
9.4. Iterator Adapters

This section covers iterator adapters provided by the C++ standard library. These special iterators allow algorithms to operate in reverse, in insert mode, and with streams.

9.4.1. Reverse Iterators

Reverse iterators redefine increment and decrement operators so that they behave in reverse. Thus, if you use these iterators instead of ordinary iterators, algorithms process elements in reverse order. Most container classes — all except forward lists and unordered containers — as well as strings provide the ability to use reverse iterators to iterate over their elements. Consider the following example:

```
// iter/reviter1.cpp

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

void print (int elem)
{
    cout << elem << ' ';
}

int main()
{
    // create list with elements from 1 to 9
    list<int> coll = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

    // print all elements in normal order
    for_each (coll.begin(), coll.end(),      // range
              print);                        // operation
    cout << endl;

    // print all elements in reverse order
    for_each (coll.rbegin(), coll.rend(),    // range
              print);                        // operations
    cout << endl;
}
```

The `rbegin()` and `rend()` member functions return a reverse iterator. As with `begin()` and `end()`, these iterators define the elements to process as a half-open range. However, they operate in a reverse direction:

- **`rbegin()`** returns the position of the first element of a reverse iteration. Thus, it returns the position of the last element.
- **`rend()`** returns the position after the last element of a reverse iteration. Thus, it returns the position *before* the first element.

Thus, the output of the program is as follows:

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

Since C++11, corresponding `crbegin()` and `crend()` member functions are provided, which return read-only reverse iterators. Because we only read the elements, they could (and should) be used in this example:

```
// print all elements in reverse order
for_each (coll.crbegin(), coll.crend(),    // range
          print);                          // operations
```

Iterators and Reverse Iterators

You can convert normal iterators into reverse iterators. Naturally, the iterators must be bidirectional iterators, but note that the logical position of an iterator is moved during the conversion. Consider the following program:

```
// iter/reviter2.cpp

#include <iterator>
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
```

```
int main()
{
    // create list with elements from 1 to 9
    vector<int> coll = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

    // find position of element with value 5
    vector<int>::const_iterator pos;
    pos = find (coll.cbegin(), coll.cend(),
                5);

    // print value to which iterator pos refers
    cout << "pos:   " << *pos << endl;

    // convert iterator to reverse iterator rpos
    vector<int>::const_reverse_iterator rpos(pos);

    // print value to which reverse iterator rpos refers
    cout << "rpos: " << *rpos << endl;
}
```

This program has the following output:

```
pos:  5
rpos: 4
```

Thus, if you print the value of an iterator and convert the iterator into a reverse iterator, the value has changed. This is not a bug; it's a feature! This behavior is a consequence of the fact that ranges are half open. To specify all elements of a container, you must use the position after the last argument. However, for a reverse iterator, this is the position before the first element. Unfortunately, such a position may not exist. Containers are not required to guarantee that the position before their first element is valid. Consider that ordinary strings and arrays might also be containers, and the language does not guarantee that arrays don't start at address zero.

As a result, the designers of reverse iterators use a trick: They "physically" reverse the "half-open principle." Physically, in a range defined by reverse iterators, the beginning is *not* included, whereas the end *is*. However, logically, they behave as usual. Thus, there is a distinction between the physical position that defines the element to which the iterator refers and the logical position that defines the value to which the iterator refers (Figure 9.3). The question is, what happens on a conversion from an iterator to a reverse iterator? Does the iterator keep its logical position (the value) or its physical position (the element)? As the previous example shows, the latter is the case. Thus, the value is moved to the previous element (Figure 9.4).

