C++

Standard library: iterators

Introduction

Iterators are the glue holding container and algorithms together.

They provide an abstract view of data so that algorithms implementation do not have to be concerned with various data-structures.

Iterators simplify code for containers by relieving them from providing various access methods.

Iterators abstract data as sequence of objects.

Introduction

Each container has an iterator with a form appropriate to the representation of the container.

Each container define the following types:
iterator - behaves like value_type*
const_iterator - behaves like const value_type*
reverse_iterator - view a container in reverse order;
  like value_type*
const_reverse_iterator - view a container in reverse order;
  like a const value_type*

Introduction

Examples:

```
// define an iterator to a vector of double
std::vector<double>::iterator vit;
// define a const iterator to a vector of double
std::vector<double>::const_iterator cvit;

// define an iterator to a list of int
std::list<int>::iterator lit;
// define a reverse iterator to a list of int
std::list<int>::reverse_iterator rlit;
```

Iterator and sequence

An iterator is an abstraction of the notion of a pointer to an element of a sequence.

It is a “pointer-like” object used to traverse containers.

Key operations are:
  Access the element currently pointed to (dereferencing, operators * and ->)
  Point to the next element in the sequence (increment, operator ++)
  Equality (operator ==)

Example: int* is an iterator over an array of int (int[]), std::list<int>::iterator is an iterator over a list of int

Iterator and sequence

With the exception of stack, queue and priority_queue, each container of the STL has its own appropriate kind of iterator.

An iterator allows to traverse the containers sequentially.
Example: traverse a vector of int and print each element

```
std::vector<int> v;
std::vector<int>::const_iterator cvit;
for (cvit = v.begin(); cvit != v.end(); ++cvit)
    std::cout << *cvit << std::endl;
```

Iterator and sequence

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. It does not have to cover the entire container but can only partially cover it.

Example:

```
std::vector<int> v;
std::vector<int>::const_iterator cvit;
// Full container; range [begin, end);
// corresponds to: v[0] … v[n-1] where n=v.size()
for (cvit = v.begin(); cvit != v.end(); ++cvit)

// Partial traversal; range [begin, begin + 5);
// corresponds to: v[0] … v[4]
for (cvit = v.begin(); cvit != v.begin() + 5; ++cvit)
```

Iterator categories

There are different categories of iterator.

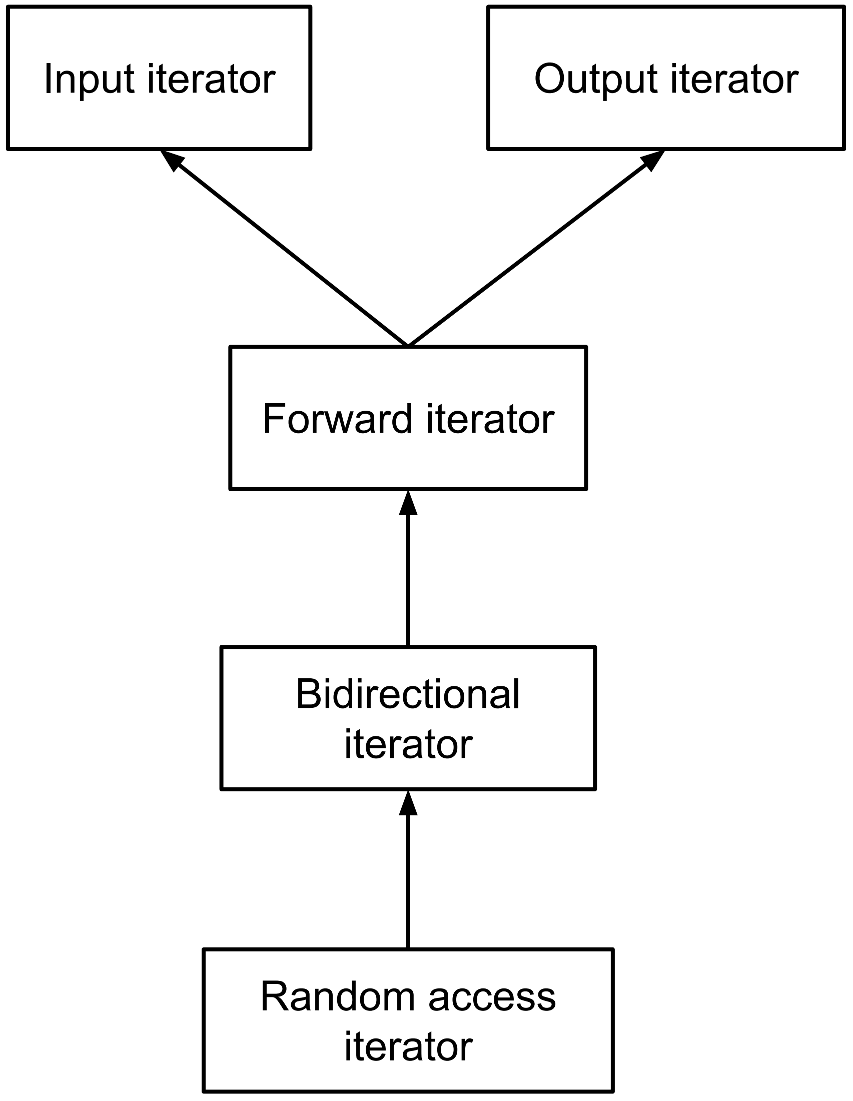
They correspond to the type of operations that are supported by an iterator. List of operations: read, write, iteration, comparison.

