Standard Template Library

Introduction

Built-into the standard C++ language, there are hundreds of useful functions that are provided as part of the C++ programming environment. These functions can provide all sorts of useful functionality, and can save the programmer many hours of work that would have been required to code the functions had they not been provided.

No significant program is written in just a bare programming language. First a set of supporting libraries are developed. These then form the basis for further work.

A Standard Library
In a general computer programming sense is defined as a set of standardized subroutines, macro definitions, global variables, class definitions, templates, and/or other commonly-used programming objects to extend some programming language beyond what the bare language makes available.

C++ Standard Library

The standard C++ library is a collection of functions, constants, classes and objects that extends the C++ language providing basic functionality to interact with the operating system and some standard classes, objects and algorithms that may be commonly needed. It is often divided in three main groups:

- C standard library
- Iostream Library
- STL (Standard Template Library)

The iostream library and STL library are generally defined together as standard C++ header files.
This description is a little crude as some files, such as complex (describes complex numbers) are not part of either.

The C++ Standard Library

- The standard library contains classes that are considered so useful they should be made a “standard” part of C++.
- The standard library includes at least 50 header files, those specified by the standard, although many compilers add one or two files to them. They typically include the 32 C++ header files plus 18 header files that define the facilities of the original C library.
- The components of the C++ standard library are contained within the *std::* namespace.

The STL

The STL represents a powerful collection of programming work that has been done before, and done well.

It is used by games programmers professionally so trying to familiarise your self with it is a good option.

The STL, amongst other things, provides a group of containers, algorithms and iterators.

Containers
Help you store and access collections of values of the same type. If you are thinking “well isn’t that what arrays do?” then yes they do, the STL containers however provide much more flexibility and power than a simple array.

The STL defines many different container types, each one is designed and works in a way to meet different of needs.

Algorithms
The algorithms defined in the STL work with its containers. The algorithms are common functions that game programmers find themselves repeatedly applying to groups of values.
These algorithms include searching, sorting, copying, merging, inserting and removing container elements.
The algorithms can be used on different container types.

Iterators
These are objects than identify elements in containers and can be manipulated to move amongst them.
They great at iterating through containers and are required by the STL algorithms.

Example: Vectors

The vector class is an example of one of the containers provided by the STL. It meets the general description of a dynamic array, it can shrink and grow in size as needed.
Vector also defines members functions to manipulate vector elements.
To summarise the vector class has all the base functionality of an array with lots of added extras.

Added the Vectors header
To use vectors, as with all things, we need to include the file that contains its definition.
For vectors we type:

```
#include <vector>
```

All STL components live within the std namespace: std::

Declaring Vectors
We declare an instance of vector by using the following syntax:

```
std::vector</* data type */> /* identifier */;
```

or if we are using namespace std:

```
vector</* data type */> /* identifier */;
```

Data type can be any valid data type including user defined types such as classes.
Identifier is a valid name that you as the programmer have given it.

There are additional ways to declare a vector:

```
std::vector</* data type */> /* identifier */(10);
```

example:
```
std::vector<string> playernames(10);
```

This sets the vectors starting size to 10.

It is also possible to provide starting values:

```
std::vector<string> playernames(10, "nobody");
```

This initialises all the strings stored in the vector to “nobody”.

```
std::vector<string> playernames(thePlayers);
```

Member Functions:

push_back()
Inserts element after the last, adding one to the end of the vector

```
playernames.push_back("Jeff");
playernames.push_back("Bob");
playernames.push_back("Tamara");
```

If we declare our vector like so:

```
std::vector<string> playernames;
```

Then the above lines sets `playernames[0]` to `"Jeff"`, `playernames[1]` to `"Bob"` and `playernames[2]` to `"Tamara"`.
The push_back() member function accepts the datatypes it was declared to hold, therefore if you declared it hold a user defined type, such as a class, it will hold instances of your class.

size()
We can use the size() function to tell us how bug the vector is, or more accurately how many elements it holds.

```
playernames.size()
```

This can be used in a variety of ways, we can print out the size():

```
cout << “Number of Players = “ << playernames.size();
```

Or we can use it as part of a for loop:

```
for(int i = 0; i < playernames.size(); i++)
```

Indexing
Just like arrays it is possible to index vectors by using the subscripting operator.
We can use these in a loop to print all element in turn, or if we are storing sprites in our vector to render them.
We can also use the subscripting operator to assign values as with an array.

```
playernames[0] = “Noob”;

for(int i = 0; i < playernames.size(); i++)
{
     cout << “Player: “ << i << “ = “ << playernames[i];
}
```

*Dangerous Practice: Although vectors are dynamic you can’t increase a vectors size by applying the subscripting operator.

The following is dangerous because we do not have an element 0!

```
std::vector<string> playernames;
playernames[0] = “Noob”;
```

Neither can you access a nonexistent element position, well not without resulting errors!
To add new elements use the push_back() member function.