Figure 9.3. Position and Value of Reverse Iterators

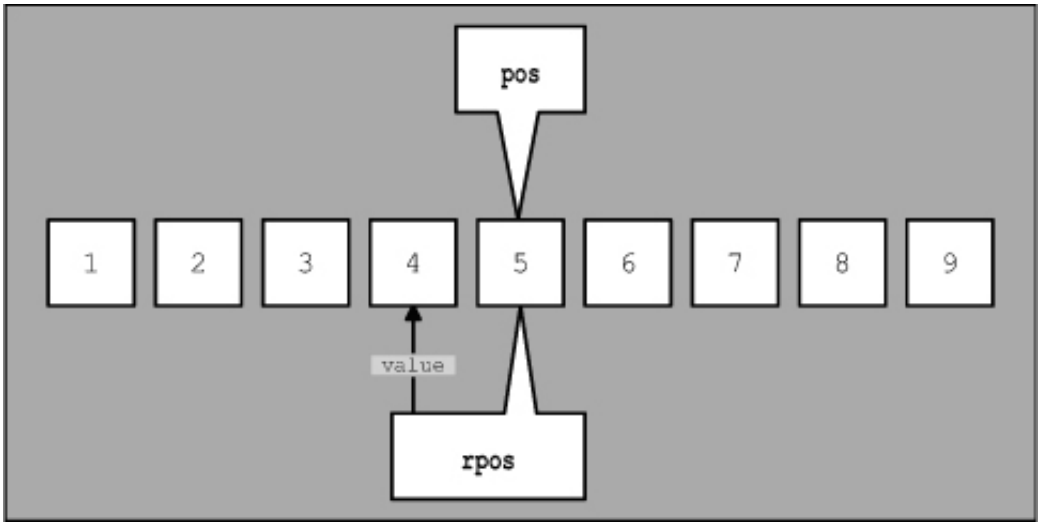

Figure 9.4. Conversion between Iterator `pos` and Reverse Iterator rpos

You can't understand this decision? Well, it has its advantages: You have nothing to do when you convert a range that is specified by two iterators rather than a single iterator. All elements remain valid. Consider the following example:

```
// iter/reviter3.cpp

#include <iterator>
#include <iostream>
#include <deque>

#include <algorithm>
using namespace std;

void print (int elem)
{
     cout << elem << ' ';
}

int main()
{
    // create deque with elements from 1 to 9
    deque<int> coll = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

    // find position of element with value 2
    deque<int>::const_iterator pos1;
    pos1 = find (coll.cbegin(), coll.cend(),     // range
                 2);                             // value

    // find position of element with value 7
    deque<int>::const_iterator pos2;
    pos2 = find (coll.cbegin(), coll.cend(),     // range
                 7);                             // value

    // print all elements in range [pos1,pos2)
    for_each (pos1, pos2,        // range
              print);            // operation
    cout << endl;

    // convert iterators to reverse iterators
    deque<int>::const_reverse_iterator rpos1(pos1);
    deque<int>::const_reverse_iterator rpos2(pos2);

    // print all elements in range [pos1,pos2) in reverse order
    for_each (rpos2, rpos1,      // range
              print);            // operation
    cout << endl;
}
```

The iterators `pos1` and `pos2` specify the half-open range, including the element with value `2` but excluding the element with value `7`. When the iterators describing that range are converted into reverse iterators, the range remains valid and can be processed in reverse order. Thus, the output of the program is as follows:

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

Thus, `rbegin()` is simply:

```
container::reverse_iterator(end())
```

and `rend()` is simply:

```
container::reverse_iterator(begin())
```

Of course, constant iterators are converted into type `const_reverse_iterator`.

Converting Reverse Iterators Back Using `base()`

You can convert reverse iterators back into normal iterators. To do this, reverse iterators provide the `base()` member function:

```
namespace std {
    template <typename Iterator>
    class reverse_iterator  ...  {
        ...
        Iterator base() const;
        ...
    };
}
```

