
6.3. Iterators

Since C++11, we can process all elements by using a range-based `for` loop. However, to find an element, we don't want to process all elements. Instead, we have to iterate over all elements until we find what we are searching for. In addition, we probably want to be able to store this position somewhere, for example, to continue with the iteration or some other processing later on. Thus, we need a concept of an object that represents positions of elements in a container. This concept exists. Objects that fulfill this concept are called **iterators**. In fact, as we will see, range-based `for` loops are a convenience interface to this concept. That is, they internally use iterator objects that iterate over all elements.

An iterator is an object that can iterate over elements (navigate from element to element). These elements may be all or a subset of the elements of an STL container. An iterator represents a certain position in a container. The following fundamental operations define the behavior of an iterator:

- **Operator** `*` returns the element of the current position. If the elements have members, you can use operator `->` to access those members directly from the iterator.
- **Operator** `++` lets the iterator step forward to the next element. Most iterators also allow stepping backward by using operator `--`.
- **Operators** `==` and `!=` return whether two iterators represent the same position.
- **Operator** `=` assigns an iterator (the position of the element to which it refers).

These operations provide exactly the interface of ordinary pointers in C and C++ when these pointers are used to iterate over the elements of an ordinary array. The difference is that iterators may be *smart pointers* — pointers that iterate over more complicated data structures of containers. The internal behavior of iterators depends on the data structure over which they iterate. Hence, each container type supplies its own kind of iterator. As a result, iterators share the same interface but have different types. This leads directly to the concept of generic programming: Operations use the same interface but different types, so you can use templates to formulate generic operations that work with arbitrary types that satisfy the interface.

All container classes provide the same basic member functions that enable them to use iterators to navigate over their elements. The most important of these functions are as follows:

- **begin()** returns an iterator that represents the beginning of the elements in the container. The beginning is the position of the first element, if any.
- **end()** returns an iterator that represents the end of the elements in the container. The end is the position *behind* the last element. Such an iterator is also called a *past-the-end iterator*.

Thus, `begin()` and `end()` define a *half-open range* that includes the first element but excludes the last (Figure 6.4). A half-open range has two advantages:

1. You have a simple end criterion for loops that iterate over the elements: They simply continue as long as `end()` is not reached.
2. It avoids special handling for empty ranges. For empty ranges, `begin()` is equal to `end()`.

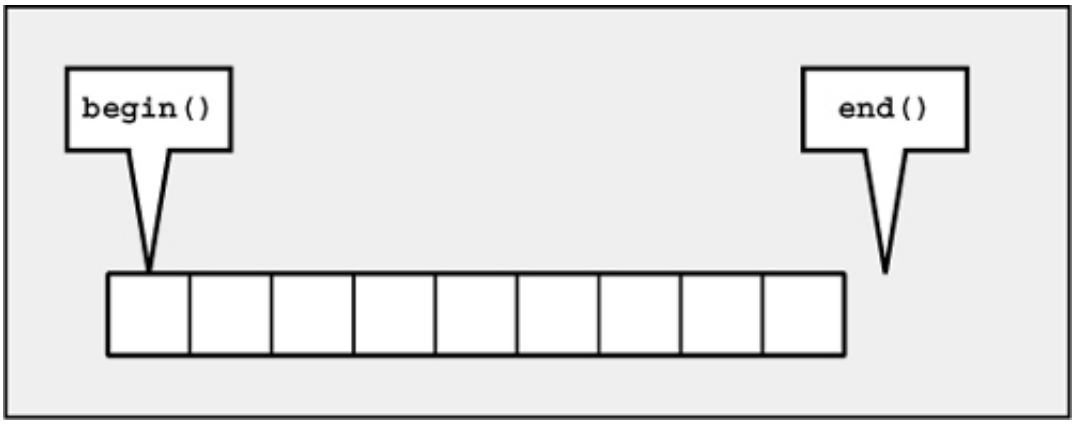

Figure 6.4. *`begin()`* **and** *`end()`* **for Containers**

The following example demonstrating the use of iterators prints all elements of a list container (it is the iterator-based version of the first list example in Section 6.2.1, page 173):

```
// stl/list1old.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:
    // - iterate over all elements
    list<char>::const_iterator pos;
    for (pos = coll.begin(); pos != coll.end(); ++pos) {
        cout << *pos << ' ';
    }
    cout << endl;
}
```

