Overcoming Limitations, Reading

The data structure we developed in the previous module demonstrates some very important coding features that we'll be using throughout most of the rest of this course:

- how to cast data structures as objects using C++ class containers,
- how to use private data members to store and track values stored in the data structure object,
- about getters and setters and the trailing const,
- about the square bracket operator in both getter and setter forms, and
- how to develop and test a class, using a test driver.

But it left us with **serious shortcomings** -- the code had to be rewritten for **different data types** and **other capacities**. The C++ Standard Template Library **(STL) overcomes these** exact same limitations in their versions of our Array class -- consider the "vector" and the C++11 "array".

The STL vector

Just briefly, to compare and contrast, here's our Array class (of 10 ints) vs the STL's vector (also for 10 ints):

```
Array a;
```

```
vector<int> a(10);
```

The 10 and the "int" are built in to our Array coding. But the 10 and the "int" are part of the vector's declaration, suggesting that the data type could be just about anything other than an int simply by **typing it in the declaration**, and the capacity can be anything other than 10, simply by **typing a different number in the declaration**. They both support **a[i]**. So vector does what we want to do.

The STL array (C++11)

Also to compare and contrast, let's compare to the C++11 STL's array -- same name, lower case:

```
Array a;
```

```
array<int, 10> a;
```

it's just like vector, except that the capacity (10) *must* be a number or a constant, while it can be a variable in the vector. It might seem like a big step backwards and why would we ever want the new (with C++11) array when we already have the vector, but there is a reason. The **array uses stack memory** while the vector uses dynamic memory, so the array is more efficient.

But don't count on the STL version initializing all values to their defaults -- zero in the case of ints.

The "T" In STL

Here's how they do it (in the STL) -- they use a (very C++) coding feature called a "**template**". What you'll see in this module may appear to be the same thing that other languages do (like "generics" in Java), but not so! The C++ implementation is incomparably more efficient because it automatically generates the code that (for the Array class) we would have has to copy, paste, and mark up ourselves.