An Introductory STL tutorial

Introduction

STL provides a template based set of collection classes, and methods for working on those collections. The collection classes give the developer access to fast and efficient collections. While the methods, which are known as the algorithms, provide template based collection manipulations functions.

The benefits of STL include

- Type safe collections
- Ease of use

Templates

If you are already familiar with templates then skip to the next section. Otherwise read this section for a brief tutorial on templates. A template can be thought of as a macro with type checking. For example to declare a template we would do the following:

```
template < class T >
class Value
{
    T _value;
    public:
    Value ( T value ) { _value = value; }

    T getValue ();

    void setValue ( T value );
};

template < class T >
```

```
T Value<T>::getValue () { return _value; }

template < class T >
void Value<T>::setValue ( T value ) { _value = value; }
```

This example declares a class **Value**, which stores a parameterized value, *_value*, of type **T**. After the keyword template, in the angled brackets, is a list of parameters. The list tells the template what types will be used in the template. A good analogy for the template parameter list is the parameter list for a class constructor. Like a constructor, the number of arguments for the template can be from one to many.

Methods for a template that are declared outside the class definition require the template keyword, as shown above. To use the Value class to declare an array of floats we would do:

```
Value<float> values[10];   // array of values of type float
```

This declares an array of values, the angled brackets tells us that Value will store its value as a float.

If we wanted to declare a list to work with our template based Value class we could do the following:

```
Template < class T  >
class ValueList
{
    Value<T> * _nodes.
public:
    ValueList ( int noElements )
    {
        _nodes = new Node<T>[noElements];
    }

    virtual ~ValueList ()
    {
        delete [] _nodes;
    }
};
```

Here we have declared a template-based class that stores a variable sized list of values.

STL Collection Types

Each STL collection type has its own template parameters, which will be discussed later. What type of collection you use is up to your needs and tastes. From past experience, the vector and map classes are the most useful. The vector class is ideal for simple and complex collection types, while the map class is used when an associative type of collection is needed. The deque collection is excellent for use in systems that have queued based processing, such as a message based system.

vector

A collection of elements of type T.

list

A collection of elements of type T. The collection is stored as a bi-directional linked list of elements, each containing a member of type T.

To include the class definition use:

Hide Copy Code

```
#include <list>
```

deque

A collection of varying length of elements of type **T**. The sequence is represented in a way that permits insertion and removal of an element at either end with a single element copy, and supports insertion and removal anywhere in the sequence, but a sequence copy follows each operation.

To include the class definition use:

Hide Copy Code

```
#include <deque>
```

map

A collection for a varying length sequence of elements of type **pair<const Key, T>**. The first element of each pair is the sort key and the second is its associated value. The sequence is represented in a way that permits lookup, insertion, and removal of an arbitrary element. The key can be a simple as a number or string or as complex a key class. The key class must support normal comparison operations so that the collection can be sorted or searched.

To include the class definition use:

Hide Copy Code

```
#include <map>
```

set

A collection that controls a varying length sequence of elements of type **const Key**. Each element serves as both a sort key and a value. The sequence is represented in a way that permits lookup, insertion, and removal of an arbitrary element.

To include the class definition use:

Hide Copy Code

```
#include <set>
```

multimap

A collection of a varying length sequence of elements of type **pair<const Key, T>**. The first element of each pair is the sort key and the second is its associated value. The sequence is represented in a way that permits lookup, insertion, and removal of an arbitrary element.

To include the class definition use:

Hide Copy Code

```
#include <map>
```

multiset

A collection of a varying-length sequence of elements of type **const Key**. Each element serves as both a sort key and a value. The sequence is represented in a way that permits lookup, insertion, and removal of an arbitrary element.

To include the class definition use:

Hide Copy Code

```
#include <set>
```

STL Strings

STL strings support both ascii and unicode character strings.

string

A string is a collection of ascii characters that supports both insertion and removal.

To include the string class definitions use:

Hide Copy Code

```
#include <string>
```

wstring

A wstring is a collection of wide characters that it supports both insertion and removal. In MFC the string class is **CString**, which provides a **Format** and other methods to manipulate the string. **CString** has the advantage of providing methods such as **Format**, **TrimLeft**, **TrimRight** and **LoadString**. It is easy to provide a string-based class that contains these methods.