Again, after the list is created and filled with the characters `'a'` through `'z'`, we print all elements. But instead of using a range-based `for` loop:

```
for (auto elem : coll) {
    cout << elem << ' ';
}
```

all elements are printed within an ordinary `for` loop using an iterator iterating over all elements of the container:

```
list<char>::const_iterator pos;
for (pos = coll.begin(); pos != coll.end(); ++pos) {
    cout << *pos << ' ';
}
```

The iterator `pos` is declared just before the loop. Its type is the iterator type for constant element access of its container class:

```
list<char>::const_iterator pos;
```

In fact, every container defines two iterator types:

1. *container* `::iterator` is provided to iterate over elements in read/write mode.
2. *container* `::const_iterator` is provided to iterate over elements in read-only mode.

For example, in class `list`, the definitions might look like the following:

```
namespace std {
    template <typename T>
    class list {
      public:
        typedef ... iterator;
        typedef ... const_iterator;
        ...
    };
}
```

The exact type of `iterator` and `const_iterator` is implementation defined.

Inside the `for` loop, the iterator `pos` first gets initialized with the position of the first element:

```
pos = coll.begin()
```

The loop continues as long as `pos` has not reached the end of the container elements:

```
pos != coll.end()
```

Here, `pos` is compared with a so-called past-the-end iterator, which represents the position right behind the last element. While the loop runs the increment operator, `++pos` navigates the iterator `pos` to the next element.

All in all, `pos` iterates from the first element, element-by-element, until it reaches the end (Figure 6.5). If the container has no elements, the body of the loop is never executed, because `coll.begin()` equals `coll.end()`, then.

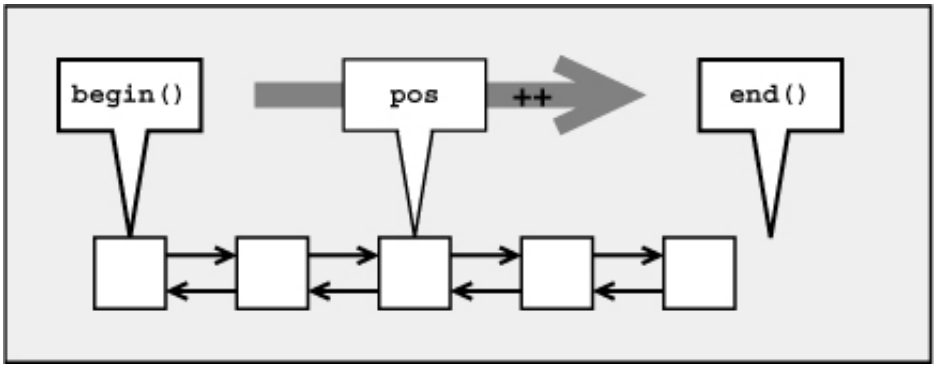

Figure 6.5. Iterator *pos* Iterating over Elements of a List

In the body of the loop, the expression `*pos` represents the current element. In this example, it is written to the standard output `cout`, followed by a space character. You can't modify the elements, because a `const_iterator` is used. Thus, from the iterator's point of view, the elements are constant. However, if you use a nonconstant iterator and the type of the elements is nonconstant, you can change the values. For example:

```
// make all characters in the list uppercase
list<char>::iterator pos;
for (pos = coll.begin(); pos != coll.end(); ++pos) {
    *pos = toupper(*pos);
}
```

If we use iterators to iterate over the elements of (unordered) maps and multimaps, `pos` would again refer to key/value pairs. Thus, the expression

```
pos->second
```

would yield the second part of the key/value pair, which is the value of the element, whereas

```
pos->first
```

would yield its (constant) key.

