Username: Pralay Patoria **Book:** The C++ Standard Library: A Tutorial and Reference, Second Edition. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

9.2. Iterator Categories

Iterators are objects that can iterate over elements of a sequence via a common interface that is adapted from ordinary pointers (see Section 6.3, page 188). Iterators follow the concept of pure abstraction: Anything that *behaves* like an iterator *is* an iterator. However, iterators have different abilities. These abilities are important because some algorithms require special iterator abilities. For example, sorting algorithms require iterators that can perform random access because otherwise, the performance would be poor. For this reason, iterators have different categories (Figure 9.1). The abilities of these categories are listed in Table 9.1 and discussed in the following subsections.

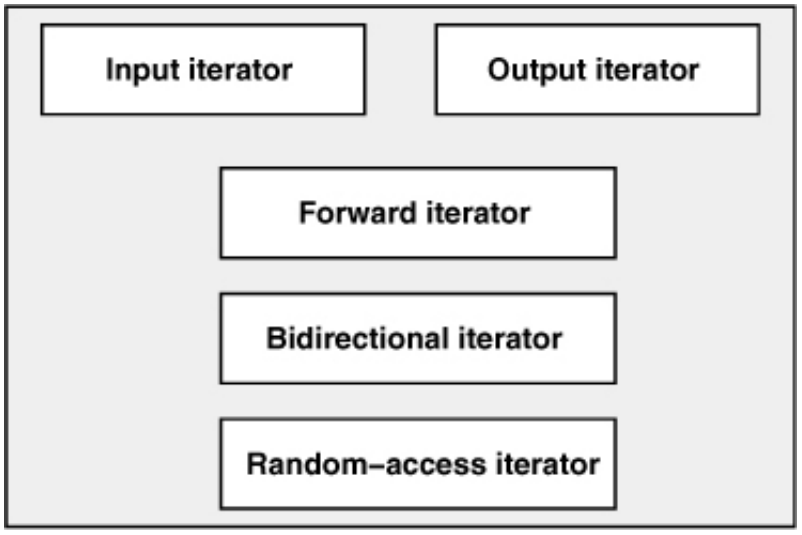

Figure 9.1. Iterator Categories

Table 9.1. Abilities of Iterator Categories

Iterator Category	Ability	Providers
Output iterator	Writes forward	Ostream, inserter
Input iterator	Reads forward once	Istream
Forward iterator	Reads forward	Forward list, unordered containers
Bidirectional iterator	Reads forward and backward	List, set, multiset, map, multimap
Random-access iterator	Reads with random access	Array, vector, deque, string, C-style array

Reading iterators that can also write are called *mutable* iterators (for example, *mutable forward iterator*).

9.2.1. Output Iterators

Output iterators can only step forward with write access. Thus, you can assign new values only element-by-element. You can't use an output iterator to iterate twice over the same range. In fact, it is even not guaranteed that you can assign a value twice without incrementing the iterator. The goal is to write a value into a "black hole" in the following way:

```
while (...) {
    *pos = ...;        // assign a value
    ++pos;             // advance (prepare for the next assignment)
}
```

Table 9.2 lists the valid operations for output iterators. The only valid use of operator * is on the left side of an assignment statement.

Table 9.2. Operations of Output Iterators

Expression	Effect
**iter = val*	Writes *val* to where the iterator refers
++iter	Steps forward (returns new position)
iter++	Steps forward (returns old position)
TYPE(iter)	Copies iterator (copy constructor)

No comparison operations are required for output iterators. You can't check whether an output iterator is valid or whether a "writing" was successful. You can only write, and write, and write values. Usually, the end of a writing is defined by an additional requirement for specific output iterators.

Often, iterators can read and write values. For this reason, all reading iterators might have the additional ability to write. In that case, they are called *mutable* iterators.

