**Iterators**

Iterators are used to step through the elements of collections of objects. These collections may be containers or subsets of containers. The major advantage of iterators is that they offer a small but **common interface for any arbitrary container type**.

Iterators are used to step through the elements of collections of objects. These collections may be containers or subsets of containers. The major advantage of iterators is that they offer a small but common interface for any arbitrary container type.

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.

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.

# Header Files for Iterators

All containers define their own iterator types, so you don’t need a special header file for using iterators of containers. However, several definitions for special iterators, such as reverse iterators, and some auxiliary iterator functions are introduced by the <iterator> header file.

# Iterators as Smart Pointers

It’s often necessary to perform an operation on all the elements in the container (or perhaps a range of elements). In an ordinary C++ array, such operations are carried out using a pointer (or the [] operator, which is the same underlying mechanism).

**Ordinary Pointers Underpowered**

However, with more sophisticated containers, plain C++ pointers have disadvantages. For one thing, if the items stored in the container are not placed contiguously in memory, handling the pointer becomes much more complicated; we can’t simply increment it to point to the next value.

We may also want to store the address of some container element in a pointer variable so we can access the element at some future time. What happens to this stored pointer value if we insert or erase something from the middle of the container? It may not continue to be valid if the container’s contents are rearranged. It would be nice if we didn’t need to worry about revising all our stored pointer values when insertions and deletions take place.

One solution to these kinds of problems is to create a class of “smart pointers.” An object of such a class basically wraps its member functions around an ordinary pointer. The ++ and \* operators are overloaded so they know how to operate on the elements in their container, even if the elements are not contiguous in memory or change their locations. Here’s how that might look, in skeleton form:

class SmartPointer {

private:

float\* p; //an ordinary pointer

public:

float operator\*() { }

float operator++() { }

};

void main() {

...

SmartPointer sptr = start\_address;

for(int j=0; j<SIZE; j++)

cout << \*sptr++;

}

**Whose Responsibility?**

Should the smart pointer class be embedded in a container, or should it be a separate class? The approach chosen by the STL is to make smart pointers, called iterators, into a completely separate class (actually a family of templetized classes). The class user creates iterators by defining them to be objects of such classes.

# Iterators as an Interface

* Iterators also determine which algorithms can be used with which containers.
* Iterators provide a surprisingly elegant way to match appropriate algorithms with containers.
* How many kinds of iterators (cables) do you need to make this scheme work? As it turns out, only five types are necessary:

Output, Input, Forward, Bidirectional and Random Access

# Iterator Categories

* three major classes of iterators: forward, bidirectional, and random access
* two specialized kinds of iterators: input iterator and output iterator

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Iterator Type | R/W | Itertor can be saved | Direction | Access | Ability |
| Output | Write | No | Forward Only | Linear | Writes Forward |
| Input | Read | No | Forward Only | Linear | Reads Forward Once |
| Forward | R & W | Yes | Forward Only | Linear | Reads Forward |
| Bidirectional | R & W | Yes | Forward and back | Linear | Reads Forward and Backward |
| Random-access | R & W | Yes | Forward and back | Random | Reads with Random Access |

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

The iterator categories are concepts rather than classes. If you need to do something advanced with iterator categories, use iterator\_traits (directly or indirectly).

## Ouput Iterator

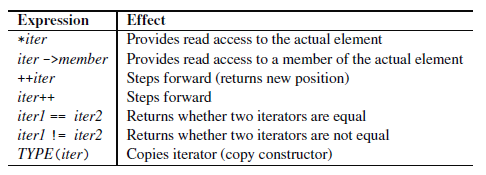
* Output iterators can only step forward with write access. Thus, you can assign new values only element-by-element.
* You can’t use an output iterator to iterate twice over the same range.
* In fact, it is even not guaranteed that you can assign a value twice without incrementing the iterator.
* the valid operations for output iterators are:

|  |  |
| --- | --- |
| 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 values.
* Usually, the end of a writing is defined by an additional requirement for specific output iterators.
* All **const\_iterators** 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.