++pos versus pos++

Note that the preincrement operator (prefix `++`) is used here to move the iterator to the next element. The reason is that it might have better performance than the postincrement operator. The latter internally involves a temporary object because it must return the old position of the iterator. For this reason, it generally is better to prefer `++pos` over `pos++`. Thus, you should avoid the following version:

Click here to view code image

```
for (pos = coll.begin(); pos != coll.end(); pos++) {
                                            ^^^^^   // OK, but slightly
slower
    ...
}
```

These kinds of performance improvements almost always don't matter. So, don't interpret this recommendation to mean that you should do everything to avoid even the smallest performance penalties. Readable and maintainable programs are far more important than optimized performance. The important point here is that in this case, you don't pay a price for preferring the preincrement over the postincrement version. For this reason, it is a good advice to prefer the preincrement and predecrement operators in general.

cbegin() and cend()

Since C++11, we can use the keyword `auto` (see Section 3.1.2, page 14) to specify the exact type of the iterator (provided that you initialize the iterator during its declaration so that its type can be derived from the initial value). Thus, by initializing the iterator directly with `begin()`, you can use `auto` to declare its type:

Click here to view code image

```
for (auto pos = coll.begin(); pos != coll.end(); ++pos) {
    cout << *pos << ' ';
}
```

As you can see, one advantage of using `auto` is that the code is more condensed. Without `auto`, declaring the iterator inside the loop looks as follows:

```
for (list<char>::const_iterator pos = coll.begin();
     pos != coll.end();
     ++pos) {
    cout << *pos << ' ';
```

```
}
```

The other advantage is that the loop is robust for such code modifications as changing the type of the container. However, the drawback is that the iterator loses its `const` ness, which might raise the risk of unintended assignments. With

```
auto pos = coll.begin()
```

`pos` becomes a nonconstant iterator because `begin()` returns an object of type *cont* `::iterator`. To ensure that constant iterators are still used, `cbegin()` and `cend()` are provided as container functions since C++11. They return an object of type *cont* `::const_iterator`.

To summarize the improvements, since C++11, a loop that allows iterating over all the elements of a container without using a range-based `for` loop might look as follows:

Click here to view code image

```
for (auto pos = coll.cbegin(); pos != coll.cend(); ++pos) {
    ...
}
```

Range-Based `for` Loops versus Iterators

Having introduced iterators, we can explain the exact behavior of range-based `for` loops. For containers, in fact, a range-based `for` loop is nothing but a convenience interface, which is defined to iterate over all elements of the passed range/collection. Within each loop body, the actual element is initialized by the value the current iterator refers to.

Thus,

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

is interpreted as

Click here to view code image

```
for (auto pos=coll.begin(), end=coll.end(); pos!=end; ++pos) {
    type elem = *pos;
    ...
}
```

Now we can understand why we should declare `elem` to be a constant reference to avoid unnecessary copies. Otherwise, `elem` will be initialized as a copy of `*pos`. See Section 3.1.4, page 17, for details.

6.3.1. Further Examples of Using Associative and Unordered Containers

Having introduced iterators, we can present some example programs using associative containers without using such language features of C++11 as range-based `for` loops, `auto`, and initializer lists. In addition, the features used here can also be useful with C++11 for some special requirements.

Using Sets before C++11

The first example shows how to insert elements into a set and use iterators to print them if C++11 features are not available:

```
// stl/set1.cpp

#include <set>
#include <iostream>

int main()
{
    // type of the collection
    typedef std::set<int> IntSet;

    IntSet coll;          // set container for int values
    // insert elements from 1 to 6 in arbitrary order
    // - note that there are two calls of insert() with value 1
    coll.insert(3);
    coll.insert(1);
    coll.insert(5);
    coll.insert(4);
    coll.insert(1);
    coll.insert(6);
    coll.insert(2);
```

```
    // print all elements
    // - iterate over all elements
    IntSet::const_iterator pos;
    for (pos = coll.begin(); pos != coll.end(); ++pos) {
        std::cout << *pos << ' ';
    }
    std::cout << std::endl;
}
```

As usual, the `include` directive defines all necessary types and operations of sets:

```
#include <set>
```

