Big Oh Considerations, Reading

Did you notice a bit of a disconnect between the previous two modules? Module 7 was about algorithmic efficiency and how we prefer O(1) to O(n). Then in Module 8's associative arrays we applied O(n) loops to just about everything in the template! It's like we just threw out all the work we did in the previous module.

Well, *this* module takes a look at the O(n) issue and looks for a solution.

The Square Bracket Operator

With the StaticArray and DynamicArray templates, the square bracket operator is efficient (in a big oh sense) because the numeric index can be used to directly calculate the memory location of the indexed value and find it with O(1). It works like this, even though *we* don't actually do the math: for a **DynamicArray<int> a;**, the 9th value is at **a[9]**. In the function, using array notation as we do, the computer calculates the *offset* in the "values" array as **9*sizeof(int)** and goes directly there to retrieve the value. No traversing, no counting. Just a simple calc resulting in O(1).

But with the AssociativeArray template, the square bracket operator uses a for-loop to traverse the whole array of nodes, searching them one-by-one for a match to the **typename K** parameter, using equals-equals. That's O(n). What would be really nice is if we could get that to be O(1) the same way that StaticArray::operator[ ] does. What *it* does is *calculate* the memory location. AssociativeArray::operator[ ] needs a way to *calculate* where the value associated with its **typename K** parameter is, instead of *searching* for it.

The solution is to **convert the parameter into an offset in a simple math calculation**, like we do with numeric whole number parameters.