## Input Iterator

* Input iterators can only step forward element-by-element with read access. Thus, they return values element-wise.
* the operations of input iterators are:



* 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.
* For input iterators, operators == and != are provided only to check whether an iterator is equal to a past-the-end iterator.
* 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

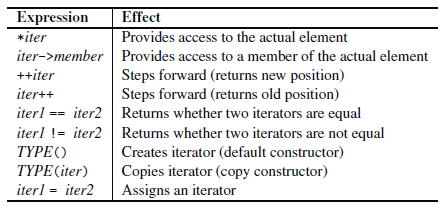
rather than

pos++ // OK, but not so fast

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

## Forward Iterator

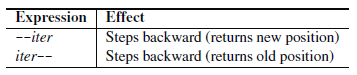
* Forward iterators are input iterators that provide additional guarantees while reading forward.
* the operations of forward iterators are:



* 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.
* 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.
* 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.

## Bi-directional Iterator

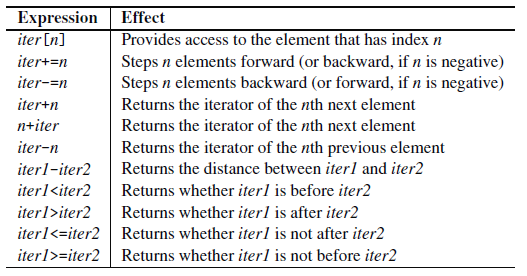
* Bidirectional iterators are forward iterators that provide the additional ability to iterate backward over the elements.
* Thus, they provide the decrement operator to step backward.



* 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.

## Random Access Iterator

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 >.



Additional Operations of Random-Access Iterators

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)
* you can use operator < as an end criterion in loops for random-access iterators only
* strictly speaking, moving an iterator to before the beginning and after the last element results in undefined behavior.
* A random-access iterator that fulfills the requirements of an output iterator is a mutable randomaccess iterator, which can be used for both reading and writing.

## The Increment and Decrement Problem of Vector Iterators

In general, you can increment and decrement temporary iterators. However, for arrays, 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.

**Reason**:

The reason for this strange problem lies in the fact that iterators of vectors, arrays, 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.

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, 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.

# Auxiliary Iterator Functions

The C++ standard library provides some auxiliary functions for dealing with iterators:

* advance()
* next()
* prev()
* distance()
* iter\_swap()

The first four give all iterators some abilities usually provided only for random-access iterators: to step more than one element forward (or backward) and to process the difference between iterators. The last auxiliary function allows you to swap the values of two iterators.

# Specialized Iterators

two specialized forms of iterators:

1. **iterator adapters**, which can change the behavior of iterators in interesting ways
2. **stream iterators**, which allow input and output streams to behave like iterators

## Iterator Adapters

STL provides three variations on the normal iterator:

* **reverse iterator** allows you to iterate backward through a container
* **insert iterator** changes the behavior of various algorithms, such as copy() and merge(), so they insert data into a container rather than overwriting existing data
* **raw storage iterator** allows output iterators to store data in uninitialized memory, but it’s used in specialized situations and we’ll ignore it here

These special iterators allow algorithms to operate in reverse, in insert mode, and with streams.

### Reverse Iterators

To iterate backward you can use a reverse iterator. You must use the member functions rbegin() and rend() when you use a reverse iterator. Confusingly, you’re starting at the end of the container, but the member function is called rbegin(). Also, you must increment the iterator. Don’t try to decrement a reverse iterator; revit-- doesn’t do what you want. With a reverse\_iterator, always go from rbegin() to rend() using the increment operator.

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.

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

Since C++11, corresponding crbegin() and crend() member functions are provided, which return read-only reverse iterators.

