each array position.

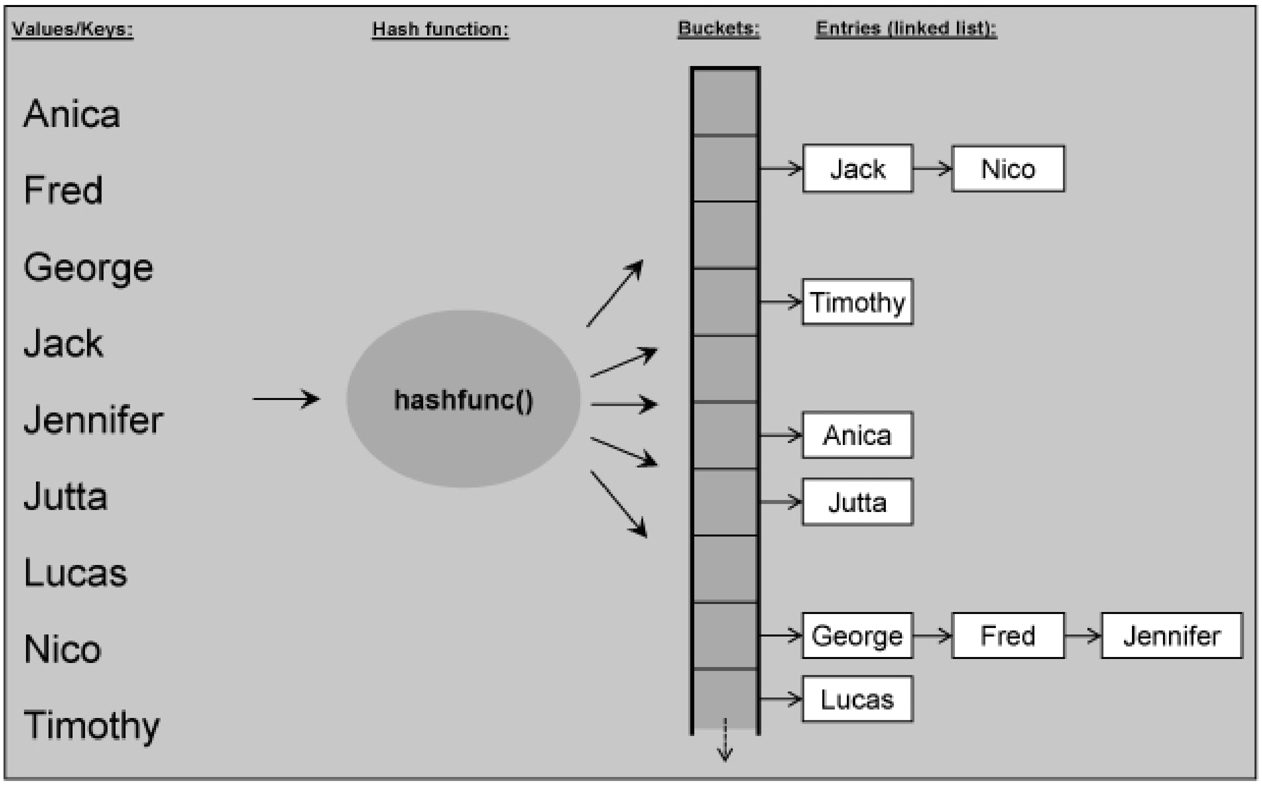

Figure 6.3. Unordered Containers Are Hash Tables

The major advantage of unordered containers is that finding an element with a specific value is even faster than for associative containers. In fact, the use of unordered containers provides amortized constant complexity, provided that you have a good hash function. However, providing a good hash function is not easy (see Section 7.9.2, page 363), and you might need a lot of memory for the buckets.

Analogous to associative containers, the following unordered containers are predefined in the STL:

- An **unordered set** is a collection of unordered elements, where each element may occur only once. Thus, duplicates are not allowed.

