C++ Iterators

This is a quick summary of **iterators** in the Standard Template Library. For information on defining iterators for new containers, see here.

> **Iterator**: a pointer-like object that can be incremented with `++`, dereferenced with `*`, and compared against another iterator with `!=`.

Iterators are generated by STL container member functions, such as `begin()` and `end()`. Some containers return iterators that support only the above operations, while others return iterators that can move forward and backward, be compared with `<`, and so on.

The generic algorithms use iterators just as you use pointers in C to get elements from and store elements to various containers. Passing and returning iterators makes the algorithms

- more generic, because the algorithms will work for **any** containers, including ones you invent, as long as you define iterators for them
- more efficient (as discussed here)

Some algorithms can work with the minimal iterators, others may require the extra features. So a certain algorithm may require certain containers because only those containers can return the necessary kind of iterators.

An Example

Here's an example call to `copy()`, an algorithm that we'll use in our examples below:

```
copy(v.begin(), v.end(), l.begin());
```

`copy()` takes three arguments, all iterators:

- an iterator pointing to the first location to copy from
- an iterator pointing **one element past** to the last location to copy from
- an iterator pointing to the first location to copy into

In this case, `v` and `l` are some STL containers and `begin()` and `end()` are member functions that return iterators pointing to locations within those containers.

> **Note 1:** `begin()` returns a location you can dereference. `end()` does not. Dereferencing the end pointer is an error. The end pointer is only to be used to see when you've reached it.

> **Note 2:** `copy()` assumes that the destination already has room for the elements being copied. It would be an error to copy into an empty list or vector. However, this limitation is easily overcome with insert operators.

Iterator Classes

Iterators are divided into classes. These are not real C++ classes, but simply categories of kind of iterators. Each category specifies the operations the iterator supports. For example, some iterators support incrementing but not decrementing, some support dereferencing for getting data but not for storing data, some support scalar arithmetic, i.e., adding `n`, and some don't. Each STL container defines what class of iterators it can return. Each algorithm specifies what class of iterators it requires. The more powerful iterator classes are usually subclasses of the weaker ones, so if an algorithm requires a minimal iterator, it will work just fine with an iterator with more power.

InputIterator

`InputIterator` is a useful but limited class of iterators. If `iter` is an `InputIterator`, you can use:

- `++iter` and `iter++` to increment it, i.e., advance the pointer to the next element
- `*iter` to dereference it, i.e., get the element pointed to
- `== and !=` to compare it another iterator (typically the "end" iterator)

All STL containers can return at least this level of iterator. Here's typical code that prints out everything in a vector:

```
vector<int> v;
vector<int>::iterator iter;

v.push_back(1);
v.push_back(2);
v.push_back(3);

for (iter = v.begin(); iter != v.end(); iter++)
  cout << (*iter) << endl;
```

This is called an input iterator because you can only use it to "read" data from a container. You can't use it to store data, that is,

```
*iter = 4;
```

is illegal if `iter` is no more than an input iterator. It will work for the iterator above because vectors return iterators more powerful than just input iterators. But if `iter` were an istream iterator (discussed shortly), then the above restriction would apply.

OutputIterator

`OutputIterator` is another limited class of iterators, basically the opposite of `InputIterator`. If `iter` is an `OutputIterator`, you can use:

- `++iter` and `iter++` to increment it, i.e., advance the pointer to the next element
- `*iter = ...` to store data in the location pointed to

Output iterators are only for storing. If something is no more an output iterator, you can't read from it with `*iter`, nor can you test it with `==` and `!=`.

It may seem like an iterator you can only write to, not read from, is about as sensible as Barnstable Bear in Pogo who could write but not read what he wrote. But there are two very useful subclasses of `OutputIterator`:

- insert operators
- ostream iterators

Insert Iterators

Insert iterators let you "point" to some location in a container and insert elements. You do this with just dereferencing and assignment:

```
*iter = value;
```

This *inserts* the value in the place pointed to by the iterator. If you assign again, a new value will be inserted. Whether value goes before or after the previous value depends on what kind of insert operator you've created. Notice that you don't need to increment the iterator. You just keep assigning.