To include the wstring class definitions use:

Hide Copy Code

```
#include <xstring>
```

STL Streams

Streams provide the developer with classes that can output to a container variable types of stream elements.

stringstream

A string stream that supports insertions of elements, and elements are inserted via the overloaded operator **<<**. The method **str()** gives a reference back to the underlying string, and the **c_str()** can be used to get a constant pointer to the string buffer.

wstringstream

A wstring stream that supports insertions of elements, and elements are inserted via the overloaded operator **<<**. The method **str()** gives a reference back to the underlying string, and the **c_str()** can be used to get a constant pointer to the string buffer.

To use a string stream we would do the follwing:

Hide Copy Code

```
stringstream strStr;
for ( long i=0; i< 10; i++ )
    strStr << "Element " << i << endl;
```

To include the string class definitions use:

Hide Copy Code

```
#include <strstring>
```

STL Collections General Class Methods

empty

Determines if the collection is empty

size

Determines the number of elements in the collection

begin

Returns a forward iterator pointing to the start of the collection. It is commonly used to iterate through a collection.

end

Returns a forward iterator pointing to one past the end of the collection. It is commonly used to test if an iterator is valid or in looping over a collection.

rbegin

Returns a backward iterator pointing to the end of the collection It is commonly used to iterate backward through a collection.

rend

Returns a backward iterator pointing to one before the start of the collection. It is commonly used to test if an iterator is valid or in looping over a collection.

clear

Erases all elements in a collection. If your collection contains pointers the elements must be deleted manually.

erase

Erase an element or range of elements from a collection. To erase simply call erase with an iterator pointing to the element or a pair of iterator show the range of elements to erase. Also, vector supports erasing based on array index.

Standard Out and Input

STL also includes classes for printing to the standard output streams. Like standard C++ the classes are cout and wcout. To use them in a console application include the file iostream. As an example:

Hide Copy Code

```
#include <iostream>

void main ()
{
    char ch;

    cin >> ch;
    cout << &#8220;This is the output terminal for STL&#8221; << endl;
}
```

Vector and Deque add and remove methods

We want to look briefly at adding/removing elements from the vector and deque collections. These collections are represented as an array, and to add an element we use the push methods with back or front depending on if we are adding at the front (start) or back (end) of an array.

The general methods are:

push_back	Add element to end of collection.
push_front	Add an element to start of a collection.
back	Get a reference to element at end of collection
front	Get a reference to element at end of collection
pop_back Remove	element at end of collection
pop_front	Remove element at end of collection

As an example, suppose we want to build a message processing system based on a message class:

Hide Copy Code

```
Class Msg
{
    int _type;
    int _priority;
    string _message;

    public:

    Msg ( int type, int priority, string & msg )
    { _type = type; priority = priority; _msg = msg; }

    Msg ( int type, int priority, char * msg )
    { _type = type; priority = priority; _msg = msg; }
```

```
    int getType () { return _type; }
    int getPriority () { reutrn _priority; }
    string & getMsg () {return _msg; }
};
```

To store the messages we would need a first in first out based collection, such as deque: typedef deque<Msg> MsgList;

To send a message we might do the following:

Hide Copy Code

```
Msg message( 0, 0, "My Message" );
msgList.push_back(msg);
```

And to process message we could would do the following:

Hide Copy Code

```
void process_msgs ()
{
    bool done = false;
    while ( !done )
    {
        // if no messages stop
        if ( msgList.size() == 0 )
        { done == true; continue;}
        // get msg and process
        Msg & msg = msgList.front();

        switch ( msg.getType() )
        { // process messages }

        // remove msg from que
        msgList.pop_front();
    }
}
```

With just a few lines of code we have created a general messaging system, if we wanted an entire system we could create a simple COM server that exposed a mail interface, and that stored the messages using a message list.

Operator []

For vector, map, deque, string and wstring collections, elements are normally added using:

Hide Copy Code

```
operator []
```

Access an element at a position, and for map, string and wstring supports insert of element.

A simple example of using this operator would be to decalre a list using map:

Hide Copy Code

```
typedef map<int, string> StringList
StringList strings;
stringstream strStr
for ( long i=0; i<10; i++ )
{
    stringstream strStr;
    strStr << "String " << i;
    strings[i] = strStr.str();
}
for ( long i=0; i<10; i++ )
{
    string str = strings[5];
    cout << str.c_str() << endl;
```

```
}
```

We have created a map, whose key is an integer, and that stores strings.