- An **unordered multiset** is the same as an unordered set except that duplicates are allowed. Thus, an unordered multiset may contain multiple elements that have the same value.
- An **unordered map** contains elements that are key/value pairs. Each key may occur only once, so duplicate keys are not allowed. An unordered map can also be used as an *associative array*, an array that has an arbitrary index type (see Section 6.2.4, page 185, for details).
- An **unordered multimap** is the same as an unordered map except that duplicates are allowed. Thus, an unordered multimap may contain multiple elements that have the same key. An unordered multimap can also be used as *dictionary* (see Section 7.9.7, page 383, for an example).

All these unordered container classes have a couple of optional template arguments to specify a hash function and an equivalence criterion. The equivalence criterion is used to find specific values and to identify duplicates. The default equivalence criterion is the operator `==` .

You can consider an unordered set as a special kind of unordered map, in which the value is identical to the key. In fact, all these unordered container types are usually implemented by using the same basic implementation of a hash table.

Examples of Using Unordered Sets and Multisets

Here is a first example, using an unordered multiset of strings:

Click here to view code image

```
// stl/unordmultiset1.cpp

#include <unordered_set>
#include <string>
#include <iostream>
using namespace std;

int main()
{
    unordered_multiset<string> cities {
        "Braunschweig", "Hanover", "Frankfurt", "New York",
        "Chicago", "Toronto", "Paris", "Frankfurt"
    };

    //print each element:
    for (const auto& elem : cities) {
        cout << elem << "  ";
    }
    cout << endl;

    //insert additional values:
    cities.insert( {"London", "Munich", "Hanover", "Braunschweig"} );

    //print each element:
    for (const auto& elem : cities) {
        cout << elem << "  ";
    }
    cout << endl;
}
```

After including the required header file

```
#include <unordered_set>
```

we can declare and initialize an unordered set of strings:

```
unordered_multiset<string> cities { … };
```

Now, if we print all elements, the order might be different because the order is undefined. The only guarantee is that duplicates, which are possible because a *multi*set is used, are grouped together in the order of their insertion. Thus, a possible output might be:

```
Paris  Toronto  Chicago  New York  Frankfurt  Frankfurt  Hanover
  Braunschweig
```

Any insertion can change this order. In fact, any operation that causes rehashing can change this order. So, after inserting a couple more values, the output might be as follows:

```
London  Hanover  Hanover  Frankfurt  Frankfurt  New York  Chicago
  Munich  Braunschweig  Braunschweig  Toronto  Paris
```

What happens depends on the rehashing policy, which can be influenced in part by the programmer. For example, you can reserve enough room so that rehashing won't happen up to a specific number of elements. In addition, to ensure that you can delete elements while processing all elements, the standard guarantees that deleting elements does not cause a rehashing. But an insertion after a deletion might cause rehashing. For details, see Section 7.9, page 355.

In general, associative and unordered containers provide the same interface. Only declarations might differ, and unordered containers provide special member functions to influence the internal behavior or to inspect the current state. Thus, in the example presented here, only the header files and types differ from the corresponding example using an ordinary multiset, which was introduced in Section 6.2.2, page 177.

Again, before C++11, you needed iterators to access the elements. See Section 6.3.1, page 193, for an example.

Examples of Using Unordered Maps and Multimaps

The example presented for multimaps on page 179 also works for an unordered multimap if you replace `map` by `unordered_map` in the include directive and `multimap` by `unordered_multimap` in the declaration of the container:

```
#include <unordered_map>
…
unordered_multimap<int,string> coll;
…
```

The only difference is that the order of the elements is undefined. However, on most platforms, the elements will still be sorted because as a default hash function, the modulo operator is used. Thus, a sorted order is also a valid undefined order. However, that's not guaranteed, and if you add more elements, the order will be different.