You create an insert iterator with one of the following:

- `back_inserter<container>` returns an `OutputIterator` pointing to the end of the container. Output to this iterator gets added to the end of the container, using the container's `push_back()` operation.
- `front_inserter<container>` returns an `OutputIterator` pointing to the front of the container. Output to this iterator gets added to the front of the container, using the container's `push_front()` operation.
- `inserter<container, iterator>` returns an `OutputIterator` pointing to the location pointed to by *`iterator`* of the container. Output to this iterator gets added to the container from that point forward, using the container's `insert()` operation.

An example of using the insert iterators appears when describing istream iterators.

Ostream OutputIterator

Ostream iterators let you "point" to an output stream and insert elements into it, i.e., write to the output stream.

We can construct an ostream iterator from a C++ output stream as follows:

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

The first line defines `outIter` to be an ostream iterator for integers. The `" "` means "put a space between each integer." If we'd said `"\n"` then `outIter` would put a newline between each integer. The second line uses the generic algorithm `copy()` to copy our vector `v` from beginning to end to `cout`. Note how much simpler this is than the equivalent `for` loop with `cout` and `<<`.

Istream InputIterator

Istream iterators for input streams work similarly to ostream iterators. Istream iterators are `InputIterators`. The following code fragment constructs an istream iterator that reads integers from `cin` and copies them into a vector `v`. We use a `back_inserter` to add the elements to the end of the vector, which could beempty:

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

The first argument to copy calls an istream iterator constructor that simply points to the input stream `cin`. The second argument calls a special constructor that creates a pointer to "the end of the input." What this actually means, especially for terminal input, depends on your operating system. So the above says "copy from the current item in the input sream to the end of the input stream into the container v."

ForwardIterator

`ForwardIterator` combines `InputIterator` and `OutputIterator`. You can use them to read and write to a container. They also support:

- saving and reusing

A trivial example of this is

```
iterSaved = iter;
iter++;
```

```
cout << "Previous element is " << (*iterSaved) << endl;
cout << "Current element is " << (*iter) << endl;
```

This will work if the iterators are `ForwardIterator`'s. Note that it can't work for `InputIterator`'s and `OutputIterator`'s, such as istream and ostream iterators. I/O streams such as standard input and output don't support backing up and starting over.

> **Note:** A saved iterator is only valid if the underlying container is not modified. If you insert elements or otherwise change the container, using a saved iterator will have undefined behavior.

`BidirectionalIterator`

If `iter` is a `BidirectionalIterator`, you can use:

- all `ForwardIterator` operations
- `--iter` and `iter--` to decrement it, i.e., advance the pointer to the previous element

`RandomAccessIterator`

If `iter1` and `iter2` are `RandomAccessIterator`'s, you can use:

- all `BidirectionalIterator` operations
- standard pointer arithmetic, i.e., `iter + n`, `iter - n`, `iter += n`, `iter -= n`, and `iter1 - iter2` (but **not** `iter1 + iter2`)
- all comparisons, i.e., `iter1 > iter2`, `iter1 < iter2`, `iter1 >= iter2`, and `iter1 <= iter2`

Since `BidirectionalIterator`'s support ++ and --, don't they support these operations too? The answer is that `RandomAccessIterator`'s support these operations **in constant time**. That is, you can jump N elements in the same time it takes to jump 1 element. So an STL `list` container can return a `BidirectionalIterator`, but not a `RandomAccessIterator`. Vectors and deques can return `RandomAccessIterator`'s.

Why Iterators Make Code General and Efficient

There are two rules for making container-based code general and efficient:

- Never pass containers into a function. Pass iterators instead.
- Never return containers. Return -- or pass -- iterators instead.

Generality

Suppose we wanted to define `product()` to multiply together the numbers in a container.

