Published on O'Reilly Network (http://www.oreillynet.com/)
http://www.oreillynet.com/pub/a/network/2005/10/18/what-is-iterator-in-c-plus-plus.html
See this if you're having trouble printing code examples

What Is an Iterator in C++, Part 1

by Ryan Stephens, coauthor of C++ Cookbook
10/18/2005

Iterator
: An iterator in C++ is a concept that refines the iterator design pattern into a specific set of behaviors that work well with the C++ standard library. The standard library uses iterators to expose elements in a range, in a consistent, familiar way. Anything that implements this set of behaviors is called an iterator. This paves the way for generic algorithms and makes it easy to implement your own iterators and have them integrate smoothly with the standard library.

In This Article:

When I buy fresh meat at my supermarket, I don't care how it got there. I want to start at one end of the refrigerated glass case and examine the contents of each bin until I'm at the other end. Along the way, if I see something on the other side of the glass that I want, I will summon the butcher to weigh it and pack it up. It makes no difference to me that the meat in one bin came from Montana, and another from a local farm. The interface is the important part; from one supermarket to another I know what to expect.

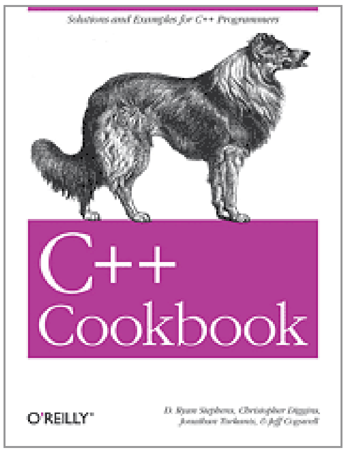
Such is the case with the iterator pattern. The iterator pattern describes a set of requirements that allows a consumer of some data structure to access elements in it with a familiar interface, regardless of the internal details of the data structure. The C++ standard library containers supply iterator interfaces, which makes them convenient to use and interoperable with the standard algorithms.

This is the first part of a two-part article. In this installment, I'll give a brief overview of the iterator pattern and what an iterator is in C++. In the second installment, I will show you how to implement your own iterator in the same manner as the standard library containers.

The Iterator Pattern

Iterators are not unique to C++. The concept of an iterator is something that allows two parties--generally the consumer of some data structure or "client code", and the implementer of the data structure, or "library code"--to communicate without concern for the other's internal details. This principle of intentional ignorance is what lets a collection of elements (in any language) expose those elements to the outside world without revealing the details of the collection's internal implementation, i.e. whether it is a hash table, linked list, tree, or some other sort of data structure.

Probably the best definition of the iterator pattern is in *Design Patterns*, by



Erich Gamma, et al, (Addison-Wesley). The authors provide a short description of the intent of iterators:

> Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

The use cases they reference use iterators to access elements in one of these "aggregate objects," a la the C++ standard containers `vector`, `list`, `map`, and so on, and in fact, the sample implementation they give is in C++. But most modern languages provide iterators in some form: Java has an `Iterator` class, and C# has enumerators.

A great deal of programming has to do with manipulating sequences of elements, which is why, despite its simplicity, the notion of an iterator is so broadly applicable. You usually do the same sorts of things to a sequence of elements regardless of its type: iterating through every element, searching, sorting, inserting, deleting, and so on. Without a common interface (whether an iterator or something else), you would have to do the same things to different data structures in different ways. This would be a sad state of affairs; luckily we have iterators.

The iterator pattern defines a handful of simple requirements. An iterator should allow its consumers to:

- Move to the beginning of the range of elements
- Advance to the next element
- Return the value referred to, often called the referent
- Interrogate it to see if it is at the end of the range

If all of these requirements are met, then consumers of an iterator will be able to traverse a range of elements in some aggregate object with a minimum of effort. As you will see, C++ iterators provided by the standard library satisfy all of these requirements, though not exactly as they are outlined in *Design Patterns*. With that in mind, let's discuss what a C++ iterator *is*.