The category of the iterator distinguishes which algorithm can be used with which container. For example: a shuffle algorithm requires random access to a sequence so it requires a random iterator.

Iterator categories

Form	Description
input iterator	Single read, forward moving
output iterator	Single write, 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.

Iterator operations and categories

Category:	output	input	forward	bidirectional	random access
Read:		=*p	=*p	=*p	=*p
Access:		->	->	->	-> []
Write:	*p=		*p=	*p=	*p=
Iteration:	++	++	++	++ --	++ -- + - += -=
Comparison:		== !=	== !=	== !=	== != < > <= >=

Iterators varieties

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

Independently of its category, an iterator can allow const or non-const access to the object it points to.

A constant iterator can only access the object it points to but not modify it.

Examples:

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

Iterator varieties

Example 1: For reading data from a container, a const iterator is sufficient

```
vector<int> v;
vector<int>::const_iterator cvit;
for (cvit = v.begin(); cvit != v.end(); ++cvit)
     cout << *cvit << endl;
```

Example 2: For modifying data in a container, a non-const iterator is needed

```
vector<int>::iterator vit;
for (vit = v.begin(); vit != end(); ++vit)
     *vit = *vit * 2;
// it would be an error to use a const iterator here
```

Iterators varieties

Remarks:
Input iterators are always const, since by definition they are used for reading

Output iterators are never constant, since by definition they are used for writing

Other iterators may or may not be constant depending on how they are declared (See the two previous examples)

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

Examples

Example 1:

```
void print(const vector<double>& vd) {
 vector<double>::const_iterator cvdit;
 for (cvdit=vd.begin(); cvdit!=vd.end(); ++cvdit)
  cout << *cvdit << " ";
}
```

Notes:
Traverse the vector in order and print the content of each element.
Only reading is needed, so a const iterator is sufficient.
Because vd is passed as a const, it is not possible to get a non-const iterator anyway.

Examples

Example 2:

```
void print(const map<string, int>& dic) {
  map<string, int>::const_iterator cit;
  for (cit = dic.begin(); cit != dic.end(); ++cit) {
    cout << cit->first << ": " << cit->second << endl;
  }
}
```

Notes:

Traverse a map; note that the order is not the one in which the elements were inserted (like for a *vector*) but the order in which the keys are stored.

Elements of the map are: *pair<Key, Value>*; access is done through the *first* and *second* fields of the dereferenced iterator.

Examples

Example 3: Generalize the printing function

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

// vector<int> v
print(v.begin(), v.end());

// list<int> l;
print(l.begin(), l.end());
```

Note: for *print* to work with *map*, *operator*<< needs to be overloaded to work with *pair* objects

Inserter

To produce output in a container through an iterator, the iterator needs to be dereferenceable (and can be written to).
This implies the possibility of overflow and memory corruption.
Consider for example:

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

If vi does not have enough space for writing 100 values, the code above will corrupt the memory.
Note: *fill_n* is declared in <*algorithm*> (in namespace *std*) and fills n elements of a container with a given value

Inserter

To solve such recurrent problems, helper functions are declared in <*iterator*> (in the namespace *std*) that allow to insert elements in a container instead of overwriting existing elements:

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

back_inserter() causes elements to be added to the end of the container
front_inserter() causes elements to be added to the front
inserter() causes elements to be added before its iterator argument (*p* above)

Inserter

With these helpers the previous function is rewritten as follow:

Example:

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

and the following usage would be now correct:

```
vector<int> v;
f(v);
```

while previously it would corrupt the memory and result in undefined behavior

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(res.begin(), res.end(), ostream_iterator<int,char>(cout, "
"));
  cout << endl;
}
// result will be: 1 2 3 4 5 6
```

Note: *copy* is defined in the header <*algorithm*> and is in the namespace *std*

Inserter

These helper functions return a *back_insert_iterator*, a *front_insert_iterator*, or *insert_iterator* object.
They are defined in the header <*iterator*> in the namespace *std*

Clearly these inserters are output iterators.

As the name suggests, *front_insert_iterator* can be constructed only out of a container supporting front insertion (that is: list or dequeue but not vector)

Reverse iterator

Standard containers provide reverse iterators for iterating in a sequence in reverse order.
Containers provide the methods *rbegin()* and *rend()* for that purpose

Example:

```
 // vector<double> vd
 vector<double>::reverse_iterator rvit;
 for (rvit = vd.rbegin(); rvit!= vd.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.
They are declared in header <iterator> and in the *std* namespace

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
```

Stream iterator example

Example 1:

```
// assume vector<int> v
copy(istream_iterator<int> (cin), istream_iterator<int> (),
back_inserter(v));
```

will read integers from stdin (until the end of the stream) and insert them into the vector v

Example 2:

```
copy(v.begin(), v.end(), ostream_iterator<int>(cout, " "));
```

will write the integers in the vector v to stdout (separated by a space: “ “)

From iterators to generic algorithm

With iterators, it is possible to write function for processing containers without having to think about details of various data-structures

Examples: find an element with a given value in a sequence, find the minimum element of a sequence, sort a sequence, etc

Example

Find an element with a given value 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 != last; ++it) {
    if (*it == v) break;
  }
  return it;
}
```

Note: *find()* returns either an iterator to the (first) found element with the value *v*; or the iterator *last* (which is past the sequence of elements inspected)

The only operations used on the iterator are: read, equality and increment; therefore an input iterator is sufficient

Example

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

```
// ...
int main(void) {
  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 the element is not found, find returns an iterator past the
  // sequence being traversed
  if (vit != vd.end()) cout << "found" << endl;
  return 0;
}
```