A typical example of a pure output iterator is one that writes to the standard output (for example, to the screen or a printer). If you use two output iterators to write to the screen, the second word follows the first rather than overwriting it. Inserters are another typical example of output iterators. Inserters are iterators that insert values into containers. If you assign a value, you insert it. If you then write a second value, you don't overwrite the first value; you just also insert it. Inserters are discussed in Section 9.4.2, page 454.

All `const_iterator`s provided by containers and their member functions `cbegin()` and `cend()` are not output iterators, because they don't allow you to write to where the iterator refers.

9.2.2. Input Iterators

Input iterators can only step forward element-by-element with read access. Thus, they return values element-wise. Table 9.3 lists the operations of input iterators.

Table 9.3. Operations of Input Iterators

Expression	Effect
**iter*	Provides read access to the actual element
iter ->member	Provides read access to a member of the actual element
++iter	Steps forward (returns new position)
iter++	Steps forward
iter1 == iter2	Returns whether two iterators are equal
iter1 != iter2	Returns whether two iterators are not equal
TYPE(iter)	Copies iterator (copy constructor)

Input iterators can read elements only once. Thus, if you copy an input iterator and let the original and the copy read forward, they might iterate over different values.

All iterators that refer to values to process have the abilities of input iterators. Usually, however, they can have more. A typical example of a pure input iterator is one that reads from the standard input, which is typically the keyboard. The same value can't be read twice. Once a word is read from an input stream, the next read access returns another word.

For input iterators, operators `==` and `!=` are provided only to check whether an iterator is equal to a *past-the-end iterator*. This is required because operations that deal with input iterators usually do the following:

```
InputIterator pos, end;

while (pos != end) {
    ... // read-only access using *pos
    ++pos;
}
```

There is no guarantee that two different iterators that are both not *past-the-end iterators* compare unequal if they refer to different positions. (This requirement is introduced with forward iterators.)

Note also that for input iterators it is not required that the postincrement operator *iter* `++` returns something. Usually, however, it returns the old position.

You should always prefer the preincrement operator over the postincrement operator because it might perform better. This is because the preincrement operator does not have to return an old value that must be stored in a temporary object. So, for any iterator `pos` (and any abstract data type), you should prefer

```
++pos      // OK and fast
```

rather than

```
pos++      // OK, but not so fast
```

The same applies to decrement operators, as long as they are defined (they aren't for input iterators).

9.2.3. Forward Iterators

Forward iterators are input iterators that provide additional guarantees while reading forward. Table 9.4 summarizes the operations of forward iterators.

Table 9.4. Operations of Forward Iterators

Expression	Effect
**iter*	Provides access to the actual element
iter->member	Provides access to a member of the actual element
++iter	Steps forward (returns new position)
iter++	Steps forward (returns old position)
iter1 == iter2	Returns whether two iterators are equal
iter1 != iter2	Returns whether two iterators are not equal
TYPE()	Creates iterator (default constructor)
TYPE(iter)	Copies iterator (copy constructor)
iter1 = iter2	Assigns an iterator

Unlike for input iterators, it is guaranteed that for two forward iterators that refer to the same element, operator `==` yields `true` and that they will refer to the same value after both are incremented. For example:

```
ForwardIterator pos1, pos2;

pos1 = pos2 = begin;      // both iterators refer to the same element
if (pos1 != end) {
    ++pos1;                    // pos1 is one element ahead
    while (pos1 != end) {
        if (*pos1 == *pos2) {
            ... // process adjacent duplicates
            ++pos1;
            ++pos2;
    }
}
```

Forward iterators are provided by the following objects and types:

- Class `forward_list<>`
- Unordered containers

However, for unordered containers, libraries are allowed to provide bidirectional iterators instead (see Section 7.9.1, page 357).

A forward iterator that fulfills the requirements of an output iterator is a *mutable* forward iterator, which can be used for both reading and writing.

9.2.4. Bidirectional Iterators

Bidirectional iterators are forward iterators that provide the additional ability to iterate backward over the elements. Thus, they provide the decrement operator to step backward (Table 9.5).