Iterators

Iterators support the access of elements in a collection. They are used throughout the STL to access and list elements in a container. The iterators for a particular collection are defined in the collection class definition. Below we list three types of iterators, iterator, reverse_iterator, and random access. Random access iterators are simply iterators that can go both forward and backward through a collection with any step value. For example using vector we could do the following:

Hide Copy Code

```
vector<int>    myVec;
vector<int>::iterator    first, last;
for ( long i=0; i<10; i++ )
    myVec.push_back(i);
first = myVec.begin();
last = myVec.begin() + 5;
if ( last >= myVec.end() )
    return;
myVec.erase( first, last );
```

This code will erase the first five elements of the vector. Note, we are setting the last iterator to one past the last element we of interest, and we test this element against the return value of end (which give an iterator one past the last valid item in a collection). Always remember when using STL, to mark the end of an operation use an iterator that points to the next element after the last valid element in the operation.

The three types of iterators are:

iterator (forward iterator through collection)

Allows a collection to be traversed in the forward direction. To use the iterator

Hide Copy Code

```
for ( iterator element = begin(); element < end(); element++ )
    t = (*element);
```

Forward iterators support the following operations:

Hide Copy Code

```
a++, ++a, *a, a = b, a == b, a != b
```

reverse_iterator (reverse iterator through collection)

Allows a collection to be traversed in the reverse direction. As an example:

Hide Copy Code

```
for ( reverse_iterator element = rbegin(); element < rend(); element++ )
    t = (*element);
```

All of the collections support forward iterators. Reverse iterators support the following operations:

Hide Copy Code

```
a++, ++a, *a, a = b, a == b, a != b
```

random access (used by vector declared as forward and reverse_iterator)

Allows a collection to be traversed in either direction, and with any step value. An example would be:

Hide Copy Code

```
for ( iterator element = begin(); element < end(); element+=2 )
    t = (*element);
```

The vector collection supports random access iterators. Iterators are the most used type of access to the collections of STL, and they are also used to remove elements from collections. Look at the following:

Hide Copy Code

```
iterator element = begin(); erase(element);
```

This will set an iterator to the first element of the collection and then remove it from the collection. If we were using a vector we could do the following

Hide Copy Code

```
iterator firstElement = begin();
          iterator lastElement = begin() + 5;
          erase(firstElement,lastElement);<pre>
          <p>to remove the first five elements of a collection.
          Random access iterators support the following:

          <pre>a++, ++a, a--, --a, a += n, a -= n, a &#8211; n, a + n*a, a[n],
a = b, a == b, a != b, a < b, a <= b, a > b, a >= b
```

It is important to remember, when you get an iterator to a collection do not modify the collection and then expect to use the iterator. Once a collection has been modified an iterator in most cases will become invalid.

Declaring collections

Each collection uses it's template paramters to determine what elements the collection will store. Shown below is a list of the collections we are discussing and beside each is the template pamaters. In the parameters *T* denotes the element type to store in the collection, *A* denotes the allocator (which allocates elements), *Key* denotes the key for the element, and Pred denotes how the collection will be sorted.

Hide Copy Code

```
template < class T, class A = allocator<T> > class vector
template < class T, class A = allocator<T> > class list
template < class T, class A = allocator<T> > class deque
template <class Key, class T, class Pred = less<Key>, class A = allocator<T> > class map
template <class Key, class Pred = less<Key>, class A = allocator<Key> > class set
template <class Key, class T, class Pred = less<Key>, class A = allocator<T> > class multimap
template <class Key, class Pred = less<Key>, class A = allocator<Key> > class multiset
```

This list looks somewhat daunting but it provides a quick reference for the collections. In most cases you will use the default arguments and your only concern will be what you are storing and how it is stored. T refers to what you will store, and for collections that support a key; Key shows how the elements will be associated.

From previous experience the vector, map and deque classes are the most often used so we can use them as an example for declaring a collection:

Using typedef to declare the collection:

Hide Copy Code

```
typedef vector<int> myVector
typedef map< string, int > myMap
typedef deque< string > myQue
```

The first declaration declares a vector of integers, the second declares a collection of integers, which have a key of type string, and the last declares a queue (or stack) of strings.

Another way to declare a collection is to derive a collection from an STL collection as in the following:

Hide Copy Code

```
class myVector : public vector<int> {};
```

Either method is useful, it a matter of preference. Another important consideration is declaring the iterators supported by the collection as separate types. If we use the above example we would declare:

Hide Copy Code

```
typedef myVector::iterator vectorIterator
```

```
typedef myVector::reverse_iterator revVectorIterator
```

This gives the user of the collection direct access to the iterator without being forced to use the following syntax:

Hide Copy Code

```
myVector coll;
for ( myVector::iterator element = coll.begin(); element < coll.end(); element++ )
```

The resolution operator can be cumbersome.

