Iterators

The STL facilities make widespread use of iterators to mediate among the various algorithms and the sequences upon which they act. The name of an iterator type (or its prefix) indicates the category of iterators required for that type. In order of increasing power, the categories are summarized here as:

- Output. An output iterator *X* can only have a value *V* stored indirect on it, after which it *must* be incremented before the next store, as in (**X*++ = *V*), (**X* = *V*, ++*X*), or (**X*= *V*, *X*++).
- Input. An input iterator *X* can represent a singular value that indicates end of sequence. If an input iterator does not compare equal to its end-of-sequence value, it can have a value *V* accessed indirect on it any number of times, as in (*V* = **X*). To progress to the next value or end of sequence, you increment it, as in ++*X*, *X*++, or (*V* = **X*++). Once you increment any copy of an input iterator, none of the other copies can safely be compared, dereferenced, or incremented thereafter.
- Forward. A forward iterator *X* can take the place of an output iterator for writing or an input iterator for reading. You can, however, read (through *V* = **X*) what you just wrote (through **X* = *V*) through a forward iterator. You can also make multiple copies of a forward iterator, each of which can be dereferenced and incremented independently.
- Bidirectional. A bidirectional iterator *X* can take the place of a forward iterator. You can, however, also decrement a bidirectional iterator, as in --*X*, *X*--, or (*V* = **X*--).
- Random access. A random-access iterator *X* can take the place of a bidirectional iterator. You can also perform much the same integer arithmetic on a random-access iterator that you can on an object pointer. For *N*, an integer object, you can write *x*[*N*], *x* + *N*, *x* - *N*, and *N* + *X*.

Note that an object pointer can take the place of a random-access iterator or any other iterator. All iterators can be assigned or copied. They are assumed to be lightweight objects and are often passed and returned by value, not by reference. Note also that none of the operations previously described can throw an exception when performed on a valid iterator.
The hierarchy of iterator categories can be summarized by showing three sequences. For write-only access to a sequence, you can use any of:

```
output iterator
   -> forward iterator
   -> bidirectional iterator
   -> random-access iterator
```

The right arrow means "can be replaced by." Any algorithm that calls for an output iterator should work nicely with a forward iterator, for example, but *not* the other way around.
For read-only access to a sequence, you can use any of:

```
input iterator
   -> forward iterator
   -> bidirectional iterator
   -> random-access iterator
```

An input iterator is the weakest of all categories, in this case.
Finally, for read/write access to a sequence, you can use any of:

```
forward iterator
   -> bidirectional iterator
   -> random-access iterator
```

An object pointer can always serve as a random-access iterator, so it can serve as any category of iterator if it supports the proper read/write access to the sequence it designates.
An iterator **Iterator** other than an object pointer must also define the member types required by the specialization **iterator_traits<Iterator>**. Note that these requirements can be met by deriving **Iterator** from the public base class iterator.
This "algebra" of iterators is fundamental to practically everything else in the Standard Template Library. It is important to understand the promises and limitations of each iterator category to see how iterators are used by containers and algorithms in the STL.

Note

You can also use for each, in to iterate over STL collections.