Here is another example using an unordered map. In this case, we use an unordered map where the keys are strings and the values are doubles:

Click here to view code image

```
// stl/unordmap1.cpp

#include <unordered_map>
#include <string>
#include <iostream>
using namespace std;

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

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

    // print each element (key and value):
    for (const auto& elem : coll) {
        cout << elem.first << ": " << elem.second << endl;
    }
}
```

After the usual includes for maps, strings, and iostreams, an unordered map is declared and initialized by two elements. Here, we use nested initializer lists so that

```
{ "tim", 9.9 }
```

and

```
{ "struppi", 11.77 }
```

are the two elements used to initialize the map.

Next, we square the value of each element:

```
for (pair<const string,double>& elem : coll) {
    elem.second *= elem.second;
}
```

Here again, you can see the internal type of the elements, which is a `pair<>` (see Section 5.1.1, page 60) of constant strings and doubles. Thus, we could not modify the key member `first` in the element:

```
for (pair<const string,double>& elem : coll) {
    elem.first = …;      // ERROR: keys of a map are constant
}
```

As usual since C++11, we don't have to specify the type of the elements explicitly, because in a range-based `for` loop, it is deduced from the container. For this reason, the second loop, which outputs all elements, uses `auto`. In fact, by declaring `elem` as `const auto&`, we avoid having copies created:

```
for (const auto& elem : coll) {
    cout << elem.first << ": " << elem.second << endl;
}
```

As a result, one *possible* output of the program is as follows:

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

This order is not guaranteed, because the actual order is undefined. If we used an ordinary `map` instead, the order of the elements would be guaranteed to print the element with key `"struppi"` before the element with key `"tim"`, because the map sorts the elements according to the key, and the string `"struppi"` is less than `"tim"`. See Section 7.8.5, page 345, for a corresponding example using a map and also using algorithms and lambdas instead of range-based `for` loops.

Other Examples for Unordered Containers

The classes for unordered containers provide a couple of additional optional template arguments, such as the hash function and the equivalence comparison. A default hash function is provided for fundamental types and strings, but we would have to declare our own hash function for other types. This is discussed in Section 7.9.2, page 363.

The next section gives an example for using a map as a so-called *associative array*. Section 7.9 discusses unordered containers in detail, with additional examples. Unordered multimaps can also be used as *dictionaries* (see Section 7.9.7, page 383, for an example).

6.2.4. Associative Arrays

Both maps and unordered maps are collections of key/value pairs with unique keys. Such a collection can also be thought of as an *associative array*, an array whose index is not an integer value. As a consequence, both containers provide the subscript operator `[]`.

Consider the following example:

Click here to view code image

```
// stl/assoarray1.cpp

#include <unordered_map>
#include <string>
#include <iostream>
using namespace std;

int main()
{
    // type of the container:
    // - unordered_map: elements are key/value pairs
    // - string: keys have type string
    // - float: values have type float
    unordered_map<string,float> coll;

    // insert some elements into the collection
    // - using the syntax of an associative array
    coll["VAT1"] = 0.16;
    coll["VAT2"] = 0.07;
    coll["Pi"] = 3.1415;
    coll["an arbitrary number"] = 4983.223;
    coll["Null"] = 0;

    // change value
    coll["VAT1"] += 0.03;

    // print difference of VAT values
    cout << "VAT difference: " << coll["VAT1"] - coll["VAT2"] << endl;
}
```

The declaration of the container type must specify both the type of the key and the type of the value:

```
unordered_map<string,float> coll;
```

This means that the keys are strings and the associated values are floating-point values.

According to the concept of associative arrays, you can access elements by using the subscript operator `[]`. Note, however, that the subscript operator does not behave like the usual subscript operator for arrays: Not having an element for an index is *not* an error. A new index (or key) is taken as a reason to create and insert a new map element that has the index as the key. Thus, you can't have an invalid index.

