Redefining The "Index", Reading

Ever since you learned about arrays and their square brackets, you only ever put a whole number *inside* the square brackets -- either as a non-negative number or as an int variable, like this:

```
a[10]
```

```
a[n]
```

But since we learned about redefining the square bracket operator in Module 3, its prototypes might have suggested other possibilities:

```
V& operator[ ](int);
```

```
V operator[ ](int) const;
```

That "int" specifies the data type for the *index*. Remember that for objects, "**a[i]**" means "**a.operator[ ](i)**"? It replaced **a.getAtIndex(i)**. "i", the index, is the parameter. Here's the thing: that "int" parameter *can be any valid C++ data type*! "Associative arrays" are data structure objects with a square bracket operator whose parameter is *not* necessarily an int. The index does not have to be an int!

The Evolution Of The Index

Consider a non-templated data structure object named "phoneBook" that has a lookup function with this prototype: **unsigned long long getForName(string name) const;**. ("unsigned long long" because phone numbers have 10 digits.) The parameter is a name, and the return is a phone number as an unsigned long long. So the following returns the phone number for Joe:

```
phoneBook.getForName("Joe")
```

Doing exactly what we did in Module 3, we can replace the function name "getForName" with "operator[ ]". Now the prototype is: **unsigned long long operator[ ](string name) const;** and we get this:

```
phoneBook.operator[ ]("Joe")
```

same as

```
phoneBook["Joe"]
```

Remember that there are *two* square bracket operator functions -- a getter and a setter. That means it should be possible not only to look up a phone number based on a name, is should also be possible to *set* a phone number like this:

```
phoneBook["Jane"] = 9255551212;
```

Associative Arrays In C++ And Other Languages

Most modern languages support associative arrays, all with just about the same syntax. Without getting into the details of the declarations, here's how you'd write a statement in various languages to store a phone number for George:

C++ STL

```
phoneBook["George"] = 9256251230;
```

PHP

```
$phoneBook["George"] = 9256251230;
```

Python

```
phoneBook["George"] = 9256251230
```

JavaScript
```
phoneBook["George"] = 9256251230
```

Java
```
phoneBook.put("George", 9256251230);
```

All except Java have square bracket syntax.

Generalizing Associative Arrays In Templates

When we went to *templated* data structure classes, we wrote these prototypes:

```
template <typename V>
class Array
{
  ...
  V operator[ ](int) const;
  V& operator[ ](int);
```

That's with an *int* as the parameter. Now that we've reasoned that the parameter can be *any* data type, we need *another* template variable to stand for the int -- something in addition to "V". But before we go ahead and call it "typename I" for "index", we're going to use a different, more generic word. "Index" definitely means a 0-based numeric position in an array. That generic word in programming terms is "key".

So in **a[i]**, "i" is the "index". In **phoneBook["Joe"]**, "Joe" is the "key". Here's how the template structure changes to allow the key to be any data type:

```
template <typename K, typename V>
class AssociativeArray
{
  ...
  V operator[ ](const K&) const;
  V& operator[ ](const K&);
```

"K" is the "key" data type. But it's not a simple substitution of "K" for "int" as the parameter. It's **const K&**. That's because the data type can be an *object*, like a C++ string, and when we share objects in parameter lists, our preference is as a read-only reference instead of a copy. If the key is *not* an object, it still works.

Declaration Statement

We're ready for our specification for the template. It's very much the same as it was for the **StaticArray** and **DynamicArray**, except that the declaration statement needs to include the data types of both the value *and* the key, like this:

```
  AssociativeArray<string, unsigned long long> phoneBook;
```