The type of the container is used in several places, so first, a shorter type name gets defined:

```
typedef set<int> IntSet;
```

This statement defines type `IntSet` as a set for elements of type `int`. This type uses the default sorting criterion, which sorts the elements by using operator `<`, so the elements are sorted in ascending order. To sort in descending order or use a completely different sorting criterion, you can pass it as a second template parameter. For example, the following statement defines a set type that sorts the elements in descending order:

```
typedef set<int,greater<int>> IntSet;
```

`greater<>` is a predefined function object discussed in Section 6.10.2, page 239. For a sorting criterion that uses only part of the data of an object, such as the ID, see Section 10.1.1, page 476. All associative containers provide an `insert()` member function to insert a new element:

```
coll.insert(3);
coll.insert(1);
...
```

Since C++11, we can simply call:

```
coll.insert ( { 3, 1, 5, 4, 1, 6, 2 } );
```

Each inserted element receives the correct position automatically according to the sorting criterion. You can't use the `push_back()` or `push_front()` functions provided for sequence containers. They make no sense here, because you can't specify the position of the new element.

After all values are inserted in any order, the state of the container is as shown in Figure 6.6. The elements are sorted into the internal tree structure of the container, so the value of the left child of an element is always less, with respect to the current sorting criterion, and the value of the right child of an element is always greater. Duplicates are not allowed in a set, so the container contains the value 1 only once.

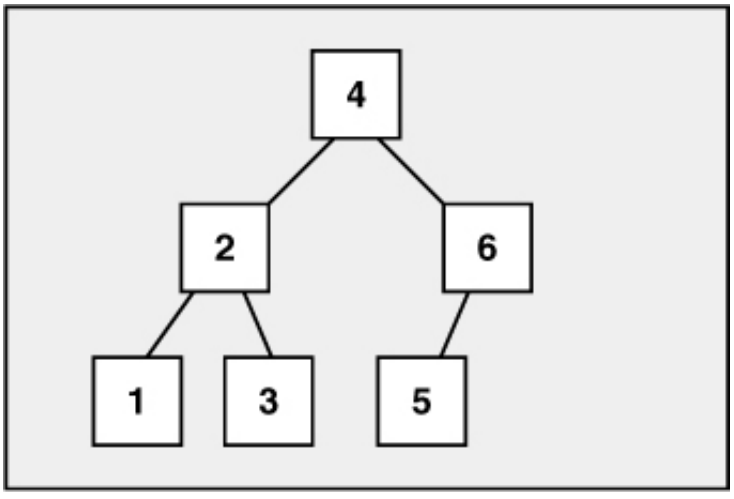

Figure 6.6. A Set of Six Elements

To print the elements of the container, you use the same loop as in the previous list example. An iterator iterates over all elements and prints them:

```
IntSet::const_iterator pos;
for (pos = coll.begin(); pos != coll.end(); ++pos) {
    cout << *pos << ' ';
}
```

Because it is defined by the container, the iterator does the right thing, even if the internal structure of the container is more complicated. For example, if the iterator refers to the third element, operator `++` moves to the fourth element at the top. After the next call of operator

`++`, the iterator refers to the fifth element at the bottom (Figure 6.7). The output of the program is as follows:

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

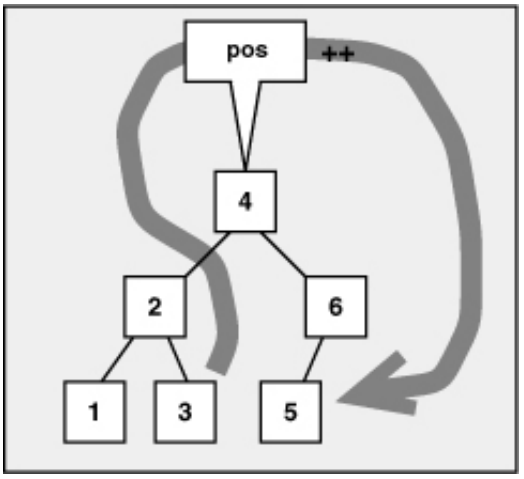

Figure 6.7. Iterator *pos* Iterating over Elements of a Set

To use a multiset rather than a set, you need only change the type of the container; the header file remains the same:

```
typedef multiset<int> IntSet;
```