Iterators in C++

The C++ standard library provides iterators for the standard containers (for example, `list`, `vector`, `deque`, and so on) and a few other noncontainer classes. You can use an iterator to print the contents of, for example, a `vector` like this:

```
vector<int> v;
// fill up v with data...
for (vector<int>::iterator it = v.begin();
     it != v.end(); ++it) {
   cout << *it << endl;
}
```

The variable `it`
is the iterator. This use case illustrates how the iterator pattern requirements are implemented in C++. Obtain an iterator to the first element in a container by calling that container's `begin` member function. Advance an iterator to the next element with the pre- or post-increment operator, as in `++it` or `it++`. Get the value it refers to with the pointer dereference operator `*`, as in `*it`. Finally, you can see if an iterator is at the end of a range by comparing it to the iterator returned by the container's `end`
member function, which returns an iterator that refers to one past the end of the elements. This is why the continuation test for the `for` loop in the example above is `it != v.end()`. These one-past-the-end semantics are important in C++, so let's talk about that for a moment.

The iterator returned by a container's `end`
member function represents a logical element that's one past the last element in a container, not the physical memory location that's just beyond the last element (which doesn't make sense for some kinds of data structures anyway). You should never dereference it, because it is just a marker and holds nothing of value. The point of such a construct is to provide a logical end marker for a range, regardless of the context in which that range is used. Standard algorithms and all member functions that work with a range of iterators use this convention. This is why you can use standard algorithms, such as `sort` in `<algorithm>`, like this:

```
sort(v.begin(), v.end());
```

`sort` sorts everything in the range up to, but not including, the iterator supplied as its second argument.

You can see that C++ iterators permit the same operations as the iterator pattern requires, but not literally. It's all there: move to the beginning, advance to the next element, get the referent, and test to see if you're at the end. In addition, different categories of iterators support additional operations, such as moving backward with the decrement operators (`--it` or `it--`), or advancing forward or backward by a specified number of elements. I will explain iterator categories a little later.

The above example won't work if you are working with a `const` container though (it won't even compile). In this case, you need to use a `const_iterator`, which works just like the `iterator` type in the example above, except that when you dereference it, you get back a `const` object. Therefore, a function to print the contents of a `const` container might look like this:

```
void printCont(const vector<int>& v) {
   for (vector<int>::const_iterator it = v.begin();
        it != v.end(); ++it) {
      cout << *it << endl;
   }
}
```

Incidentally, even if you aren't working with a `const` object, it is a good idea to use a `const_iterator` if you don't plan on modifying the elements in a container. This way, the compiler will keep you honest if you mistakenly try to modify an element.

There are a couple of important points to make here. First, note that the `iterator` and `const_iterator` types above are not part of the C++ language, they are just classes implemented in the standard library (just like the containers). Second, the exact type of an iterator is specific to the container (or other object) it is being used with, which is why you have to declare it using a name that includes the container name, such as `vector<int>::iterator`. Furthermore, each standard library implementation is free to choose the specific type of a container's iterator types, so long as it exposes it to the user of a container using the names `iterator` and `const_iterator`. This can be done with a public `typedef`. The actual type may be a pointer or a class whose structure and behavior are different from one standard library to another (though they must all support the specific behavior required by the standard).

Use Cases

If you use the standard containers, then you are probably using iterators in one of a few use cases. First of all, there is the simple task of iterating through elements, as I showed in the examples above. You will probably also want to use the standard algorithms with iterators, such as `copy`:

```
list<string> x;
x.push_back("peach");
x.push_back("apple");
x.push_back("banana");

// Make y the same size as x.  This creates
// empty strings in y's elements.
list<string> y(x.size());

// This is std::copy...
copy(x.begin(), x.end(), y.begin());
```

Or, you may want nest calls to standard algorithms. The following line copies elements from `x` to `y` starting at the element "apple," if it is found:

```
copy(find(x.begin(), x.end(), "apple"), x.end(), y.begin());
```

`find` returns an iterator that refers to the first element it finds that is equal to its argument. If no element is found, it returns its second argument, which is one past the end of the range.

You can also use the `for_each` algorithm to call a function for each element in the range:

```
template<typename T>
void print(const T& val) {
   cout << val << endl;
}
// ...
for_each(y.begin(), y.end(), print<string>);
```

`for_each` works with a function like my `print`
function above, or with a function object. Each of these algorithms will work on any of the standard containers, because the algorithms use ranges of iterators and therefore make no assumptions about the type of container.

What Is an Iterator?

So, what's really going on with the syntax? If `vector<int>::const_iterator` can be a `typedef` for some other type, then what is the actual *type* of the iterator?