Here is an example of the use of `base()`:

```
// iter/reviter4.cpp

#include <iterator>
#include <iostream>

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

int main()
{
    // create list with elements from 1 to 9
    list<int> coll = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

    // find position of element with value 5
    list<int>::const_iterator pos;
    pos = find (coll.cbegin(), coll.cend(),     // range
                5);                             // value

    // print value of the element
    cout << "pos:     " << *pos << endl;

    // convert iterator to reverse iterator
    list<int>::const_reverse_iterator rpos(pos);

    // print value of the element to which the reverse iterator refers
    cout << "rpos:    " << *rpos << endl;

    // convert reverse iterator back to normal iterator
    list<int>::const_iterator rrpos;
    rrpos = rpos.base();

    // print value of the element to which the normal iterator refers
    cout << "rrpos: " << *rrpos << endl;
}
```

The program has the following output:

```
pos:   5
rpos:  4
rrpos: 5
```

Thus, the conversion with `base()`

```
*rpos.base()
```

is equivalent to the conversion in a reverse iterator. That is, the physical position (the element of the iterator) is retained, but the logical position (the value of the element) is moved. See Section 11.5.3, page 539. for another example of the use of `base()` .

9.4.2. Insert Iterators

Insert iterators, also called *inserters*, are iterator adapters that transform an assignment of a new value into an insertion of that new value. By using insert iterators, algorithms can insert rather than overwrite. All insert iterators are in the output-iterator category. Thus, they provide only the ability to assign new values (see Section 9.2.1, page 433).

Functionality of Insert Iterators

Usually, an algorithm assigns values to a destination iterator. For example, consider the `copy()` algorithm (described in Section 11.6.1, page 557):

Click here to view code image

```
namespace std {
    template <typename InputIterator, typename OutputIterator>
    OutputIterator copy (InputIterator from_pos,   // beginning of source
                         InputIterator from_end,   // end of source
                         OutputIterator to_pos)    // beginning of dest.
    {
        while (from_pos != from_end) {
            *to_pos = *from_pos;     // copy values
            ++from_pos;              // increment iterators
            ++to_pos;
        }
        return to_pos;
    }
}
```

The loop runs until the actual position of the source iterator has reached the end. Inside the loop, the source iterator, `from_pos`, is assigned to the destination iterator, `to_pos`, and both iterators get incremented.

The interesting part is the assignment of the new value:

```
*to_pos = value
```

An insert iterator transforms such an assignment into an insertion. However, two operations are involved: First, operator `*` returns the current element of the iterator; second, operator `=` assigns the new value. Implementations of insert iterators usually use the following two-step trick:

1. Operator `*` is implemented as a no-op that simply returns `*this`. Thus, for insert iterators, `*pos` is equivalent to `pos`.
2. The assignment operator is implemented so that it gets transferred into an insertion. In fact, the insert iterator calls the `push_back()`, `push_front()`, or `insert()` member function of the container.

Thus, for insert iterators, you could write `pos=` *value* instead of `*pos=` *value* to insert a new value. However, I'm talking about implementation details of input iterators. The correct expression to assign a new value is `*pos=` *value*.

Similarly, the increment operator is implemented as a no-op that simply returns `*this`. Thus, you can't modify the position of an insert iterator. Table 9.7 lists all operations of insert iterators.

Table 9.7. Operations of Insert Iterators

Expression	Effect
**iter*	No-op (returns *iter*)
iter = value	Inserts *value*
++iter	No-op (returns *iter*)
iter++	No-op (returns *iter*)

Kinds of Insert Iterators

The C++ standard library provides three kinds of insert iterators: back inserters, front inserters, and general inserters. They differ in their handling of the position at which to insert a value. In fact, each uses a different member function, which it calls for the container to which it belongs. Thus, an insert iterator must be always initialized with its container.

Each kind of insert iterator has a convenience function for its creation and initialization. Table 9.8 lists the kinds of insert iterators and their abilities.

Table 9.8. Kinds of Insert Iterators

Name	Class	Called Function	Creation
Back inserter	`back_insert_iterator`	`push_back(`*value*`)`	`back_inserter(`*cont*`)`
Front inserter	`front_insert_iterator`	`push_front(`*value*`)`	`front_inserter(`*cont*`)`
General inserter	`insert_iterator`	`insert(`*pos*`,`*value*`)`	`inserter(`*cont*`,`*pos*`)`

Of course, the container must provide the member function that the insert iterator calls; otherwise, that kind of insert iterator can't be used. For this reason, back inserters are available only for vectors, deques, lists, and strings; front inserters are available only for deques and lists.