A multiset allows duplicates, so it would contain two elements that have value `1`. Thus, the output of the program would change to the following:

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

Details of Using an Unordered Multiset

As another example, let's look in detail at what happens when we iterate over all elements of an unordered multiset. Consider the following example:

```
// stl/unordmultiset2.cpp

#include <unordered_set>
#include <iostream>

int main()
{
    // unordered multiset container for int values
    std::unordered_multiset<int> coll;

    // insert some elements
    coll.insert({1,3,5,7,11,13,17,19,23,27,1});

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

    // insert one more element
    coll.insert(25);

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

The order of the elements is undefined. It depends on the internal layout of the hash table and its hashing function. Even when the elements are inserted in sorted order, they have an arbitrary order inside the container.[8] Adding one more element might change the order of all existing elements. Thus, one of the possible outputs of this program is as follows:

[8] Note that the order might also be sorted because this is one possible arbitrary order. In fact, if you insert `1`, `2`, `3`, `4`, and `5`, this will typically be the case.

```
11 23 1 1 13 3 27 5 17 7 19
23 1 1 25 3 27 5 7 11 13 17 19
```

As you can see, the order is indeed undefined, but might differ if you run this example on your platform. Adding just one element might

change the whole order. However, it is guaranteed that elements with equal values are adjacent to each other.

When iterating over all elements to print them:

```
for (auto elem : coll) {
    std::cout << elem << ' ';
}
```

this is equivalent to the following:

```
for (auto pos = coll.begin(); pos != coll.end(); ++pos) {
    auto elem = *pos;
    std::cout << elem << ' ';
}
```

Again, the internal iterator `pos` internally used in the `for` loop has a type provided by the container so that it "knows" how to iterate over all the elements. So, with this output, the internal state of the unordered multiset might look like Figure 6.8 when the iterator is used to print all the elements the first time.

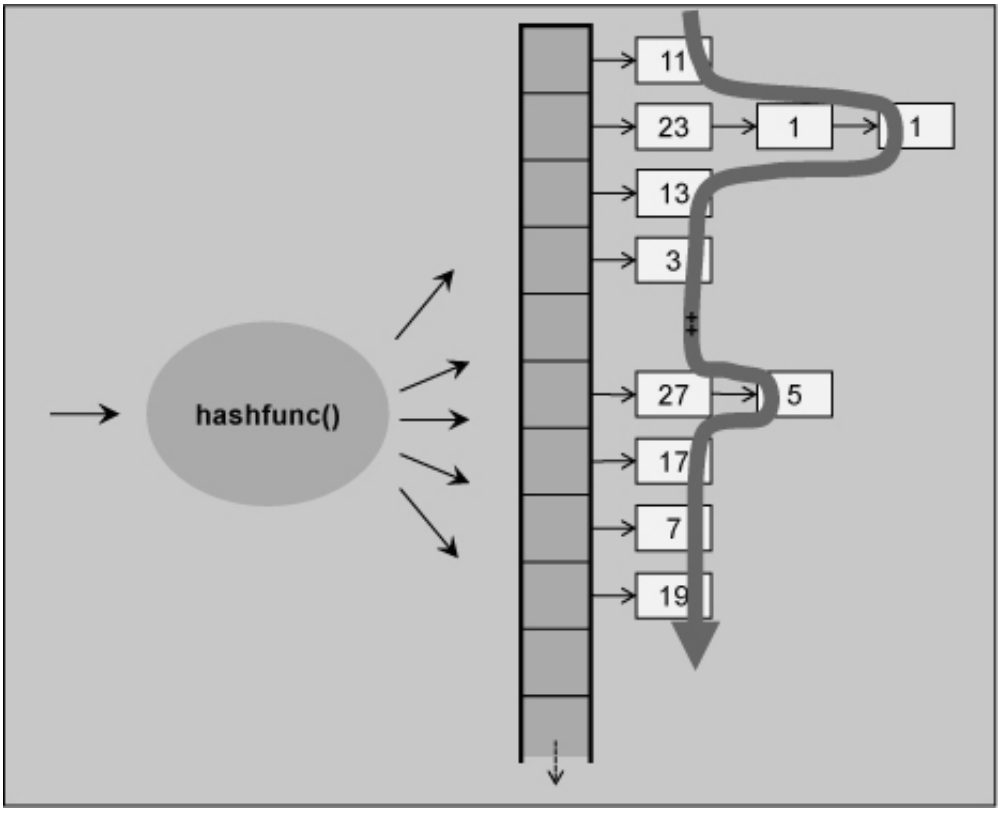

Figure 6.8. Iterator *pos* Iterating over Elements of an Unordered Multiset

When not allowing duplicates by switching to an unordered set:

```
std::unordered_set<int> coll;
```

the output might look as follows:

```
11 23 1 13 3 27 5 17 7 19
11 23 1 13 25 3 27 5 17 7 19
```

6.3.2. Iterator Categories

In addition to their fundamental operations, iterators can have capabilities that depend on the internal structure of the container type. As usual, the STL provides only those operations that have good performance. For example, if containers have random access, such as vectors or deques, their iterators are also able to perform random-access operations, such as positioning the iterator directly at the fifth element.

Iterators are subdivided into *categories* based on their general abilities. The iterators of the predefined container classes belong to one of the following three categories:

1. **Forward iterators** are able to iterate only forward, using the increment operator. The iterators of the class `forward_list` are forward iterators. The iterators of the container classes `unordered_set`, `unordered_multiset`, `unordered_map`, and `unordered_multimap` are "at least" forward iterators (libraries are allowed to provide bidirectional iterators instead, see Section 7.9.1, page 357).
2. **Bidirectional iterators** are able to iterate in two directions: forward, by using the increment operator, and backward, by using the decrement operator. The iterators of the container classes `list`, `set`, `multiset`, `map`, and `multimap` are bidirectional iterators.

3. **Random-access iterators** have all the properties of bidirectional iterators. In addition, they can perform random access. In particular, they provide operators for *iterator arithmetic* (corresponding to *pointer arithmetic* of an ordinary pointer). You can add and subtract offsets, process differences, and compare iterators by using relational operators, such as `<` and `>`. The iterators of the container classes `vector`, `deque`, `array`, and iterators of strings are random-access iterators.

In addition, two other iterator categories are defined:

- **Input iterators** are able to read/process some values while iterating forward. Input stream iterators are an example of such iterators (see Section 6.5.2, page 212).
- **Output iterators** are able to write some values while iterating forward. Inserters (Section 6.5.1, page 210) and output stream iterators (see Section 6.5.2, page 212) are examples of such iterators.

Section 9.2, page 433, discusses all iterator categories in detail.

To write generic code that is as independent of the container type as possible, you should not use special operations for random-access iterators. For example, the following loop works with any container:

Click here to view code image

```
for (auto pos = coll.begin(); pos != coll.end(); ++pos) {
    ...
}
```

However, the following does *not* work with all containers:

Click here to view code image

```
for (auto pos = coll.begin(); pos < coll.end(); ++pos) {
    ...
}
```

The only difference is the use of operator `<` instead of operator `!=` in the condition of the loop. Operator `<` is provided only for random-access iterators, so this loop does not work with lists, sets, and maps. To write generic code for arbitrary containers, you should use operator `!=` rather than operator `<`. However, doing so might lead to code that is less safe. The reason is that you may not recognize that `pos` gets a position behind `end()` (see Section 6.12, page 245, for more details about possible errors when using the STL). It's up to you to decide which version to use. It might be a question of the context or even of taste.

To avoid misunderstanding, note that I am talking about "categories," *not* "classes." A category defines only the abilities of iterators. The type doesn't matter. The generic concept of the STL works with *pure abstraction*: anything that *behaves* like a bidirectional iterator *is* a bidirectional iterator.