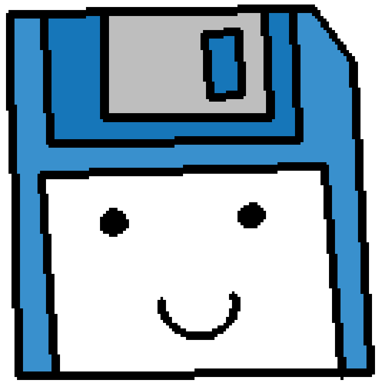
The Standard Template Library

Written by Rachel J. Morris, last updated 2016-04-16

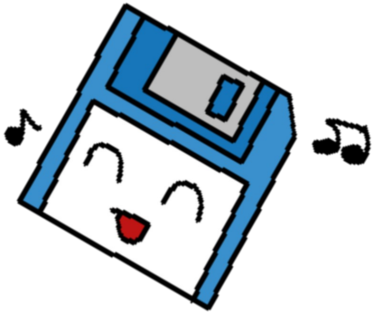
Why reinvent the wheel?



Data structures

The C++ Standard Template Library is a library that contains functionality for containers, as well as other things.

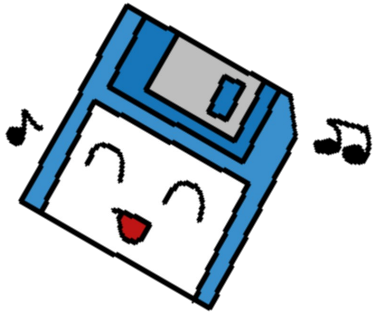
These come in really handy, so we don't have to write our own data structures and optimize them ourselves – we can use the containers that are part of the library.



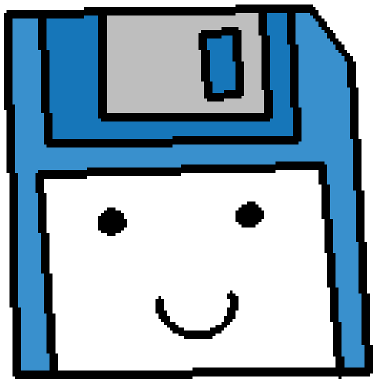
Data structures

Before we get into writing our own data structures like linked lists, let's look at the structures available in the C++ Standard Template Library.

By utilizing these structures before we get into how to implement our own, you can get a better vision for how they are supposed to work.



vector



vector

The vector class essentially behaves like an array, but the vector class will handle resizing on its own.

We can insert items into a vector object with the **`push_back( … )`** function

And randomly access elements of the vector with the subscript operator **`[ ]`**

We can also get the amount of items in the vector with the **`size()`** function

There is also functionality to clear out the entire vector, or just one element.

vector

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

int main()
{
    return 0;
}
```

First, you need to include the
`<vector>`
Library.

vector

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

int main()
{
    vector<string> cities;

    return 0;
}
```

Declare the vector variable, specifying the data-type that it will store within the < and > signs.

vector

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

int main()
{
    vector<string> cities;

    cities.push_back( "Raytown" );
    cities.push_back( "Lee's Summit" );
    cities.push_back( "Independence" );

    return 0;
}
```

Add items to the vector with the **`push_back`** function.

Element at index 0

Element at index 1

Element at index 2

vector

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

int main()
{
    vector<string> cities;

    cities.push_back( "Raytown" );
    cities.push_back( "Lee's Summit" );
    cities.push_back( "Independence" );

    cout << "Size: " << cities.size() << endl;

    return 0;
}
```

You can get the size of the vector at any time with the **`size()`** function.

vector

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

int main()
{
    vector<string> cities;

    cities.push_back( "Raytown" );
    cities.push_back( "Lee's Summit" );
    cities.push_back( "Independence" );

    cities.erase( cities.begin() + 1 );