We can immediately reject any definition like this:

```
double product( vector<double> v ) ...
```

because it's

- not general; it only works for vectors
- inefficient; it copies the container

But we could define it like this:

```
template <class Container>
double product( const Container & container )
```

```
{
  Container::iterator i = container.begin();
  double prod = 1;

  while ( i != container.end() ) prod *= *i++;

  return prod;
}
```

This definition seems general. It works for any STL container, e.g.,

```
vector<double> nums;
...
return product( nums );
```

Unfortunately, it won't work with regular arrays, e.g.,

```
double nums[] = { 1.2, 3.0, 3.5, 2.8 };
return product( nums );
```

because there are no `begin()` or `end()` methods for regular C-style arrays. Furthermore, it doesn't let us calculate the product of a subrange of the container.

The following definition is clearly more general:

```
template <class Iter>
double product( Iter start, Iter stop )
{
  double prod = 1;

  while ( start != stop ) prod *= *start++;

  return prod;

}
```

This works fine with regular arrays:

```
double nums[] = { 1.2, 3.0, 3.5, 2.8 };

return product( nums, nums + 4 );
```

as well as with STL containers and subranges.

Efficiency

Both definitions of `product()` above are efficient. Neither copies the container involved. But consider a function -- let's call it `generate()` -- that has to generate multiple answers. There are two common situations:

- There's a container of objects and `generate()` is supposed to return certain ones, e.g., all the odd numbers in the container.
- There is no container. `generate()` is supposed to generate new objects, e.g., all the prime numbers less than N.

The obvious way to define `generate()` is like this:

```
template <class Container>
Container generate()
```

```
{
  Container temp;
  ...
  return temp;
}
```

We would call `generate()` like this:

```
vector<int> v = generate<vector< int > >( ... );
```

but this has two bad properties:

- It's ugly to write calls to `generate()`
- `generate()` creates a container which then has to be copied and returned

If the container already exists and `generate()` is returning a subset of the elements in it, e.g., all the odd numbers, then the thing to do is to

- pass `generate()` iterators pointing to the start and end, as usual, and
- have `generate()` return an iterator pointing to the first answer, if any, else the end iterator

The template for this kind of definition is simple:

```
template <class Iter >
Iter generate( Iter start, Iter stop )
{ Iter temp;
  ...set temp to first answer ...
  return temp;
}
```

The following loop would then get the answers and print or store them, as desired:

```
vector<int> nums;
...
vector<int>::iterator ans
  = generate( nums.begin(), nums.end() );

while ( ans != nums.end() ) {
  process ans
  ans = generate( ans, nums.end() );
}
```

If the container does not exist, you have to be a little smarter. In particular, both of the following are **very bad** (and won't even compile without some more work):

```
template <class Container>
Container & generate()
{
  Container temp;
  ...
  return temp;
}
```

```
template <class Container>
Container::iterator generate()
{
  Container temp;
  ...
```

```
  return temp.begin();
}
```

Both of these return references to `temp` which no longer exists when `generate()` exits.

The following is better

```
template <class Container>
Container * generate()
{
  Container *temp = new Container();
  ...
  return temp;
}
```

but now we're dealing with pointers and we have to guarantee that the container that `generate()` dynamically allocates gets deallocated by some other piece of code. Otherwise, we have a bad memory leak.

> Avoid code that allocates memory in one place and deallocates it in another.

The clever way to solve this problem is to do this:

```
template <class Iter>
void generate( Iter iter )
{
  ...
  *iter++ = ...;
  ...
}
```

In other words, don't have any container in `generate()` at all. Just pass `generate()` an iterator to store answers in. As before, what container to use (if any) is up to the calling code. Two ways we could call `generate()` are:

```
generate( back_inserter( v ) ); // stores answers in v

generate( ostream_iterator<type>( cout, "\n" ) ); //print answers
```

As before, by using iterators, we have made `generate()` far more useful and general and efficient than any version that takes or returns containers.

> **Warning**: `generate()` has to be a template function for this to work. If `generate()` is actually a member function of some class, then `generate()` has to be a **template member function**. Older compilers (and some current ones, alas) have poor support for template member functions.

Comments?  Let me know!