#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(), print); // range operation

cout << endl;

// print all elements in reverse order

for\_each (coll.rbegin(), coll.rend(), print); // range operations

cout << endl;

// print all elements in reverse order

for\_each (coll.crbegin(), coll.crend(), print); // range operations

cout << endl;

return 0;

}

Output:

1 2 3 4 5 6 7 8 9

9 8 7 6 5 4 3 2 1

9 8 7 6 5 4 3 2 1

**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.

#include <iterator>

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

void print (int elem) {

cout << elem << ' ';

}

int main() {

vector<int> coll = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

vector<int>::const\_iterator pos;

pos = find (coll.cbegin(), coll.cend(), 5);

cout << "pos: " << \*pos << endl;

vector<int>::const\_reverse\_iterator rpos(pos);

cout << "rpos: " << \*rpos << endl;

/\* convert iterators to reverse iterators \*/

vector<int>::const\_iterator pos1;

vector<int>::const\_iterator pos2;

pos1 = find (coll.cbegin(), coll.cend(), 2);

pos2 = find (coll.cbegin(), coll.cend(), 7);

vector<int>::const\_reverse\_iterator rpos1(pos1);

vector<int>::const\_reverse\_iterator rpos2(pos2);

for\_each (pos1, pos2, print); cout << endl;

for\_each (rpos2, rpos1, print); cout << endl;

return 0;

}

Output:

pos: 5

rpos: 4

2 3 4 5 6 6 5 4 3 2

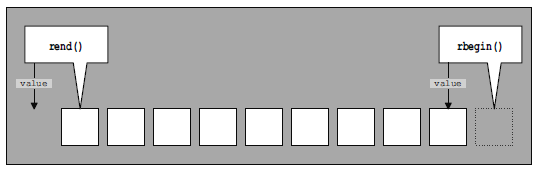
Thus, if you print the value of an iterator and convert the iterator into a reverse iterator, the value has changed. This is not a bug; it’s a feature! This behavior is a consequence of the fact that ranges are half open.

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.

rbegin() is simply: container::reverse\_iterator(end())

rend() is simply: container::reverse\_iterator(begin())

Of course, constant iterators are converted into type const\_reverse\_iterator.



Position and Value of Reverse Iterators

**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:

#include <iterator>

#include <iostream>

#include <list>

#include <algorithm>

using namespace std;

int main() {

list<int> coll = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

list<int>::const\_iterator pos;

pos = find (coll.cbegin(), coll.cend(), 5); // range value

cout << "pos : " << \*pos << endl;

// convert iterator to reverse iterator

list<int>::const\_reverse\_iterator rpos(pos);

cout << "rpos : " << \*rpos << endl;

// convert reverse iterator back to normal iterator

list<int>::const\_iterator rrpos;

rrpos = rpos.base();

cout << "rrpos : " << \*rrpos << endl;

return 0;

}

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.

### Insert Iterators

Some algorithms, such as copy(), overwrite the existing contents (if any) of the destination container. Sometimes, however, you’d rather that copy() inserted new elements into a container along with the old ones, instead of overwriting the old ones. You can cause this behavior by using an insert iterator.

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.

An insert iterator transforms such an assignment into an insertion. However, two operations are involved:

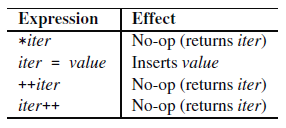
1. First, operator \* returns the current element of the iterator;
2. second, operator = assigns the new value.

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.

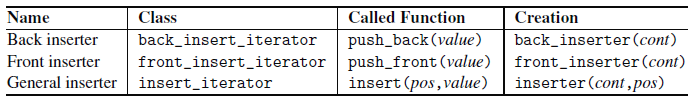
Thus, for insert iterators, you could write pos=value instead of \*pos=value to insert a new value. However, I’m talking about implementation details of input iterators. The correct expression to assign a new value is \*pos=value. Similarly, the increment operator is implemented as a no-op that simply returns \*this. Thus, you can’t modify the position of an insert iterator.

List of all operations of insert iterators:



Operations of Insert Iterators

**Kinds of Insert Iterators**



Kinds of Insert Iterators

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.

### Move Iterators

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

For example:

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()),

make\_move\_iterator(s.end())); // move strings into v2

One application of these iterators is to let algorithms move instead of copy elements from one range into another.

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. 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).

## Stream Iterators

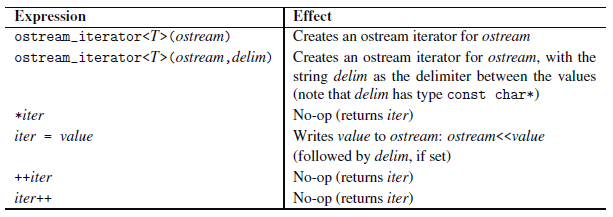
A stream iterator is an iterator adapter that allows you to use a stream as a source or destination of algorithms. Stream iterators allow you to treat files and I/O devices (such as cin and cout) as if they were iterators. This makes it easy to use files and I/O devices as arguments to algorithms.