    return 0;
}
```

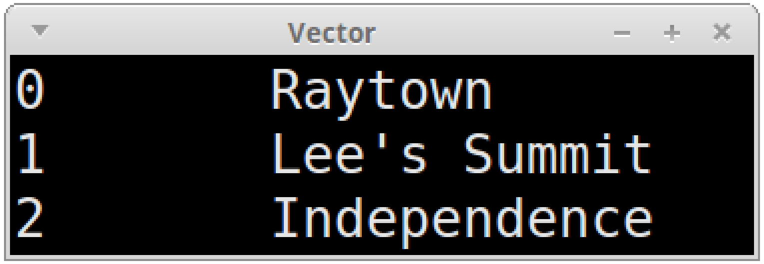
Remove item #1, Lee's Summit

To remove an item at a specific index, use the **`erase`** function.

The argument will be **`variable.begin()`** , then add the index #. (more on what this is later.)

vector

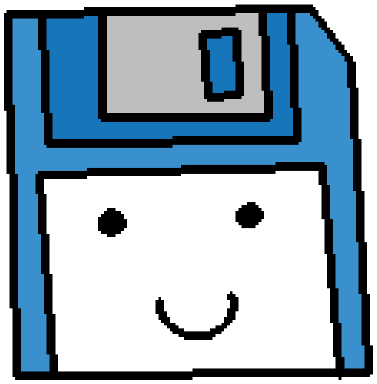
```
vector<string> cities;

cities.push_back( "Raytown" );
cities.push_back( "Lee's Summit" );
cities.push_back( "Independence" );

for ( unsigned int i = 0; i < cities.size(); i++ )
{
    cout << i << "\t" << cities[i] << endl;
}
```



You can easily output all the elements of the vector with a for loop, going from 0 to the size of the vector.

We can also use **iterators** to iterate through every element, but more on that later!

map



map

The map class is like an array, but the indices can be integers or other data-types. This is known as a key-value pair.

Each index must be unique, as that is the *key* that represents the *value* object.

To insert an item into a map, you need to create a **pair** object. Both the **map** and **pair** need to have the same template types.

map

```
#include <iostream>
#include <string>
#include <map>
using namespace std;

int main()
{
    return 0;
}
```

First, you need to include the <map> library.

map

```
#include <iostream>
#include <string>
#include <map>
using namespace std;

int main()
{
    map<int, string> employees;

    return 0;
}
```

The map object needs two template types, the **key** first and the **value** second.

For this example, the key is an integer – could be an employee ID

The value is a string – could be an employee name

map

```
#include <iostream>
#include <string>
#include <map>
using namespace std;

int main()
{
    map<int, string> employees;

    employees.insert( pair<int, string>( 1001, "Dorian" ) );
    employees.insert( pair<int, string>( 1004, "Turk" ) );
    employees.insert( pair<int, string>( 1007, "Reid" ) );

    return 0;
}
```

Use the **insert** function to insert a new element into the map.

But you need to pass in a **pair** object as the item being inserted.

You could declare the **pair** outside of the insert function, or just as the argument.

map

```
#include <iostream>
#include <string>
#include <map>
using namespace std;

int main()
{
    map<int, string> employees;

    employees.insert( pair<int, string>( 1001, "Dorian" ) );
    employees.insert( pair<int, string>( 1004, "Turk" ) );
    employees.insert( pair<int, string>( 1007, "Reid" ) );

    return 0;
}
```

Use the **insert** function to insert a new element into the map.

The **pair** types need to match the **map** types.

Then, within parenthesis (the constructor), you pass in the values.

map

To access the value at a particular index, you can use the subscript operator with the key, or use the **at** function.

```
#include <iostream>
#include <string>
#include <map>
using namespace std;

int main()
{
    map<int, string> employees;

    employees.insert( pair<int, string>( 1001, "Dorian" ) );
    employees.insert( pair<int, string>( 1004, "Turk" ) );
    employees.insert( pair<int, string>( 1007, "Reid" ) );

    cout << "Employee # 1001:\t" << employees[1001] << endl;
    cout << "Employee # 1004:\t" << employees.at( 1004 ) << endl;

    return 0;
}
```

map

```
map<int, string> employees;

employees.insert( pair<int, string>( 1001, "Dorian" ) );
employees.insert( pair<int, string>( 1004, "Turk" ) );
employees.insert( pair<int, string>( 1007, "Reid" ) );

employees.erase( 1001 );

employees.clear();
```

You can clear out the map with **clear()** function

You can erase a specific item with the **erase()** function, and pass in the key of the element.

map