Calling member functions and data of an Element
Just like arrays we can call the member functions and data of the type stored in the vector.
The following for instance calls the string function size, returns the size (number of characters) of the string:

```
playernames[0].size();
```

If you are storing any classes you are then able, as with arrays, to call any of your own classes member functions.

pop_back()
This member function deletes the last element of a vector and reduces the vector size by one.

```
std::vector<string> playernames;

playernames.push_back("Jeff");
playernames.push_back("Bob");
playernames.push_back("Tamara");
```

If we declare out vector like so then by using the pop back member function:

```
playernames.pop_back();
```

We delete “Tamara” as that was the last name added.

clear()
This member function removes all members from the vector, it effectively deletes everything stored in the vector and sets its size to one.

```
playernames.clear()
```

empty()
The empty member function simply checks to see if the vector is equal to 0 – if it is empty or not. If it is empty then true is returned.

```
playernames.empty()
```

Iterators

Iterators are the key to realising the full potential of containers. Iterators, as there name suggests, allow you to iterate through a sequence container. Many important parts of the STL also require iterators, many member functions and algorithms taking iterators as arguments.

Declaring Iterators
Declaring an iterator is very similar to declaring an instance of the vector class:

```
std::vector<string>::iterator      myIterator;
```

It is important to note that your iterator needs to be declared to contain the same data type as your vector. I have been using strings in my example so my vector iterator has string declared as its data type.

What are Iterators
They are essentially values that indentify a particular element in a container. With an iterator you can access the value of the element, you can also change value with the right iterator.

Iterators move among elements via familiar arithmetic operations.

Iterators act as pointers to elements, enabling us to perform operations on what there are pointing to.

const Iterators
It is possible to have iterator constants:

```
std::vector<string>::const_iterator          constIter;
```

A constant iterator is in many respects similar to a normal iterator, only as with all constants its value can't change. This means that the constant integrator can move around the vector it was declared for, it however be used to alter the value of any elements stored.

You may use a constant iterator to be explicit about your use, it tells other programmers you don't intend to alter any elements.
It is also safer, there is no way you can accidentally change a container element.

Looping through a vector
The following code loops through the contents of a vector from start to finish, displaying all the players names:

```
cout << "Player Names:\n";

for(constIter = playernames.begin(); constIter != 
playernames.end(); ++constIter)
{
     cout << *constIter << endl;
}
```

Very similar to using a for loop to iterate through an array, but rather than needing to know the size of the array we can us the end() member function, and we set our iterator to the beginning of our vector using the begin() member function, updating the interator instead of an integer after every pass of the loop body.

Deferencing an iterator: In the above code when we printed out we used `*constIter` this is using the deferencing operator to access the value to what the iterator refers. Note that we still can't change the value if it is a constant iterator.
Effectively what we are doing when we use the dereference operator is saying "treat this as the thing that the iterator references, not as the iterator its self"

Altering a iterator
In the above code we use the increment (++) operator on the iterator, although it is possible to perform other mathematical operations on operators to move them around a container, generally we find we simply need to increment or decrement.

Using iterators to change values

Although we can't use constant iterators to change values, there is nothing stopping us from using a standard iterator to do this.
To alter what is stored in an element using an iterator we use the dereferencing operator:

```
// set the iterator to the first element
myIterator = playernames.begin();
// change it to a new value
*myIterator = "Rhubarb";
```

By using the dereference operator in this way we are effectively saying assign the string "`Rhubarb`" to the element that `myIterator` references.
It does not change the iterator only what is stored at that location, in this case that location is the first element in the vector.

Accessing member functions of vector elements
We have two ways of accessing the member functions of data types stored in the vector elements:

```
(*myIterator).size();
```

or

```
*myIterator -> size();
```

I would tend to go for using the member selection (->) operator as it looks nicer and is more readable, this is known as syntactic sugar. To the computer they mean the same thing.

insert()
There are a few forms of insert, the following inserts a new member into the vector before the first parameter – which is an interator.

```
playernames.insert(playernames.begin(), "Rachael");
```

Inserting an element this way causes the other elements to "move up" by one.

This version of insert returns an iterator to the newly inserted element.

It is important to note that if you have any iterators set up and then call insert, you invalidate your iterators as they now point to different places, effectively one down from where they did originally.

erase()
Erase works in much the same way as insert, but rather than inserting an element at the specified position it deletes one:

```
Playernames.erase(playernames.begin()+1);
```

The code above will delete the second element. We add one to the iterator passed as an argument.
As a result of the deletion all the elements after the deletion will effectively "move down" by one.

An iterator to element after the one deleted is returned.

Again it is important to note that if you have any iterators set up and then call insert, you invalidate your iterators as they now point to different places, effectively one up from where they did originally.