Therefore, in this example, the statement

```
coll["VAT1"] = 0.16;
```

creates a new element, which has the key `"VAT1"` and the value `0.16`.

In fact, the following expression creates a new element that has the key `"VAT1"` and is initialized with its default value (using the default constructor or 0 for fundamental data types):

```
coll["VAT1"]
```

The whole expression yields access to the value of this new element, so the assignment operator assigns `0.16` then.

Since C++11, you can, alternatively, use `at()` to access values of elements while passing the key. In this case, a key not found results in an `out_of_range` exception:

```
coll.at("VAT1") = 0.16;     // out_of_range exception if no element found
```

With expressions such as

```
coll["VAT1"] += 0.03;
```

or

```
coll["VAT1"] - coll["VAT2"]
```

you gain read and write access to the value of these elements. Thus, the output of the program is as follows:

```
VAT difference: 0.12
```

As usual, the difference between using an unordered map and a map is that the elements in an unordered map have arbitrary order, whereas the elements in a map are sorted. But because the complexity for element access is amortized constant for unordered maps rather than logarithmic for maps, you should usually prefer unordered maps over maps unless you need the sorting or can't use an unordered map because your environment does not support features of C++11. In that case, you simply have to change the type of the container: remove the " `unordered_` " in both the include directive and the container declaration.

Section 7.8.3, page 343, and Section 7.9.5, page 374, discuss `map` s and `unordered_map` s as associative arrays in more detail.

6.2.5. Other Containers

Strings

You can also use strings as STL containers. By *strings*, I mean objects of the C++ string classes (`basic_string<>` , `string` , and `wstring`), which are introduced in Chapter 13. Strings are similar to vectors but have characters as elements. Section 13.2.14, page 684, provides details.

Ordinary C-Style Arrays

Another kind of container is a type of the core C and C++ language rather than a class: an ordinary array ("C-style array") that has a declared fixed size or a dynamic size managed by `malloc()` and `realloc()` . However, such ordinary arrays are not STL containers, because they don't provide member functions, such as `size()` and `empty()` . Nevertheless, the STL's design allows you to call algorithms for them.

The use of ordinary arrays is nothing new. What is new is using algorithms for them. This is explained in Section 7.10.2, page 386.

In C++, it is no longer necessary to program C-style arrays directly. Vectors and `array` s provide all properties of ordinary C-style arrays but with a safer and more convenient interface. See Section 7.2.3, page 267, and Section 7.3.3, page 278, for details.

User-Defined Containers

In principle, you can give any container-like object a corresponding STL interface to be able to iterate through elements or provide standard operations to manipulate its content. For example, you might introduce a class that represents a directory where you can iterate over the files as elements and manipulate them. The best candidates for STL-container-like interfaces are the common container operations introduced in Section 7.1, page 254.

However, some container-like objects do not fit into the concept of the STL. For example, the fact that STL containers have a begin and an end makes it hard for circular container types, such as a ring buffer, to fit into the STL framework.

6.2.6. Container Adapters

In addition to the fundamental container classes, the C++ standard library provides so-called *container adapters*, which are predefined containers that provide a restricted interface to meet special needs. These container adapters are implemented by using the fundamental container classes. The predefined container adapters are as follows:

- A **stack** (the name says it all) manages its elements by the LIFO (last-in-first-out) policy.
- A **queue** manages its elements by the FIFO (first-in-first-out) policy. That is, it is an ordinary buffer.
- A **priority queue** is a container in which the elements may have different priorities. The priority is based on a sorting criterion that the programmer may provide (by default, operator `<` is used). A priority queue is, in effect, a buffer in which the next element is always one having the highest priority inside the queue. If more than one element has the highest priority, the order of these elements is undefined.

Container adapters are historically part of the STL. However, from a programmer's viewpoint, they are just special container classes that use the general framework of the containers, iterators, and algorithms provided by the STL. Therefore, container adapters are described apart from the STL core in Chapter 12.