```
map<int, string> employees;

employees.insert( pair<int, string>( 1001, "Dorian" ) );
employees.insert( pair<int, string>( 1004, "Turk" ) );
employees.insert( pair<int, string>( 1007, "Reid" ) );

for (
    map<int, string>::iterator it = employees.begin();
    it != employees.end();
    it++ )
{
    int id      = it->first;
    string name = it->second;
    cout << "ID: " << id << ", Name: " << name << endl;
}
```

If you want to iterate through every item in the map, you will have to use **iterators**.

Yep, it looks weird. Let's break down this loop...

map

```
for (
    map<int, string>::iterator it = employees.begin();

    it != employees.end();

    it++ )
{
    int id      = it->first;
    string name = it->second;
    cout << "ID: " << id << ", Name: " << name << endl
}
```

This is a for loop, but you're used to seeing it like

for (a = 0; a < size; a++)

This loop is in the same order:

1. Declare a variable

2. Specify the criteria for the loop to continue

3. Specify what gets done every time through the loop.

map

```
for (
    map<int, string>::iterator it = employees.begin();

    it != employees.end();

    it++ )
{
    int id      = it->first;
    string name = it->second;
    cout << "ID: " << id << ", Name: " << name << endl;
}
```

First, we have to create an **iterator**.
It will be an iterator of your map, so use the map data-type with the same template types. Use ::iterator at the end. This whole thing is the data-type.

Next, “it” is the variable name of our iterator.

The initial value is the beginning of our map – which we can get with *variablename.begin()*

map

```
for (
    map<int, string>::iterator it = employees.begin();

    it != employees.end();

    it++ )
{
    int id      = it->first;
    string name = it->second;
    cout << "ID: " << id << ", Name: " << name << endl;
}
```

Second, we specify the criteria for when the loop will continue looping.

You will loop until you reach the end of the map, so it will run into *variablename.end()*

map

```
for (
    map<int, string>::iterator it = employees.begin();

    it != employees.end();

    it++ )
{
    int id      = it->first;
    string name = it->second;
    cout << "ID: " << id << ", Name: " << name << endl;
}
```

Finally, every time through the loop, we are going to increment the iterator. This will go to the next element in the map.

map

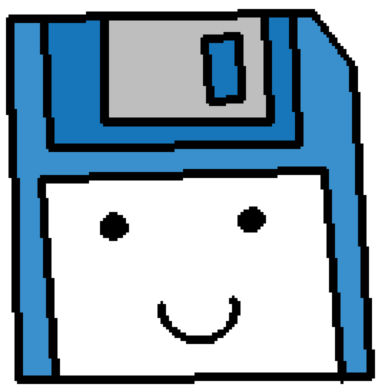
```
for (
    map<int, string>::iterator it = employees.begin();

    it != employees.end();

    it++ )
{
    int id      = it->first;
    string name = it->second;
    cout << "ID: " << id << ", Name: " << name << endl;
}
```

The iterator is essentially a pointer.

You can get the **key** of the element by accessing the **first** member.

You can get the **value** of the element by accessing the **second** member.

list



list

```
#include <list>

list<string> states;
```

To use a **list**, you will need to include the <list> library.

A list is similar to a vector, but there are differences!

list

```
#include <list>

list<string> states;

// Insert at end
states.push_back( "Missouri" );
states.push_back( "California" );

// Insert at start
states.push_front( "Kansas" );
states.push_front( "Nebraska" );

// Insert at any position
list<string>::iterator pos = states.begin();
pos++; pos++; // start at 0 and move right twice
states.insert( pos, "Washington" );
```

With vector, you are generally just adding items to the end of the array with the push_back function

With a list, you can push to the end of the list

The beginning of the list

Or somewhere in-between!

list

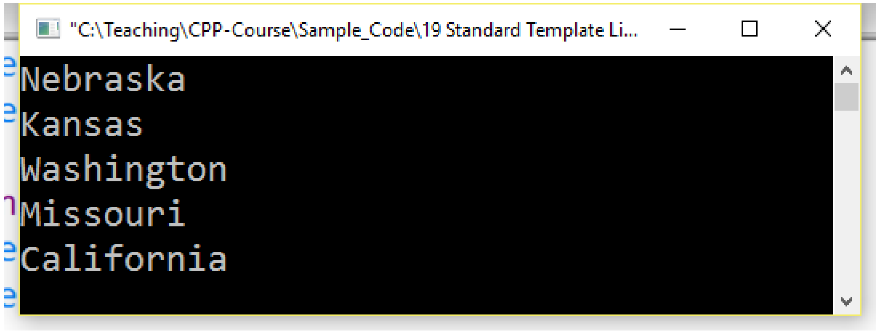
```
list<string> states;