Back Inserters

A *back inserter*, or *back insert iterator*, appends a value at the end of a container by calling the `push_back()` member function (see Section 8.7.1, page 415, for details about `push_back()`). `push_back()` is available only for vectors, deques, lists, and strings, so these are the only containers in the C++ standard library for which back inserters are usable.

A back inserter must be initialized with its container at creation time. The `back_inserter()` function provides a convenient way of doing this. The following example demonstrates the use of back inserters:

```
// iter/backinserter1.cpp

#include <vector>
#include <algorithm>
#include <iterator>
#include "print.hpp"
using namespace std;

int main()
{
    vector<int> coll;

    // create back inserter for coll
```

```
    // - inconvenient way
    back_insert_iterator<vector<int> > iter(coll);

    // insert elements with the usual iterator interface
    *iter = 1;
    iter++;
    *iter = 2;
    iter++;
    *iter = 3;
    PRINT_ELEMENTS(coll);

    // create back inserter and insert elements
    // - convenient way
    back_inserter(coll) = 44;
    back_inserter(coll) = 55;
    PRINT_ELEMENTS(coll);

    // use back inserter to append all elements again
    // - reserve enough memory to avoid reallocation
    coll.reserve(2*coll.size());
    copy (coll.begin(), coll.end(),     // source
          back_inserter(coll));         // destination
    PRINT_ELEMENTS(coll);
}
```

The output of the program is as follows:

```
1 2 3
1 2 3 44 55
1 2 3 44 55 1 2 3 44 55
```

Note that you must not forget to reserve enough space before calling `copy()`. The reason is that the back inserter inserts elements, which might invalidate all other iterators referring to the same vector. Thus, if not enough space is reserved, the algorithm invalidates the passed source iterators while running.

Strings also provide an STL container interface, including `push_back()`. Therefore, you could use back inserters to append characters in a string. See Section 13.2.14, page 688, for an example.

Front Inserters

A *front inserter*, or *front insert iterator*, inserts a value at the beginning of a container by calling the `push_front()` member function (see Section 8.7.1, page 414, for details about `push_front()`). `push_front()` is available only for deques, lists, and forward lists, so these are the only containers in the C++ standard library for which front inserters are usable.

A front inserter must be initialized with its container at creation time. The `front_inserter()` function provides a convenient way of doing this. The following example demonstrates the use of front inserters:

```
// iter/frontinserter1.cpp

#include <list>
#include <algorithm>
#include <iterator>
#include "print.hpp"
using namespace std;

int main()
{
    list<int> coll;

    // create front inserter for coll
    // - inconvenient way
    front_insert_iterator<list<int> > iter(coll);

    // insert elements with the usual iterator interface
    *iter = 1;
    iter++;
    *iter = 2;
    iter++;
    *iter = 3;

    PRINT_ELEMENTS(coll);

    // create front inserter and insert elements
    // - convenient way
    front_inserter(coll) = 44;
    front_inserter(coll) = 55;

    PRINT_ELEMENTS(coll);
```

```
    // use front inserter to insert all elements again
    copy (coll.begin(), coll.end(),      // source
          front_inserter(coll));         // destination

    PRINT_ELEMENTS(coll);
}
```

The output of the program is as follows:

```
3 2 1
55 44 3 2 1
1 2 3 44 55 55 44 3 2 1
```

Note that the front inserter inserts multiple elements in reverse order. This happens because it always inserts the next element in front of the previous one.

General Inserters

A *general inserter*, or *general insert iterator*,[1] is initialized with two values: the container and the position that is used for the insertions. Using both, a general inserter calls the `insert()` member function with the specified position as argument. The `inserter()` function provides a convenient way of creating and initializing a general inserter.