Table 9.5. Additional Operations of Bidirectional Iterators

Expression	Effect
--iter	Steps backward (returns new position)
iter--	Steps backward (returns old position)

Bidirectional iterators are provided by the following objects and types:

- Class `list<>`
- Associative containers

A bidirectional iterator that fulfills the requirements of an output iterator is a *mutable* bidirectional iterator, which can be used for both reading and writing.

9.2.5. Random-Access Iterators

Random-access iterators provide all the abilities of bidirectional iterators plus random access. Thus, they provide operators for *iterator arithmetic* (in accordance with the *pointer arithmetic* of ordinary pointers). That is, they can add and subtract offsets, process differences, and compare iterators with relational operators, such as `<` and `>`. Table 9.6 lists the additional operations of random-access iterators.

Table 9.6. Additional Operations of Random-Access Iterators

Expression	Effect
iter[*n*]	Provides access to the element that has index *n*
iter+=*n*	Steps *n* elements forward (or backward, if *n* is negative)
iter−=*n*	Steps *n* elements backward (or forward, if *n* is negative)
iter+*n*	Returns the iterator of the *n*th next element
n+*iter*	Returns the iterator of the *n*th next element
iter−*n*	Returns the iterator of the *n*th previous element
iter1−*iter2*	Returns the distance between *iter1* and *iter2*
iter1<*iter2*	Returns whether *iter1* is before *iter2*
iter1>*iter2*	Returns whether *iter1* is after *iter2*
iter1<=*iter2*	Returns whether *iter1* is not after *iter2*
iter1>=*iter2*	Returns whether *iter1* is not before *iter2*

Random-access iterators are provided by the following objects and types:

- Containers with random access (`array` , `vector` , `deque`)
- Strings (`string` , `wstring`)
- Ordinary C-style arrays (pointers)

The following program shows the special abilities of random-access iterators:

Click here to view code image

```
// iter/itercategory1.cpp

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

int main()
{
   vector<int> coll;

   // insert elements from -3 to 9
   for (int i=-3; i<=9; ++i) {
       coll.push_back (i);
   }

   // print number of elements by processing the distance between beginning and end
   // - NOTE: uses operator - for iterators
   cout << "number/distance: " << coll.end()-coll.begin() << endl;

   // print all elements
   // - NOTE: uses operator < instead of operator !=
   vector<int>::iterator pos;
   for (pos=coll.begin(); pos<coll.end(); ++pos) {
       cout << *pos << ' ';
   }
   cout << endl;

   // print all elements
   // - NOTE: uses operator [] instead of operator *
   for (int i=0; i<coll.size(); ++i) {
       cout << coll.begin()[i] << ' ';
   }
   cout << endl;

   // print every second element
   // - NOTE: uses operator +=
   for (pos = coll.begin(); pos < coll.end()-1; pos += 2) {
       cout << *pos << ' ';
   }
   cout << endl;
}
```

The output of the program is as follows:

```
number/distance: 13
```

```
-3 -2 -1 0 1 2 3 4 5 6 7 8 9
-3 -2 -1 0 1 2 3 4 5 6 7 8 9
-3 -1 1 3 5 7
```

This example won't work with (forward) lists, (unordered) sets, and (unordered) maps, because all operations marked with "*NOTE:*" are provided only for random-access iterators. In particular, keep in mind that you can use operator `<` as an end criterion in loops for random-access iterators only.