The answer is shamelessly dodgy: it depends. As far as the standard containers go, the standard does not specify exactly what an iterator is. What it does say is that, for example, the class `vector` has to at least have a public `typedef` that looks like the following, where the token `unspecified` is a type that can be whatever the implementation wants:

```
// In <vector>, within the declaration of vector
typedef unspecified iterator;
typedef unspecified const_iterator;
```

Your favorite standard library implementation can use any type it wants in place of `unspecified`, so long as that type meets the strict requirements for the operations it must provide as defined by the standard. These requirements, apart from those I enumerated a moment ago, are dictated by the *category* of the iterator provided by the container. Categories are collections of requirements that describe the different forms of iterator, and they are the subjects of the next section. Allowing implementations to choose their iterator type permits flexibility to use different approaches in different libraries.

All categories satisfy the requirements defined by the iterator pattern, but, as I said, not literally. Table 1 explains how the iterators in the C++ standard library satisfy the requirements of the pattern.

Table 1. How C++ meets iterator requirements

Requirement	C++ Manifestation
Move to the beginning of the range	All standard containers, and some collections that are not strictly containers, provide a `begin` member function, which returns an iterator that refers to the first element in the container, if it exists.
Advance to the next element	Using pointer semantics, the prefix and postfix forms of the `++` operator move the iterator to the next element. Some iterator categories also support `--`, `-=`, `+=`, `+`, and `-` operators.
Return the referent	Also using pointer semantics, the dereference operator `*` is used to return the referent. For example, if `p` is an iterator, then `x = *p` would assign the value referred to by `p` to `x` (assuming `x` and the referent are compatible types).
Interrogate the iterator to see if it is at the end of the range	Similar to the `begin` member function, there is an `end` member function on all standard containers that returns an iterator that refers to one past the end of the container. Dereferencing this iterator yields undefined behavior, but you can compare the value returned by `end` to the current iterator to test if you are done iterating through the range.

The definition I usually read for C++ iterators is something like, "a generalization of a pointer", or, "an abstraction of a pointer." Both of these are technically correct, but somewhat high-level. Here's a more concrete one: an iterator is any type that behaves like an iterator. What this means is that any type that supports the interface described in Table 1 is an iterator, including a pointer to an element in a plain, static array. The following types are iterators:

- An `iterator` or `const_iterator typedef` on a standard container
- A pointer to an element in an array of objects
- Your `Iterator` class that you will read about in the second part of this article

But it does not mean that these types are equivalent, nor does it mean that they all inherit from a common base class. It means that you can use each one of these with pointer semantics: if `i` is an iterator, then `*i` returns the object referred to by the iterator; `i++` advances the iterator to the next element; and where `(*i).f` is valid, meaning `i` refers to something that has a member function or variable named `f`, `i->f` is also valid.

Categories

An iterator category is a set of requirements that defines a certain type of behavior. A category is an *interface*, though not in any mechanical sense, i.e. it is not an abstract base class.

Since an iterator is just a collection of requirements, and not a class hierarchy, expressing different kinds of iterators is a little unusual in this object-oriented world we live in. The way the standard describes it, there are five sets of requirements that define five different categories of iterators: input, output, forward, bidirectional, and random access.

The requirements for each category are little more than the list of member functions each iterator category supports and their associated semantics, with a few footnotes about behavior. Table 2 shows which member functions are supported by each iterator category. Assume that `Iter` is the iterator type, and `x` and `y` are variables of type `Iter`.

Table 2. Iterator categories and required member functions

Member Function	Input	Output	Forward	Bidirectional	Random Access
`Iter x(y)`	Y	Y	Y	Y	Y
`Iter x = y`	Y	Y	Y	Y	Y
`x == y`	Y	N	Y	Y	Y

Member Function	Input	Output	Forward	Bidirectional	Random Access
`x != y`	Y	N	Y	Y	Y
`x++, ++x`	Y	Y	Y	Y	Y
`x--, --x`	N	N	N	Y	Y
`*x`	As rvalue	As lvalue	Y	Y	Y
`(*x).f`	Y	N	Y	Y	Y
`x->f`	Y	N	Y	Y	Y
`x + n`	N	N	N	N	Y
`x += n`	N	N	N	N	Y
`x - n`	N	N	N	N	Y
`X -= n`	N	N	N	N	Y
`X[n]`	N	N	N	N	Y

In Table 2, the categories become more functional as you move from left to right. Input and output iterators permit relatively few operations, while random access iterators do everything.