// Insert at end
states.push_back( "Missouri" );
states.push_back( "California" );

// Insert at start
states.push_front( "Kansas" );
states.push_front( "Nebraska" );

// Insert at any position
list<string>::iterator pos = states.begin();
pos++; pos++; // start at 0 and move right twice
states.insert( pos, "Washington" );
```

```
for (
    list<string>::iterator it = states.begin();
    it != states.end();
    it++
)
{
    cout << *it << endl;
}
```

But you can't randomly access items at some index with the subscript operator [ ] like you can with a vector...

You will have to use an iterator to access every item in the list.

(Or to navigate to a specific index)

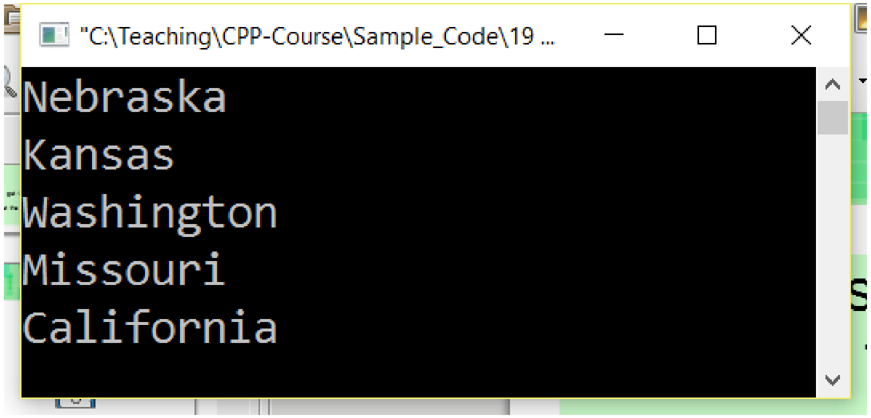


list

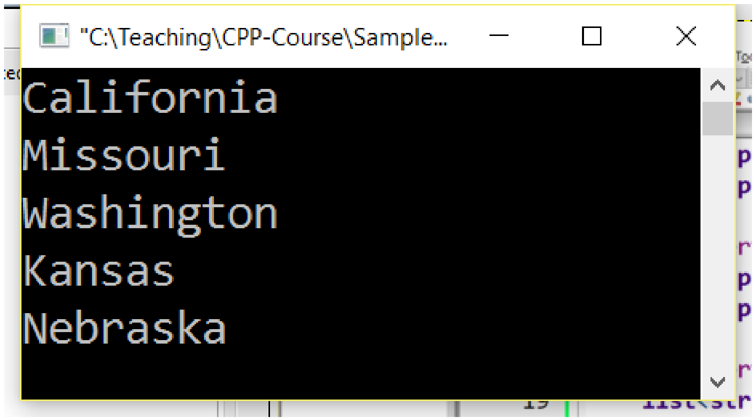
There is also a function called reverse() that will reverse your list elements