Algorithms

Up to this point we have discussed how to use STL at a bare minimum, now we need to delve into the most important part of the collections. How do we manipulate a collection? For example, if we had list of strings, what would we need to sort the list in alphabetical order, or if we wanted to search a collection for a set of elements that matched a given criterion. This is where STL algorithms are used. In your visual studio installation, under include directory, you will find an include file, algorithm. In algorithm a set of template based functions are declared. These functions can be used to manipulate STL collections. The functions can be categorized in the following: sequence, sorting and numeric. Using these categories, we can list all of the methods of algorithms:

Sequence

count, count_if, find, find_if, adjacent_find, for_each, mismatch, equal, search copy, copy_backward, swap, iter_swap, swap_ranges, fill, fill_n, generate, generate_n, replace, replace_if, transform, remove, remove_if, remove_copy, remove_copy_if, unique, unique_copy, reverse, reverse_copy, rotate, rotate_copy, random_shuffle, partition, stable_partition

Sorting

Sort, stable_sort, partial_sort, partial_sort_copy, nth_element, binary_search, lower_bound, upper_bound, equal_range, merge, inplace_merge, includes, set_union, set_intersection, set_difference, set_symmetric_difference, make_heap, push_heap, pop_heap, sort_heap, min, max, min_element, max_element, lexographical_compare, next_permutation, prev_permutation

Numeric

Accumulate, inner_product, partial_sum, adjacent_difference

Since this is an extensive list, we will only examine a few of the methods in the algorithms. It is very important to note that the methods here are templated so we are not required to use the STL containers to use the methods. For example, we could have a list of ints and to sort this list then we could do:

Hide Copy Code

```
#include <vector>
#include <algorithm>
#include <iostream>

vector<int>              myVec;
vector<int>::iterator    item;
ostream_iterator<int>    out(cout," ");
// generate array
for ( long i=0; i<10; i++ )
    myVec.push_back(i);
// shuffle the array
random_shuffle( myVec.begin(), myVec.end() );
copy( myVec.begin(), myVec.end(), out );
// sort the array in ascending order
sort( myVec.begin(), myVec.end() );
copy( myVec.begin(), myVec.end(), out );
```

This example shows how declare the vector and then sort it, using STL containers. We could do the same without using

containers:

Hide Copy Code

```
ostream_iterator<int>  out(cout," ");
// generate array (note: one extra element, end in STL is one element past last valid)
int myVec[11];
for ( long i=0; i<10; i++ )
    myVec[i] = i;
int * begin = &myVec[0];
int * end = &myVec[10];
// shuffle the array
random_shuffle( begin, end );
copy( begin, end, out );
// sort the array in ascending order
sort( begin, end );
copy( begin, end, out );
```

How you use the algorithms is largely up to you, but they provide a rich set of methods for manipulating containers.

Multithreading Issues

STL is not thread protected, so you must provide locks on your collections if they will be used in multithreaded environment. The standard locking mechanisms of Mutexes, Semaphores and Critical Sections can be used. One simple mechanism for providing locking is to declare a lock class. In this class the constructor creates the lock, and the destructor destroys the lock. Then provide **lock()** and **unlock()** methods. For example:

Hide Copy Code

```
class Lock
{
public:
    HANDLE    _hMutex;            // used to lock/unlock object

    Lock() : _hMutex(NULL)
    { _hMutex = ::CreateMutex( NULL, false,NULL) ); }

    virtual ~Lock() { ::CloseHandle( _hMutex ); }

    bool lock ()
    {
        if ( _hMutex == NULL )
            return false;
        WaitForSingleObject( _hMutex, INFINITE );
        return true;
    }

    void unlock () { ReleaseMutex(_hMutex);      }

};
```

Then declare a class that is derived from one of the STL collections, and in the class override the access methods to the collection that might cause an insertion or deletion of an element. For example a general vector class would be:

Hide Copy Code

```
template <class T>
class LockVector : vector<T>, Lock
{
public:
    LockVector () : vector<T>(), Lock()
    {}
    virtual LockVector ()
    {}

    void insert ( T & obj )
    {
```

```
        if ( !lock())
            return;
        vector<T>::push_back (obj);
        unlock();
    }
};
```

Conclusion

Hopefully I've given you a good tutorial on how to use STL. If not then please try some of the web sites listed below or drop by your local bookstore or amazon.com and purchase one of the many books on the subject. I believe STL can provide many benefits and I hope you will to.