C++

Standard library: iterators

Introduction

An iterator is a pointer-like object used to traverse containers.

Each container has an iterator with a form appropriate to the representation of the container.
Example:
vector<double>::iterator vit; // to get an iterator to a vector of double

There are different categories of iterator. The category of the iterator distinguishes which algorithm can be used with which container.

Introduction

Iterators are generalizing pointers.

Like a pointer, an iterator can be used to access or modify an element (by dereference).

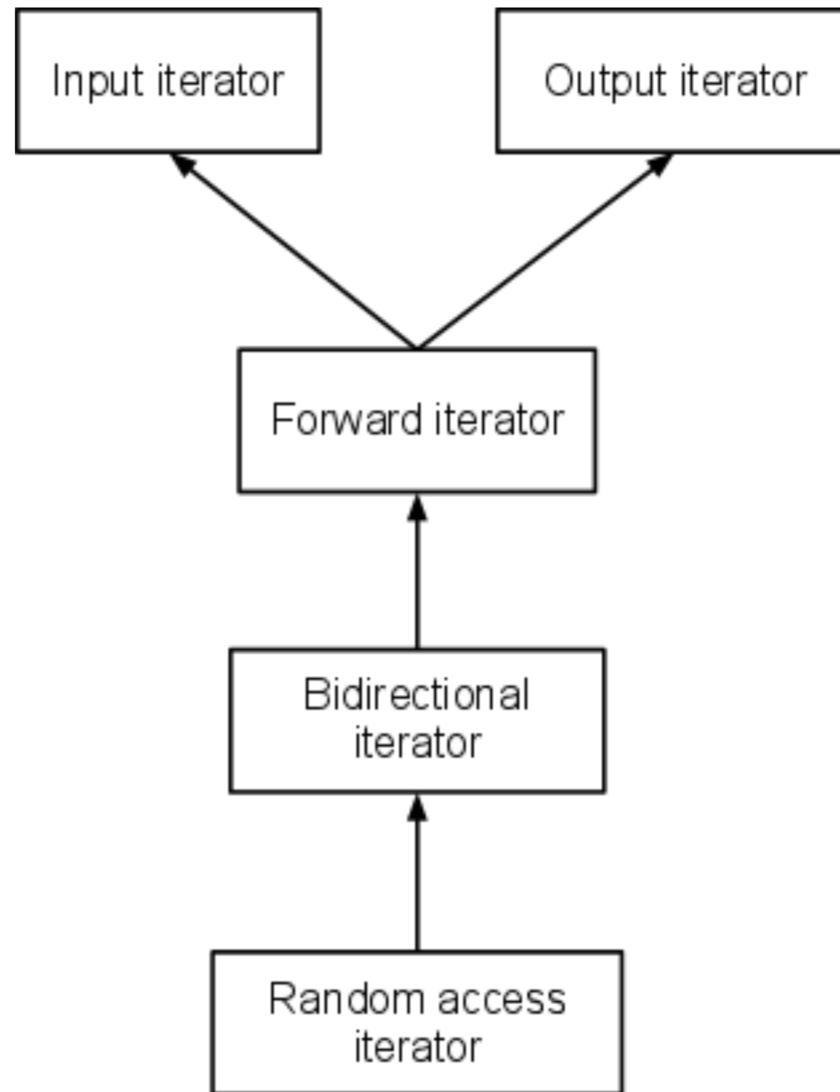
Iterator are modified by post-increment or pre-increment (operator++()) like pointers.

Ranges can be used to describe the content of a container with an iterator to the beginning of the container and with a special ending iterator.

Category of iterators

Form	Description
input iterator	Read only, forward moving
output iterator	Write only, forward moving
forward iterator	Read and Write, forward moving
Bidirectional iterator	Same as forward and can move backward
Random access iterator	Same as bidirectional with random access

Iterators hierarchy



Iterators hierarchy

Iterators are hierarchical.

Forward iterators can be used wherever an input or output iterators is used.

A bidirectional iterator can be used wherever a forward iterator is used.

A random iterator can be used wherever a bidirectional iterator is used.

