Tutorial – Standard Template Library

For this tutorial, we will explore a couple of different types of STL containers, and explore the **<algorithms>** header.

STL Containers

There are a number of different types of containers that STL implements. Each of them have their own advantages and disadvantages, and you will be learning about them as part of your other studies. For this tutorial, we will explore the STL Vector container.

Vectors work as a dynamically resizing arrays – as you add more items, the containers resize to be large enough to fit the information.

Vectors are continuous in memory – that means you can access them via index, exactly like you do with a normal array.

STL Vector

1. Create a new Visual studio C++ console project.
2. Set up your main.cpp file to look like the following:

#include <vector>

int *main*()

{

return 0;

}

1. We are now ready to create a simple vector. The std::vector class is templated, meaning that we can store any type of data in them.

To start with, we’ll create a vector that holds ints:

*std*::*vector*<int> vectorOfInts;

1. We can add items to the end of the vector using the **push\_back**() function, like so:

vectorOfInts.*push\_back*(5);

vectorOfInts.*push\_back*(2);

vectorOfInts.*push\_back*(7);

1. To access a value store in a vector, we can either us the **at**() function, or access it using the subscript operator. Just like arrays, vectors are accessed via an index starting at 0:

int secondValue = vectorOfInts.*at*(1);

int thirdValue = vectorOfInts[2];

1. You can also change the values sorted in a vector the same way as with an array – using the subscript operator again:

vectorOfInts[2] = 10;

1. Just like with arrays, your program will crash (or worse, do something unexpected), if you try and access or write to somewhere outside of the bounds of the vector. You can use the **size**() function to return how many elements are stored in your vector, to protect against reading outside the vector:

if (vectorOfInts.*size*() > 2)

{

vectorOfInts[2] = 10;

}

The **empty**() function can be used to check if the vector is empty. It is a little faster than checking if the size of the vector is larger than 0.

1. Spend some time looking over <https://msdn.microsoft.com/en-us/library/9xd04bzs.aspx> to see a list of all the other functions that can be found inside the std::vector class.

STL Iterators

The STL contains a number of useful functions that act on STL containers. Because each container has a slightly different way of storing memory, they often need a way of iterating through or accessing elements in the container in a consistent manner.

For example –a vector is stored as an array stored on the heap. That means each bit of information is stored continuously in memory. A STL list is built using individual nodes that hold a pointer to the next node in the list (you’ll cover these later in the course). To get to the next bit of information, you need to follow the pointer at the current node, and look at the data stored in the new node.

To deal with this, STL uses the concept of iterators. An iterator is a special object which points to a specific element inside the STL container. If we increment the iterator, the iterator starts looking at the next element – no matter how that may be stored internally behind the container.

1. To get a iterator to the first element of a STL container, we can use the **begin**() function:

*std*::*vector*<int>::*iterator* it = vectorOfInts.*begin*();

1. To access the information that the iterator is pointing to, you need to dereference it like you would a pointer:

//iValue will equal 5

int iValue = (\*it);

(\*it) = 12;

1. To move the iterator, you can use use the + and – operators to tell the iterator to move that many elements:

it = vectorOfInts.*begin*();

it = it + 2;

//it now points at the third element in our vector

++, +=, -= and – all work as you would expect. Some types of containers will only allow you to iterate forward (using ++, +=, +), and others will only allow iterating by a single element (++, --).

1. The end() function returns a iterator that points to the end of the container. This iterator does not actually point to a valid element (it will crash if you try and modify it), but can be used to know when you’ve ‘fallen’ off the end of the container. This is useful for things such as for loops:

for (it = vectorOfInts.*begin*(); it != vectorOfInts.*end*(); ++it)

{

*std*::*cout* << (\*it) << *std*::*endl*;

}

STL Algorithms

Now that you know how to use iterators, you can start to explore some of the helper functions that come with the standard template library.

Take a look over <https://msdn.microsoft.com/en-us/library/yah1y2x8.aspx> to see a list of some of the different algorithms that are available to use.

Exercises

1. Create a vector with 20 different random elements, using the **rand**() function.
2. Print out all the values.
3. Use the std::sort function to sort the values in ascending order.
4. Print out all the values.
5. Change every second value in the vector to a 0.
6. Print out all the values.

Advanced Exercises

For these exercises, you may need to carefully read the <algorithm> documentation to understand how to use some of the more advanced overloads of certain functions.

1. Use the **std::sort** function sort the values in descending order.
2. Change every second value in the vector to a 0 using the **std::for\_each** function.
3. Use the **std::count** function to count the number of 0’s in the vector