How To Get a[key] To Zero Out Its Value?, Reading

For an associative array, which saves key-value pairs of data, doing the following returns the *value* for the matching *key*:

```
AssociativeArray<string, long long> phoneBook;
phoneBook["Fred"]
```

If there is not already a matching key, a key-value pair for that key is created, and the key copied to it, and then a reference to the *uninitialized* value gets returned (by the square bracket *setter*).

But it might be useful for the *value* to be initialized too, when the key-value pair is created, so that a default value gets assigned. Then the following would never result in an unpredictable result:

```
cout << phoneBook["Nancy"];
```

Good defaults for ints and doubles would be zero. False for Booleans, blanks for strings, and the null terminator for chars. For objects, their default constructor could be called. This does not add a restriction on the data type of **typename V**, because **V dummy;** already requires that there be a default constructor or none at all.

To implement this feature, add this line to the code block that assigns the key (from the parameter list) to the array location or newly created Node:

```
// arrayed
data[indexOfFirstUnusedKey].key = key;
data[indexOfFirstUnusedKey].value = V( );
data[indexOfFirstUnusedKey].inUse = true;
```

```
// linked
Node* temp = new Node;
temp->key = key;
temp->value = V( );
temp->next = firstNode;
```

Remember that **int x = int( );** is another way to initialize a new int to zero? Try it and see! Works with any other data type (as long as its not an object that has a constructor but no default constructor, which is rare).

Alternatively you can do this in C++11, in the **struct Node** definition:

```
// arrayed
struct Node
{
  K key;
  V value{};
  bool inUse{};
};
```

```
// linked
struct Node
{
  K key;
  V value{};
  Node* next{};
};
```

These initialize the value, in-use flag, and next pointer all to zero when Nodes are created. (There's no point in initializing the key, because it gets initialized when its Node is put into use.)

An Application: Counting Occurrences Of Keys

This feature is particularly useful when counting the number of occurrences of keys, because you would not have to call containsKey to decide if you have to set the returned value reference to 1 or to add 1 to it, like this:

```
AssociativeArray<string, int> count;
```

```
 ...
 while (true)
 {
   string key = ...
   if (count.containsKey(key))
     count[key]++;
   else
     count[key] = 1;
   ...
 }
```

Instead, you could simply write this:

```
AssociativeArray<string, int> count;
 ...
 while (true)
 {
   string key = ...
   count[key]++;
   ...
 }
```

...because if the key does not yet exist, **count[key]** creates the key-value pair **and sets the value to zero** *before* returning a reference to the value, where it's incremented by 1, to now equal 1.