[1] A general inserter is often simply called an *insert iterator*, or *inserter*. This means that the words *insert iterator* and *inserter* have different meanings: They are a general term for all kinds of insert iterators and are also used as names for a special insert iterator that inserts at a specified position rather than in the front or in the back. To avoid this ambiguity, I use the term *general inserter* in this book.

A general inserter is usable for all standard containers (except for arrays and forward lists), because these containers provide the needed `insert()` member function (see Section 8.7.1, page 413). However, for associative and unordered containers, the position is used only as a hint, because the value of the element defines the correct position. See Section 8.7.1, page 413, for details.

After an insertion, the general inserter gets the position of the new inserted element. In particular, the following statements are called:

```
pos = container.insert(pos,value);
++pos;
```

The assignment of the return value of `insert()` ensures that the iterator's position is always valid. Without the assignment of the new position for deques, vectors, and strings, the general inserter would invalidate itself. The reason is that each insertion does, or at least might, invalidate all iterators that refer to the container.

The following example demonstrates the use of general inserters:

Click here to view code image

```
// iter/inserter1.cpp

#include <set>
#include <list>

#include <algorithm>
#include <iterator>
#include "print.hpp"
using namespace std;

int main()
{
    set<int> coll;

    // create insert iterator for coll
    // - inconvenient way
    insert_iterator<set<int> > iter(coll,coll.begin());

    // insert elements with the usual iterator interface
    *iter = 1;
    iter++;
    *iter = 2;
    iter++;
    *iter = 3;

    PRINT_ELEMENTS(coll,"set:   ");

    // create inserter and insert elements
    // - convenient way
    inserter(coll,coll.end()) = 44;
    inserter(coll,coll.end()) = 55;

    PRINT_ELEMENTS(coll,"set:   ");
```

```
    // use inserter to insert all elements into a list
    list<int> coll2;
    copy (coll.begin(), coll.end(),              // source
          inserter(coll2,coll2.begin()));        // destination

    PRINT_ELEMENTS(coll2,"list: ");

    // use inserter to reinsert all elements into the list before the second element
    copy (coll.begin(), coll.end(),              // source
          inserter(coll2,++coll2.begin()));      // destination

    PRINT_ELEMENTS(coll2,"list: ");
}
```

The output of the program is as follows:

```
set:  1 2 3
set:  1 2 3 44 55
list: 1 2 3 44 55
list: 1 1 2 3 44 55 2 3 44 55
```

The calls of `copy()` demonstrate that the general inserter maintains the order of the elements. The second call of `copy()` uses a certain position inside the range that is passed as argument.

A User-Defined Inserter for Associative Containers

As mentioned previously, for associative containers, the position argument of general inserters is used only as a hint. This hint might help to improve speed but also might cause bad performance. For example, if the inserted elements are in reverse order, the hint may slow down programs a bit because the search for the correct insertion point always starts at a wrong position. Thus, a bad hint might even be worse than no hint. This is a good example of the need for supplementation of the C++ standard library. See Section 9.6, page 471, for such an extension.

9.4.3. Stream Iterators

A *stream iterator* is an iterator adapter that allows you to use a stream as a source or destination of algorithms. In particular, an istream iterator can be used to read elements from an input stream, and an ostream iterator can be used to write values to an output stream.

A special form of a stream iterator is a *stream buffer iterator*, which can be used to read from or write to a stream buffer directly. Stream buffer iterators are discussed in Section 15.13.2, page 828.

Ostream Iterators

Ostream iterators write assigned values to an output stream. By using ostream iterators, algorithms can write directly to streams. The implementation of an ostream iterator uses the same concept as the implementation of insert iterators (see Section 9.4.2, page 454). The only difference is that they transform the assignment of a new value into an output operation by using operator `<<`. Thus, algorithms can write directly to streams by using the usual iterator interface. Table 9.9 lists the operations of ostream iterators.

When the ostream iterator is created, you must pass the output stream on which the values are written. An optional string can be passed, written as a separator between single values. Without the delimiter, the elements directly follow each other.

