Standard Template Library
Introduction

The Standard Template Library (STL) is a collection of classes that represent commonly used data structures and algorithms. These classes should be treated like another feature of the C++ language. Although you should be able to write your own linked list (see my Linked List tutorial if you don't know what a linked list is), there's really no need to write your own for every project you do. The classes in the STL are thoroughly tested and optimized. In fact, there are some cases, such as the string class, where you should almost always use the standard library.

The STL classes belong to the namespace std, so you have to either put using namespace std; at the top of your file or precede any references you make to the library with std::. STL uses templates extensively, so see my Templates Tutorial if you aren't familiar with them.

There are already a lot of STL references out there, so I'm going to stick to showing you functionality in these tutorials. Treat it like when you first learned how to use arrays; you didn't want to know how to program them, you just wanted to know when and how to use them.

Currently I have the following discussions available:

std::string

std::vector

std::stack

std::queue

The string Class

A lot of books that teach introductory C++ tell you to use char pointers for string handling. I learned it this way and it drove me nuts. I was learning Java at the same time and couldn't figure out why C++ didn't have string support. It's such a necessary aspect of programming! What I didn't know is that C++ does have a standard string class, which is aptly named string.

Whenever possible, I recommend using string instead of char pointers. The string class is

clean, provides great functionality, and you don't need to worry about buffer overflows like you do with char pointers.

This discussion will cover some of the functionality offered by the string class. Most of the functions are self-explanatory, so I'll just list the most useful ones here with a description of what each one does. I've also included a sample application that demonstrates the use of the different string functions.

Note that you can declare and use a string variable just like any variable in C++. As I said before, you should treat the classes in the STL just like any other C++ programming language feature.

```
c_str()                     // returns a const char* representation
of the string
length()                    // returns the length of the string
at(int index)               // returns the char at the specified
index, counting starts at 0
operator[](int index)       // overloaded so you can access
characters like an array
clear()                     // erases the string
substr(int start, int end)  // returns the substring from start to
end
find(char*/string str)      // searches the string for a substring
(takes a string or a char*)
```

Click here to download the source code for this tutorial.

The vector Class

First off, the STL vector class does **not** represent a mathematical vector. Actually, it's almost identical to a one dimensional array. The vector class just comes with some great added functionality. It also resizes dynamically so you don't have to worry about it growing too large or wasting too much memory.

The vector class is templated, so you would declare a vector of integers like this:

```
vector<int> intVector;
```

The <> notation might seem a little odd to you. If it does, check out my Templates Tutorial.

A powerful feature of the STL is the use of iterators. Iterators are pointers that point into the structures of the STL. You can have an iterator that points into a vector that stores integers, and point it to the first element of the vector. You can then increment the iterator and it will point to the second element of the vector. You use the increment operator (++) to do this.

You can compare the locations of two iterators using standard conditional operators (<, >, <=, >=). The statement itr1 < itr2 will return true if itr1 points to an element that is closer to the start of the vector.

If you want to get the value stored at the element an iterator points to, use the de-reference operator (*) like this:

```
*itr;
```

STL structures have `begin()` and `end()` functions which return iterators at the beginning of the stucture and one past the end of the structure (not the element at the end, one past it) respectively.

Here's a quick example of using iterators, see the downloadable source code for a more thorough demonstration.

```
// Declare some iterators that point into vectors of integers
vector<int>::iterator itr1 = intVector.begin();
vector<int>::iterator itr2 = intVector.end();

itr1++;       // move the iterator one element forward
itr1 < itr2;  // true if itr1 points to an element coming before itr2
itr += 2;     // move the iterator up by 2 elements

// This is how you would use iterators to loop through a vector
for (vector<int>::iterator itr = intVector.begin(); itr <
intVector.end(); itr++)
{
    cout << *itr;  // *itr gets the value from the element itr points
                   //to
}
```