```cpp
states.reverse();
```

No reverse



Reverse

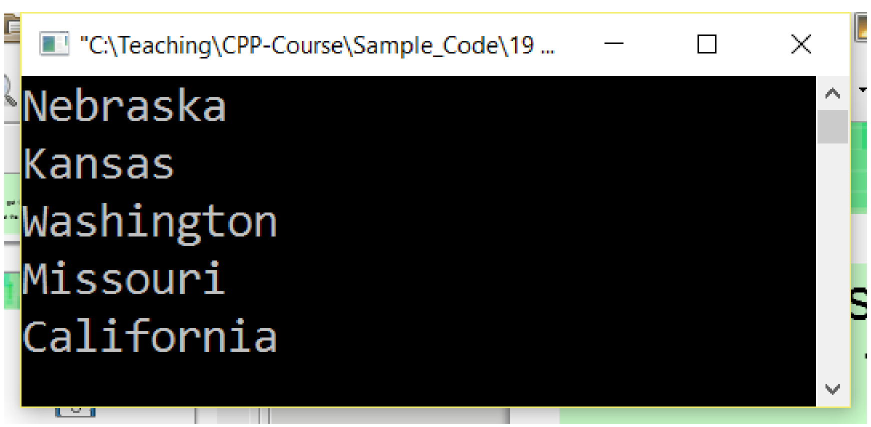


list

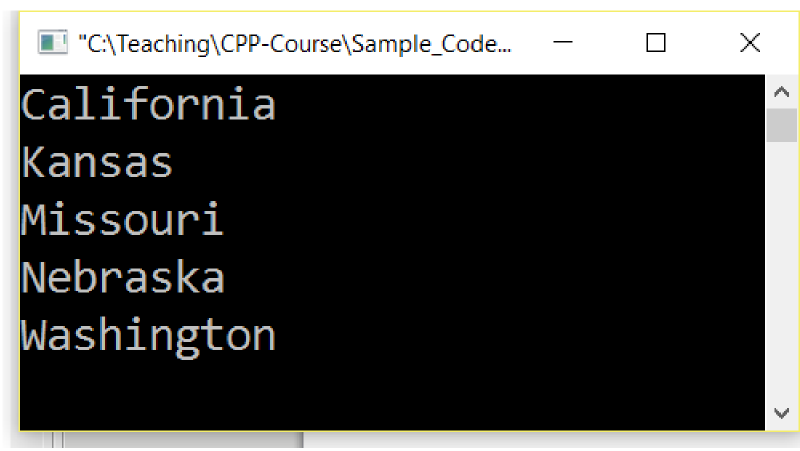
And a handy sort() function that will put your elements in order.

```
states.sort();
```

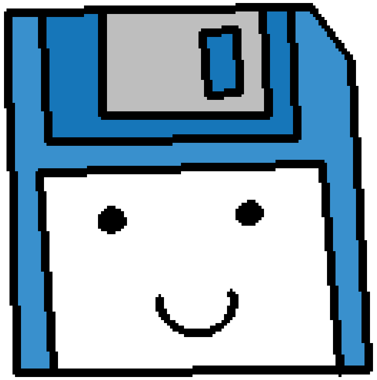
No sort



Sort



queue



queue

A queue is known as a
first-in-first-out
structure

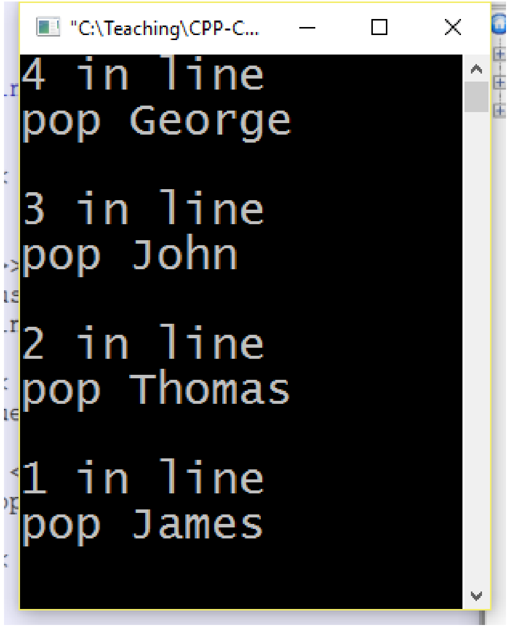
You can add items to the queue and remove items from the queue, and the first item to be added will be the first item to be removed.

0 George	1 John	2 Thomas	3 James

In C++ terms, you have
push_back() and pop_front()
But for queue, it is just push() and pop().

queue

```cpp
queue<string> presidents;

presidents.push( "George" );
presidents.push( "John" );
presidents.push( "Thomas" );
presidents.push( "James" );

while ( !presidents.empty() )
{
    cout << presidents.size() << " in line" << endl;
    cout << "pop " << presidents.front() << endl << endl;
    presidents.pop();
}
```



As you push items into the queue, the first item goes in front and everything else “lines up” behind it.

A queue structure might be useful for modeling *incoming work to be processed*, so that items that are received first are dealt with first.

queue

```
queue<string> presidents;

presidents.push( "George" );
presidents.push( "John" );
presidents.push( "Thomas" );
presidents.push( "James" );
```

Use push() to add an item to the queue