Ostream iterators are defined for a certain element type `T`:

```
namespace std {
    template <typename T,
              typename charT = char,
              typename traits = char_traits<charT> >
    class ostream_iterator;
}
```

Table 9.9. Operations of ostream Iterators

Expression	Effect
`ostream_iterator<T>(ostream)`	Creates an ostream iterator for *ostream*
`ostream_iterator<T>(ostream,delim)`	Creates an ostream iterator for *ostream*, with the string *delim* as the delimiter between the values (note that *delim* has type `const char*`)
**iter*	No-op (returns *iter*)
iter = value	Writes *value* to *ostream*: *ostream<<value* (followed by *delim*, if set)
++iter	No-op (returns *iter*)
iter++	No-op (returns *iter*)

The optional second and third template arguments specify the type of stream that is used (see Section 15.2.1, page 749, for their meaning).[2]

[2] In older systems, the optional template arguments for the stream type are missing.

The following example demonstrates the use of ostream iterators:

```
// iter/ostreamiter1.cpp

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

int main()
{
    // create ostream iterator for stream cout
    // - values are separated by a newline character
    ostream_iterator<int> intWriter(cout,"\n");

    // write elements with the usual iterator interface
    *intWriter = 42;
    intWriter++;
    *intWriter = 77;
    intWriter++;
    *intWriter = -5;

    // create collection with elements from 1 to 9
    vector<int> coll = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

    // write all elements without any delimiter
    copy (coll.cbegin(), coll.cend(),
          ostream_iterator<int>(cout));
    cout << endl;

    // write all elements with " < " as delimiter
    copy (coll.cbegin(), coll.cend(),
          ostream_iterator<int>(cout," < "));
    cout << endl;
}
```

The output of the program is as follows:

```
42
77
-5
123456789
1 < 2 < 3 < 4 < 5 < 6 < 7 < 8 < 9 <
```

Note that the delimiter has type `const char*`. Thus, if you pass an object of type `string`, you must call its member function `c_str()` (see Section 13.3.6, page 700) to get the correct type. For example:

```
string delim;
...
ostream_iterator<int>(cout,delim.c_str());
```

Istream Iterators

Istream iterators are the counterparts of ostream iterators. An istream iterator reads elements from an input stream. By using istream iterators, algorithms can read from streams directly. However, istream iterators are a bit more complicated than ostream iterators (as usual, reading is more complicated than writing).

At creation time, the istream iterator is initialized by the input stream from which it reads. Then, by using the usual interface of input iterators (see Section 9.2.2, page 435), the istream iterator reads element-by-element, using operator `>>`. However, reading might fail (due to end-of-file or an error), and source ranges of algorithms need an "end position." To handle both problems, you can use an *end-of-stream iterator*, which is created with the default constructor for istream iterators. If a read fails, every istream iterator becomes an end-of-stream iterator. Thus, after any read access, you should compare an istream iterator with an end-of-stream iterator to check whether the iterator has a valid value. Table 9.10 lists all operations of istream iterators.

Table 9.10. Operations of istream Iterators

Expression	Effect
`istream_iterator<T>()`	Creates an end-of-stream iterator
`istream_iterator<T>(`*istream*`)`	Creates an istream iterator for *istream* (and might read the first value)
**iter*	Returns the value, read before (reads first value if not done by the constructor)
iter->member	Returns a member, if any, of the actual value, read before
++iter	Reads next value and returns its position
iter++	Reads next value but returns an iterator for the previous value
iter1== iter2	Tests *iter1* and *iter2* for equality
iter1 != iter2	Tests *iter1* and *iter2* for inequality

Note that the constructor of an istream iterator opens the stream and usually reads the first element. Otherwise, it could not return the first element when operator * is called after the initialization. However, implementations may defer the first read until the first call of operator * . So, you should not define an istream iterator before you need it.