The most basic iterator categories are input and output iterators. Input iterators are generally for reading elements from some collection in a single pass, such as an input stream. The idea is that the input iterator refers to a range of elements that can be read from, but not written to. As a result, the dereference operator yields rvalues. An output iterator is just the opposite, where you write elements to a collection that will only be traversed once. The biggest difference between the two is that dereferencing an output iterator yields an lvalue, so you can write to it, but you cannot read from it. And output iterators do not support testing for equality.

A forward iterator can do everything an input or output iterator can do, which means you can read from a dereferenced value or write to it, but since it is a "forward" iterator, not surprisingly, you can only go forward using a prefix or postfix `++` operator; for example, `++p` or `p++`.

Bidirectional and random access iterators do what their name implies. With a bidirectional iterator, you can advance forward or backward with the `++` or `--`
operators. A random access iterator can do everything any other iterator can do, and it can advance a given number of places a la pointer arithmetic. For example, the standard container `vector` supports random access iterators, which means that the following code will move the iterator around in various ways:

```
vector<string> v;
// Fill up v with some data
vector<string>::iterator p = v.begin();
p += 5;  // Now p refers to the 5th element
p[5];    // Now p refers to the 10th element
p -= 10; // Back to the beginning...
```

Different standard containers offer different types of iterators, depending on what can be efficiently supported, based on the type of data structure that is used by the container. For example, a `list` (declared in the `<list>` header) provides bidirectional iterators because `lists` are usually implemented as a doubly-linked `list`, which makes it efficient to iterate forward and backward one element at a time. `list` does not provide random access iterators, though, not because it's impossible to implement, but because it can't be implemented efficiently. Random access in a `list`
would require linear complexity for advancing forward or backward more than one element.

Each standard container supports the category of iterator it can implement efficiently. Standard algorithms advertise their requirements for an iterator by the category of iterator each requires. This declaration of iterator categories by algorithms and containers is what determines which algorithms will work with which containers. Table 3 contains a list of the standard containers and the category of iterator each supports.

Table 3. Iterator categories for standard containers

Container	Iterator Category
`basic_string`	Random access
`deque`	Random access

Container	Iterator Category
`list`	Bidirectional
`map`, `multimap`	Bidirectional
`set`, `multiset`	Bidirectional
`vector`	Random access

`basic_string`
isn't a standard container proper, but it supports most of the same operations as a container, so I included it in the table. If the name `basic_string` doesn't look familiar to you, it might be because you've been using its `typedef`'d shortcut: `string` or `wstring`.

Right now, if you glance back up at Table 2 you might say, "What happened to the input, output, and forward iterators?" And you'd be making a good point, if perhaps whining a little. Those categories aren't supplied by the containers; they're used for other things. In particular, input and output iterators are used with input and output stream iterators (which I will describe in the second part of this article). Forward iterators, even if they aren't supplied by any container, allow algorithms to make it clear that they only require the iterators to go forward. Consider the remove standard algorithm. It operates on a range, but only needs to go forward, so its declaration looks like this:

```
template<typename Fwd, typename Val>
Fwd remove(Fwd start, Fwd end, const Val& val)
```

The `Fwd` template parameter is supposed to let you know that its type is a forward iterator. `Val` is the type of elements in the range. (All elements that are equal to `val` are moved to the end of the range and an iterator to the first one of these elements is returned so you can erase them with the container's erase member function.)

Recap

The C++ standard library contains iterators for the standard containers that implement the iterator pattern, although not literally. Using iterators (or const iterators) is the preferred method of traversing elements in a container. The exact type of an iterator is implementation-defined, but that doesn't matter to you because regardless of exactly what it is, it still supports the interface its category requires. And finally, iterator categories define groups of requirements that each container or algorithm (standard or otherwise) can supply or require.

In part two of this article, I'll describe some more flavors of iterator (namely reverse iterators and stream iterators), and show you how to write an iterator for your own class.

Ryan Stephens is a software engineer, writer, and student living in Tempe, Arizona. He enjoys programming in virtually any language, especially C++.

Return to the O'Reilly Network