```
while ( !presidents.empty() )
{
```

Empty() will return true or false

```
    cout << presidents.size() << " in line" << endl;
```

Or you can get the # of things in the queue with size()

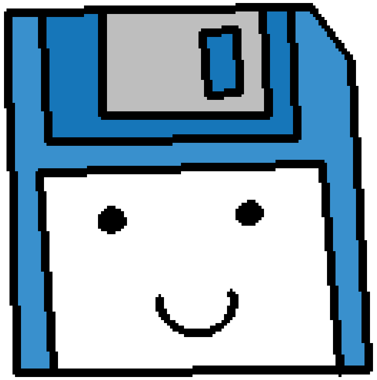
```
    cout << "pop " << presidents.front() << endl << endl;
```

Access the item at the front of the queue with front()

```
    presidents.pop();
}
```

Remove the item at the front with pop()

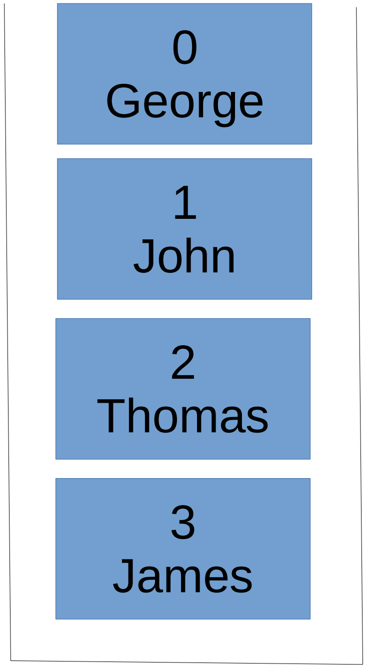
stack



queue

A stack is known as a
first-in-last-out
structure

The first item added to the stack will be the last item to be removed. (push_back, pop_back)

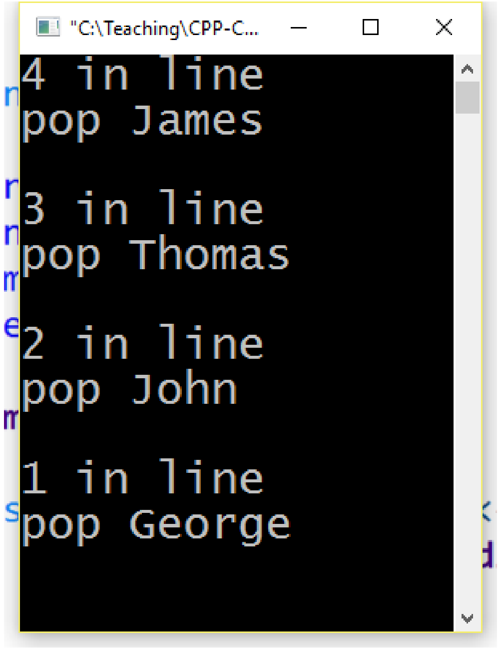


So when you **push** something onto the stack, it goes “on top”.

If you **pop** something off of the stack, it removes the “top-most” item.

queue

```
stack<string> presidents;

presidents.push( "George" );
presidents.push( "John" );
presidents.push( "Thomas" );
presidents.push( "James" );

while ( !presidents.empty() )
{
    cout << presidents.size() << " in line" << endl;
    cout << "pop " << presidents.top() << endl << endl;
    presidents.pop();
}
```



The first item goes on the bottom, and all subsequent items end up on top of it.

And with a stack, you can only access the top-most item.

queue

```
stack<string> presidents;

presidents.push( "George" );
presidents.push( "John" );
presidents.push( "Thomas" );
presidents.push( "James" );

while ( !presidents.empty() )
{
    cout << presidents.size() << " in line" << endl;

    cout << "pop " << presidents.top() << endl << endl;

    presidents.pop();
}
```

Use **push()** to add items to the stack

Check if the stack is empty with **empty()**

Get the # of elements in the stack with **size()**

Look at the top-most item with **top()**

Remove the top-most item with **pop()**

Data structures

The C++ STL is pretty handy and it is good to be familiar with how to use them.

The cplusplus.com reference page also has a list of functions and sample code for using these structures!