Here is a list of useful vector functions. See the downloaded source code for working examples of each of these.

```
push_back(T value)              // add an element to the end of the
list
insert(iterator itr, T value)   // insert an element at a specific
position
operator[](int index)           // access an element
at(int index)                   // access an element, throws
exception if index is too big
begin()                         // return an iterator to the
beginning of the vector
end()                           // return an iterator to one past the
end of the vector
clear()                         // erase everything in the vector
pop_back()                      // remove the element at the end of
the vector
```

If we need to sort our vector, we can use the STL `sort()` function. This function takes an iterator to the first element and one past the last element we want sorted. If you want to sort the whole vector, use `begin()` and `end()` like so:

vector

I find the vector class to be the most useful structure in the STL. In fact, I use it in almost all of the tutorials on this site. It's really nice having the instant access of an array with the dynamic size of a linked list. I suggest you try using the vector in place of arrays, unless you have a good reason to be using an array.

Click here to download the source code for this tutorial.

The stack Class

Before we discuss the STL stack, let's talk about what a stack is. I'll use the common example of a stack of plates. If you want to add a plate to the stack, you add it to the top. If you want to take a plate off of the stack, you take it from the top. This behaviour is known as *Last In, First Out* because the last element you add to the stack is the first one to come off.

The stack data structure acts just like a stack of plates. We call a function to add an element and it adds the element to the top of the stack. When we want to access an element from the stack, we call a function that returns the top element. When we want to remove an element, we call a function that removes the top element.

Note that we don't have access to any element other than the one on top (the one most recently added). The act of adding an element to a stack is called "pushing" and the act of removing an element is called "popping".

If you want to understand how a stack is implemented internally, check out my tutorial on Stacks and Queues. Be warned though, stacks are normally built from linked lists so you'll probably have to read that tutorial too if you don't already know how a linked list works. If you just want to use a stack and don't want to have to write it, use the STL stack. Also realize that the STL stack is highly optimized and that STL provides algorithms to do things like sort the data in it (just like with std::vector).

The STL stack is so easy to use that I'm just going to list the functions STL provides and let you download a demonstration to see it in action.

```
// Declare a stack, dataType can be any data type
stack<dataType> myStack;
```

```
myStack.push(item);    // pushes an element onto the top of the stack
myStack.pop();         // pops the top element off of the stack
myStack.top();         // returns the element on the top of the stack
myStack.size();        // returns the number of elements on the stack
myStack.empty();       // returns true if the stack has no elements
```

Click here to download the source code for this tutorial.

The queue Class

The queue data structure is a structure that behaves in a *First In, First Out* manner. Think of a line up at the store. Customers enter the line at the back and exit the line from the front. With the stack, the most recently added element is processed first. With the queue, the most recently added element has to wait its turn.

The STL queue provides a great implementation of a queue. It also adds the functionality to access the back of the queue (maybe the customer at the back gets tired and decides to leave). A better example of a std::queue then might be a road block. The first car that gets to the road block gets to go first (unless the driver gets busted), the car at the back of the road block can make a run for it, and the cars in the middle are stuck there.

If you want to understand how a queue is implemented internally, check out my tutorial on Stacks and Queues. Be warned though, as with stacks, queues are normally built from linked lists so you'll probably have to read that tutorial too if you don't already know how a linked list works. Either way, I suggest you use the STL queue because it's well tested and optimized.

Here's the list of functions that comes with the queue. And don't forget to download the source code and see the queue at work.

```
// We declare a queue like this, dataType can be any data type
queue<dataType> myQueue;

myQueue.push(item); // adds an element to the back of the queue
myQueue.pop();      // removes the element at the front of the queue
myQueue.front();    // returns the element at the front of the queue
myQueue.back();     // returns the element at the back of the queue
myStack.size();     // returns the number of elements in the queue
myStack.empty();    // returns true if the queue has no elements
```

Click here to download the source code for this tutorial.