Note that in the last loop the following expression requires that `coll` contains at least one element:

```
pos < coll.end()-1
```

If the collection was empty, `coll.end()-1` would be the position before `coll.begin()`. The comparison might still work, but, strictly speaking, moving an iterator to before the beginning results in undefined behavior. Similarly, the expression `pos += 2` might result in undefined behavior if it moves the iterator beyond the `end()` of the collection. Therefore, changing the final loop to the following is very dangerous because it results in undefined behavior if the collection contains an odd number of elements (Figure 9.2):

```
for (pos = coll.begin(); pos < coll.end(); pos += 2) {
    cout << *pos << ' ';
}
```

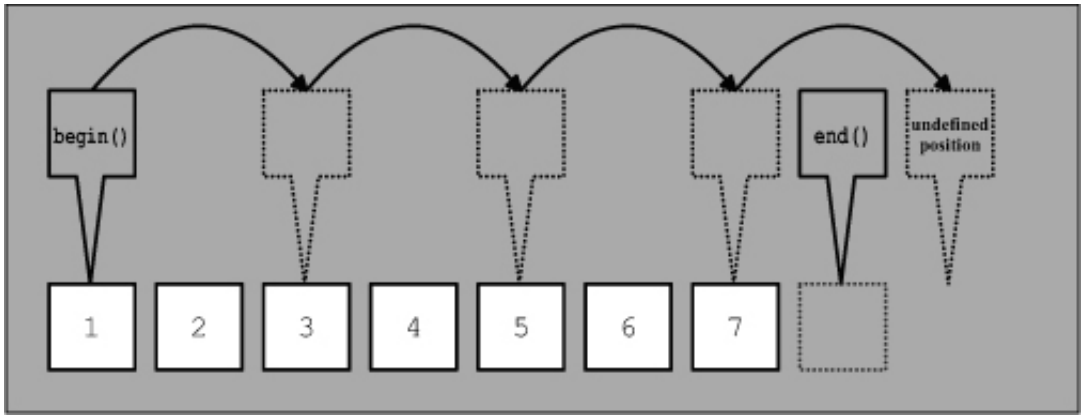

Figure 9.2. Incrementing Iterators by More than One Element

A random-access iterator that fulfills the requirements of an output iterator is a *mutable* random-access iterator, which can be used for both reading and writing.

9.2.6. The Increment and Decrement Problem of Vector Iterators

The use of the increment and decrement operators of iterators includes a strange problem. In general, you can increment and decrement temporary iterators. However, for `array`s, vectors, and strings, this might not compile on some platforms. Consider the following example:

```
std::vector<int> coll;
...
// sort, starting with the second element
// - NONPORTABLE version
if (coll.size() > 1) {
    std::sort(++coll.begin(), coll.end());
}
```

Depending on the platform, the compilation of `++coll.begin()` might fail. However, if you use, for example, a deque rather than a vector, the compilation always succeeds.

The reason for this strange problem lies in the fact that iterators of vectors, `array`s, and strings might be implemented as ordinary pointers. And for all fundamental data types, such as pointers, you are not allowed to modify temporary values. For structures and classes, however, doing so is allowed. Thus, if the iterator is implemented as an ordinary pointer, the compilation fails; if implemented as a class, it succeeds.

As a consequence, the preceding code always works with all containers except `array`s, vectors, and strings, because you can't implement iterators as ordinary pointers for them. But for `array`s, vectors, and strings, whether the code works depends on the implementation. Often, ordinary pointers are used. But if, for example, you use a "safe version" of the STL (as is more and more the case), the iterators are implemented as classes.

To make your code portable, the utility function `next()` is provided since C++11 (see Section 9.3.2, page 443), so you can write:

```
std::vector<int> coll;
...
// sort, starting with the second element
// - PORTABLE version since C++11
```

```
if (coll.size() > 1) {
    std::sort(std::next(coll.begin()), coll.end());
}
```

Before C++11, you had to use an auxiliary object instead:

```
std::vector<int> coll;
...
// sort, starting with the second element
// - PORTABLE version before C++11
if (coll.size() > 1) {
    std::vector<int>::iterator beg = coll.begin();
    std::sort(++beg, coll.end());
}
```

The problem is not as bad as it sounds. You can't get unexpected behavior, because the problem is detected at compile time. But it is tricky enough to spend time solving it.