Algorithms

The STL defines a set of algorithms that allow you to manipulate elements in containers through iterators.
There are algorithms that exist for most common tasks including searching, sorting, copying etc. By using these algorithms, many of which are very efficient, for mundane work to the STL you are able to get with the process of game programming.
These algorithms also conform to the idea of generic programming, the same algorithms can be used with different container elements.

include
To use the STL algorithms we need to include the algorithm header:

```
#include <algorithm>
```

Again, as mentioned earlier, the STL components live within the std namespace.

random_shuffle()
As with generating random numbers it is best to seed the random number generator:

```
srand(time(NULL));
```

We can then call the random_shuffle() function:

```
random_shuffle(playernames.begin(), playernames.end()):
```

To use this function you need to supply, as iterators, the starting and ending points of the sequence to shuffle. This can be the whole sequence, as the shown above, or part of it.

sort()
The sort() algorithm sorts the elements of a sequence in ascending order, again as with random_shuffle() we have to supply, as iterators, the first and last elements of the sequence to be sorted.
The elements are sorted in ascending order, in terms of value.

merge()
This member function combines two sorted sequences in to another sorted sequence.
To use this algorithm we need to pass the beginning and end points of the two sequences to be merged along with the beginning element of the vector that is to hold the merged sequences.

```
std::vector(int)    vScoresOne(5);
std::vector(int)    vScoresTwo(5);
std::vector(int)    vTheScores(10);

merge(vScoresOne.begin(), vScoresOne.end(),
     vScoresTwo.begin(), vScoresTwo.end(),
     vTheScores.begin());
```

In the above code we merge two vector sequences into a third vector sequence.

Note: The container you specify to hold the results of merge must be large enough to accommodate the merged sequence. *Merge() does not increase the vectors size!*

STL – Funky Stuff

The STL algorithms are that versatile that they can work with containers defined outside of the STL. They just have to meet certain pre-requisites. The string data type defined in <string> is an example of this. Although these strings are not part of the STL you are able to use appropriate STL algorithms on them:

```
std::string sName = "Jeff Noob";
random_shuffle(sName.begin(), sName.end());
```

Vector Performance

STL containers, which includes vectors, offer game programmers a variety of sophisticated and ready made ways for them to manipulate data. Such levels of sophistication of come with an associated price tag called performance. Performance is often at the foremost of game programmer's thoughts, well when you are dealing with high end games anyway.
You, and indeed other game programmers, need not worry the STL containers and there associated algorithms are incredibly efficient. Each container does have its own associated strengths and weaknesses so as good programmer you should learn the strengths and weaknesses of each one.

A will mention that as well as good performance the STL containers are robust, probably much more so that your code, having been refined through trial, error and revision over many years.

Vector growth
Although vectors grow dynamically as needed every vector has a specific size, when a new element is added that pushes the vector beyond its current size (or memory limit) the computer reallocates memory. This reallocation may result in all of the vector elements being copied into new memory. This can cause slow down.

If you are dealing with small data, or reallocating when nothing much is happening in the game (in terms of resource usage) then it probably isn't an issue, if however the action is coming thick and fast and you suddenly need to reallocate a vector sequence holding some large classes it probably isn't going to help.

capacity()
The capacity of a vector is simply the number of elements that a vector can hold before it needs to reallocate memory.
It is important to distinguish between size and capacity here: size() is the number of elements the vector is currently holding, capacity() is how many more it can hold before reallocation.

```
// 10 element vector to hold scores
// intialise the elements to 0
std::vector(int)    vTheScores(10,0);
// print the size and capacity
cout << "Vector size = " << vTheScores.size();
cout << "Vector capacity = " << vTheScores.capacity();

// add an element
vTheScores.push_back(0);
// print the new size and capacity
cout << "Vector size is now = " << vTheScores.size();
cout << "Vector capacity is now = " <<
vTheScores.capacity();
```

This code should demonstrate the difference between size and capacity. It also demonstrates what happens to both when you increase the size of you vector – the memory

reserve()
The reserve member function increases the capacity of a vector by the number supplied as an argument.
It has effect of giving you control of when additional memory reallocation occurs.

```
 // print the capacity
cout << "Vector capacity = " << vTheScores.capacity();
```

```
// reserve some more memory
vTheScores.reserve(10);

// print the new capacity
cout << "Vector capacity is now = " << vTheScores.size();
```

Insertion and Deletion
Adding or removing an element from the end of a vector is a very easy and efficient task to do. However adding or removing from any other point using insert() or erase(), for example, can have greater overheads especially if you have a vector storing a large sequence of data types that require a large amount of memory.
With the STL list the overhead for insertions and deletions is reduced. This demonstrates that there are different solutions that fit different problems. Investigate the STL and choose the best container for the problem at hand.

Notes

Those using older compilers such as Microsoft Visual C++ 6.0 may get some warnings when using the STL. This is because they don't implement it very well.

Resources

Paul Johnson provided supplementary lectures on the STL and even if you do not wish to attend these lectures then his lectures are available for download, although I have not personally looked at the lectures there quite a few and so cover a lot more than this brief introduction could every manage.

See Programming 1 week 7 for an introduction to namespaces.

Links:
http://www.msoe.edu/eecs/cese/resources/stl/
http://www.cs.rpi.edu/~musser/stl-book/
http://www.mochima.com/tutorials/STL.html