* **istream iterator,** can be used to read elements from an input stream
* **ostream iterator,** can be used to write values to an output stream
* **stream buffer iterator**, which can be used to read from or write to a stream buffer directly

### The ostream\_iterator Class

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. 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.

operations of ostream iterators are:



Operations of ostream Iterators

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()

For example:

string delim;

ostream\_iterator<int>(cout, delim.c\_str());

### The istream\_iterator Class

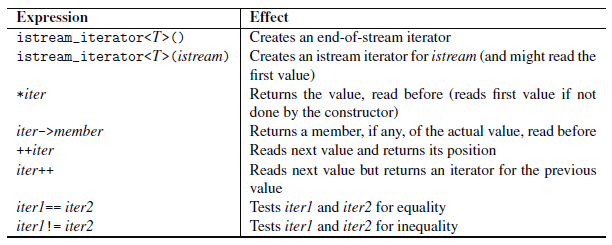
Istream iterators are the counterparts of ostream iterators. An istream iterator reads elements from an input stream. By using istream iterators, algorithms can read from streams directly.

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, 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.

operations of istream iterators are:



Operations of istream Iterators

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

Calling operator \* for an end-of-stream iterator results in undefined behavior.

# Iterator Traits

It might be useful or even necessary to be able to overload behavior for different iterator categories. By using iterator tags and iterator traits (both provided in <iterator>), such an overloading can be performed. For each iterator category, the C++ standard library provides an iterator tag that can be used as a “label” for iterators:

namespace std {

struct output\_iterator\_tag { };

struct input\_iterator\_tag { };

struct forward\_iterator\_tag : public input\_iterator\_tag { };

struct bidirectional\_iterator\_tag : public forward\_iterator\_tag { };

struct random\_access\_iterator\_tag : public bidirectional\_iterator\_tag { };

}

If you write generic code, you might not be interested only in the iterator category. For example, you may need the type of the elements to which the iterator refers.

Therefore, the C++ standard library provides a special template structure to define the iterator traits. This structure contains all relevant information about an iterator and is used as a common interface for all the type definitions an iterator should have (the category, the type of the elements, and so on):

namespace std {

template <typename T>

struct iterator\_traits {

typedef typename T::iterator\_category iterator\_category;

typedef typename T::value\_type value\_type;

typedef typename T::difference\_type difference\_type;

typedef typename T::pointer pointer;

typedef typename T::reference reference;

};

}

In this template, T stands for the type of the iterator. Thus, you can write code that, for any iterator, uses its category, the type of its elements, and so on. For example, the following expression yields the value type of iterator type T:

typename std::iterator\_traits<T>::value\_type

This structure has two advantages:

1. It ensures that an iterator provides all type definitions
2. It can be (partially) specialized for (sets of) special iterators

The latter is done for ordinary pointers that also can be used as iterators:

namespace std {

template <typename T>

struct iterator\_traits<T\*> {

typedef T value\_type;

typedef ptrdiff\_t difference\_type;

typedef random\_access\_iterator\_tag iterator\_category;

typedef T\* pointer;

typedef T& reference;

};

}

Thus, for any type “pointer to T,” it is defined as having the random-access iterator category. A corresponding partial specialization exists for constant pointers (const T\*).

Note that output iterators can be used only to write something. Thus, in the case of an output iterator, value\_type, difference\_type, pointer, and reference may be defined as void.

# Writing User-Defined Iterators

Let’s write an iterator. As mentioned in the previous section, you need iterator traits provided for the user-defined iterator. You can provide them in one of two ways:

1. Provide the necessary five type definitions for the general iterator\_traits structure
2. Provide a (partial) specialization of the iterator\_traits structure

# Iterators at Work

common uses of iterators:

1. Data Access
2. Data Insertion
3. Algorithms and Iterators: Algorithms use iterators as arguments (and sometimes as return values)