STL Iterators. The Standard Template Library provides a rich variety of containers:

`vector`
`list`
`deque`
`stack`
`queue`
`priority_gueue`
`map` and `multimap`
`set` and `multiset`

The elements of a `vector<T>` can be accessed using an index and the subscript operator, but this is not true for other containers such as `list<T>`. Stated differently, using the subscript operator and indices is not a generic way to access an element in a container.

In order for a Standard Template Library algorithm to work on *any* STL container, some truly generic means of accessing the elements in a container is required. For this purpose, STL provides objects called **iterators** that can "point at" an element, access the value stored there, and move from one element to another.

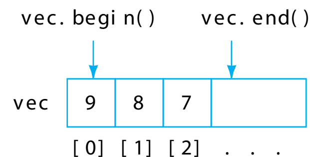
Each STL container provides its own group of iterator types and (at least) two methods that return iterators:

- `begin()`: returns an iterator positioned at the first element in the container
- `end()`: returns an iterator positioned at the element following the last value in the container

To illustrate, suppose the following statements are executed:

```
vector<int> vec;        // empty vector
vec.push_back(9);       // append 9
vec.push_back(8);       // append 8
vec.push_back(7);       // append 7
```

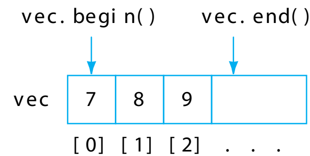
The expressions `vec.begin()` and `vec.end()` produce iterators that can be visualized as follows:



A function like `sort()` then uses these iterators

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

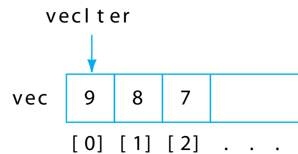
to access (and rearrange) the values that lie between the two iterators:



Defining Iterator Objects. Each STL container also declares an `iterator` type that can be used to define iterator objects. To ensure that the right type is used, the identifier `iterator` must be preceded by the name of its container and the **scope operator** (`::`). For example, the statements:

```
vector<double> vec;        // empty vector
vec.push_back(9.0);        // append 9
vec.push_back(8.0);        // append 8
vec.push_back(7.0);        // append 7
vector<double>::iterator vecIter = vec.begin();
```

define `vecIter` as an `iterator` object positioned at the first element of `vec`:



Iterator Operations. Among the operators that can be applied to an iterator are:

- The **increment operator** (`++`). Applied as a prefix or postfix operator to an iterator, `++` moves the iterator from its current position to the next element of the container.
- The **decrement operator** (`--`). Applied as a prefix or postfix operator to an iterator, `--` moves the iterator from its current position to the previous element of the container.
- The **dereferencing operator** (`*`). Applied as a prefix operator to an iterator, `*` accesses the value stored at the position to which the iterator points.

Other operators that can be applied to `vector<T>::iterator` objects include assignment (`=`), equality comparisons (`==` and `!=`), addition (`+`), subtraction (`-`), the corresponding shortcuts (`+=`, `-=`), and the subscript operator (`[]`).

The increment, decrement, and arithmetic operators can be used to *change the position* of an iterator variable. The dereferencing operator can be used to *access the value* at the iterator's current position.

The following example illustrates the use of these operators. It is an alternative way to write the function template `print()` of Figure 10-5 of the text, using an iterator and two of these operators:

```
template <typename T>
void print(ostream& out, const vector<T>& theVector)
{
   vector<T>::iterator vecIter = theVector.begin();

   while(vecIter != theVector.end())
   {
      out << *vecIter << ' ';
      vecIter++;
   }
}
```

On each pass through the `while` loop, the `*` operator is used to access the value at the iterator's current position, and the `++` operator then advances the iterator to the next element.

Unlike the function in Figure 10-5 of the text (which requires the subscript operator), this function template can be made into a generic `print()` *algorithm template* that allows the values in almost *any* STL container to be output to an `ostream`:

```
template <typename Container>
void print(ostream & out, const Container & theContainer)
{
   Container::iterator conIter = theContainer.begin();

   while(conIter != theContainer.end())
   {
      out << *conIter << ' ';
      conIter++;
   }
}
```

In summary, iterators provide a mechanism for moving among and accessing the values in a container that is independent of any particular container (i.e., a *generic* mechanism). Understanding the preceding introduction to iterators should enable you to use the STL containers and algorithms effectively.