Istream iterators are defined for a certain element type T :

```
namespace std {
    template <typename T,
              typename charT = char,
              typename traits = char_traits<charT>,
              typename Distance = ptrdiff_t>
    class istream_iterator;
}
```

The optional second and third template arguments specify the type of stream that is used (see Section 15.2.1, page 749, for their meaning). The optional fourth template argument specifies the difference type for the iterators.[3]

[3] In older systems without default template parameters, the optional fourth template argument is required as the second argument, and the arguments for the stream type are missing.

Two istream iterators are equal if

- both are end-of-stream iterators and thus can no longer read, or
- both can read and use the same stream.

The following example demonstrates the operations provided for istream iterators:

```
// iter/istreamiter1.cpp

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

int main()
{
    // create istream iterator that reads integers from cin
    istream_iterator<int> intReader(cin);

    // create end-of-stream iterator
    istream_iterator<int> intReaderEOF;

    // while able to read tokens with istream iterator
    // - write them twice
    while (intReader != intReaderEOF) {
        cout << "once:       " << *intReader << endl;
        cout << "once again: " << *intReader << endl;
        ++intReader;
    }
}
```

If you start the program with the following input:

```
1 2 3 f 4
```

the output of the program is as follows:

```
once:       1
once again: 1
once:       2
once again: 2
```

```
once:       3
once again: 3
```

As you can see, the input of character `f` ends the program. Due to a format error, the stream is no longer in a good state. Therefore, the istream iterator `intReader` is equal to the end-of-stream iterator `intReaderEOF`. So, the condition of the loop yields `false`.

Example of Stream Iterators and `advance()`

The following example uses both kinds of stream iterators and the `advance()` helper function:

```
// iter/advance2.cpp

#include <iterator>
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;

int main()
{
    istream_iterator<string> cinPos(cin);
    ostream_iterator<string> coutPos(cout," ");

    // while input is not at the end of the file
    // - write every third string
    while (cinPos != istream_iterator<string>()) {
        // ignore the following two strings
        advance (cinPos, 2);

        // read and write the third string
        if (cinPos != istream_iterator<string>()) {
            *coutPos++ = *cinPos++;
        }
    }
    cout << endl;
}
```

The `advance()` iterator function is provided to advance the iterator to another position (see Section 9.3.1, page 441). Used with istream iterators, `advance()` skips input tokens. For example, if you have the following input:[4]

[4] Thanks to Andrew Koenig for the nice input of this example.

```
No one objects if you are doing
a good programming job for
someone whom you respect.
```

the output is as follows:

```
objects are good for you
```

Don't forget to check whether the istream iterator is still valid after calling `advance()` and before accessing its value with `*cinPos`. Calling operator `*` for an end-of-stream iterator results in undefined behavior.

For other examples that demonstrate how algorithms use stream iterators to read from and write to streams, see Section 6.5.2, page 212; Section 11.6.1, page 560; and Section 11.7.2, page 582.

9.4.4. Move Iterators

Since C++11, an iterator adapter is provided that converts any access to the underlying element into a move operation. For example:

Click here to view code image

```
std::list<std::string> s;
...
std::vector<string> v1(s.begin(), s.end());              // copy strings into
v1

std::vector<string> v2(make_move_iterator(s.begin()),    // move strings into
v2
                       make_move_iterator(s.end()));
```

One application of these iterators is to let algorithms move instead of copy elements from one range into another. However, note that in general, the `move()` algorithm also does that (see Section 11.6.2, page 561).

In general, using a move iterator in algorithms only makes sense when the algorithm transfers elements of a source range to a destination

range. In addition, you have to ensure that each element is accessed only once. Otherwise, the contents would be moved more than once, which would result in undefined behavior.

Note that the only iterator category that guarantees that elements are read or processed only once is the input iterator category (see Section 9.2.2, page 435). Thus, using move iterators usually makes sense only when an algorithm has a source where the input iterator category is required and a destination that uses the output iterator category. The only exception is `for_each()`, which can be used to process the moved elements of the passed range (for example, to move them into a new container).