Iterators varieties

Another characteristic of iterators is whether they can modify the elements held by the container.

A constant iterator can only access but not modify.

Input iterators are always constant.
Output iterators are never constant.
Other iterators may or may not be constant depending on how they are declared.

Example: Containers in the standard library have constant and non-constant forms of an iterator.

```
vector<double>::iterator vit; // non-constant
vector<double>::const_iterator cvit; // constant iterator
```

Iterators in the standard library

Iterator form	Generated by
input iterator	istream_iterator
output iterator	ostream_iterator
bidirectional iterator	list, set, multiset, map, multimap
random iterator	vector, deque, regular pointer

Example

Ex1: Get an iterator for a vector of double and print the content of the vector

```
void f(vector<double>& vd) {
 vector<double>::iterator vdit;
 for (vdit=vd.begin(); vdit!=vd.end(); ++vdit)
  cout << *vdit << " ";
}
```

Ex2: We can write a general function for printing elements of a container given a range of iterators

```
template<class In> void f(In first, In last) {
 In it;
 for (it=first; it!=last; ++it) cout << *it << " ";
}
```

Example

Find an element in a container delimited by a range of iterators:

```
template<class In, class T>
In find(In first, In last, const T& v) {
  In it;
  for (it = first; it != end; ++it) {
    if (*it == v) break;
  }
  return it;
}

// ...
vector<double> vd;
double a[] = {1.0, 2.0, 5.0, 3.0};
copy(a, a+4, back_inserter(vd)); // init vd with {1.0 ... 3.0}
vector<double>::iterator vit;
vit = find(vd.begin(); vd.end(); 5.0);
if (vit != vd.end()) cout << "found" << endl;
```

Inserter

To produce output in a container though an iterator, the iterator needs to be dereferenceable.

Example:

```
void f(vector<int>& vi) {
  fill_n(vi.begin(), 100, 1); // fill vi with 100 times the value 1
}
```

If vi does not have enough space for writing 100 values, the code above will corrupt the memory.

Helper functions are declared in <iterator> (namespace std) that allow us to insert elements in a container instead of overwriting existing elements.

Inserter

```
template <class Cont> back_insert_iterator back_inserter(Cont& c);
template<class Cont> front_insert_iterator front_inserter(Cont& c);
template<class Cont, class Out> inserter(Cont& c, Out p);
```

Example:

```
void f(vector<int>& vi) {
  fill_n(back_inserter(vi), 100, 1); // fill vi with 100 times the value 1
}
```

Example

```
int main(void) {
  int a[] = {3, 2, 1};
  int b[] = {4, 5, 6};
  list<int> res;
  // first insert into the front
  copy(a, a+3, front_inserter(res));
  // then insert into the back
  copy(b, b+3, back_inserter(res));
  copy(list.begin(), list.end(), ostream_iterator<int,char>(cout, "
"));
  cout << endl;
}

// result will be: 1 2 3 4 5 6
```

Reverse iterator

Standard containers provide reverse iterators for iterating in a sequence in reverse order.

Example:

```
void f(vector<double>& vd) {
 vector<double>::reverse_iterator rvit;
 for (rvit = rbegin(); rvit!=rend(); ++rvit)
  cout << *rvit << " ";
}
```

This code will iterate through vd in reverse order. If vd contains initially: {1, 2, 3, 4, 5}, then a call to f() will print: 5 4 3 2 1.

Stream iterator

The standard library provides streams for I/O.

cout and *cin* are streams for writing to the standard output and reading from the standard input.

stream iterators are iterators to fit stream I/O in the general framework of containers and algorithms.

Four iterator types are available:
istream_iterator for reading from an istream
ostream_iterator for writing in an ostream
istreambuf_iterator for reading from a stream buffer
ostreambuf_iterator for writing in a stream buffer

Stream iterator

Example:

```
ostream_iterator<int> os(cout); // write ints on the std output
*os = 7; // output 7
++os;
*os = 5; // output 5
++os;

istream_iterator<int> is(cin); // read ints from cin
int i1 = *is; // read first int
++is; // get ready for next read
int i2 = *is; // read second int
```