Hashing and **HashMaps**

Chapter 15 discusses a few ways we might implement the **Map** introduced during the second week of the course. You should be reading through Chapter 15 now, focusing on high-level concepts, cognizant of the fact that the **HashMap** we implement in lecture is more advanced than anything you'll read in Chapter 15. I've gotten several requests asking I teach template implementation, and I'm using our **HashMap** as an opportunity to do so.

Truth be told, the **Map** we've been using is backed by a binary search tree, and we won't learn about those until next week. Fortunately, there are many choices for the internal representation, and Chapter 15 uses the **Map**'s interface as a vehicle for learning about lookup tables, hashing, and hash tables. As it turns out, we're in a better position to learn hashing and hash tables because of our recent work with linked structures, so we're going with the hashing approach first. To be clear we're presenting a different **Map** implementation than the version you've been coding against, I'm calling this version the **HashMap**. We're implementing to the same exact interface, so you'll see value in what we're covering during the rest of today's lecture.

hash-map.h

```
template <typename Key, typename Value>
class HashMap {
public:
   HashMap(int sizeHint = 10001);
   ~HashMap();

   bool isEmpty() const { return size() == 0; }
   int size() const { return count; }

   bool containsKey(const Key& key) const;
   void put(const Key& key, const Value& value);
   Value get(const Key& key) const;
   Value& operator[](const Key& key);

private:
   struct node {
      Key key;
      Value value;
      node *next;
   };

   node **buckets;
   int numBuckets;
   int count;
   int hash(const Key& key) const;
   node *ensureNodeExists(const Key& key);
   const node *findNode(const Key& key) const;
};

#include "hash-map-impl.h"
```

You'll notice that the interface here is identical (at least to the extent that I implement it) to that of the **Map**. This is, of course, intentional, as we're electing to provide the machinery to make the black-box ADT work for the purposes of the client.

The two surprises above:

- The interface doesn't commit to key and value types, but instead confesses that the **`HashMap`** class is templatized on two types determined only at the moment one is instantiated. The placement of the **`template`** directive before the class declaration informs the compiler that what follows is incomplete, and that it can't fully processed (beyond obvious parsing needs). It's only when client code **`#include`**s **`hash-map.h`** and declares something like, say, **`HashMap<char, Vector<int> >`**, that the compiler associates **`Key`** and **`Value`** with **`char`** and **`Vector<int>`** and expands the definition to be **`char`** and **`Vector<int>`**-specific for that one instantiation.
- Because it's a template, **`hash-map.h`** **`#include`**s **`hash-map-impl.h`** at the bottom of the file! Because all method implementations are also templatized, the full implementation needs to be visible in the code unit that declares a **`HashMap`**. The **`#include`** mechanism is little more than search and replace: During compilation, the **`#include "hash-map-impl.h"`** line is removed and replaced with the contents of the **`hash-map-impl.h`** file, and processed as if the code were physically typed in **`"hash-map.h"`** all along.

`hash-map-impl.h`

In most ways, implementing a template is like implementing a strongly typed class, where you operate as if the template parameters—in this case, **`Key`** and **`Value`**—are authentic data types. You sometimes need to make assumptions about how **`Key`** and **`Value`** behave and what operations they support, and when you do, those prerequisites would normally be surfaced in the official interface file documentation. In this case, we require that **`Key`** play well with **`operator==`** and that it be hashable, using either some library routines, or through some hashing code we hand-roll ourselves.

```
template <typename Key, typename Value>
HashMap<Key, Value>::HashMap(int sizeHint) {
   if (sizeHint <= 0) error("size hint passed to HashMap constructor "
                            "must be positive.");
   count = 0;
   numBuckets = sizeHint;
   buckets = new node*[numBuckets];
   for (int i = 0; i < numBuckets; i++) buckets[i] = NULL;
}

template <typename Key, typename Value>
HashMap<Key, Value>::~HashMap() {
   for (int i = 0; i < numBuckets; i++) {
      node *curr = buckets[i];
      while (curr != NULL) {
         node *next = curr->next;
         delete curr;
         curr = next;
      }
   }
}

template <typename Key, typename Value>
```

```
bool HashMap<Key, Value>::containsKey(const Key& key) const {
   return findNode(key) != NULL;
}

template <typename Key, typename Value>
void HashMap<Key, Value>::put(const Key& key, const Value& value) {
   ensureNodeExists(key)->value = value;
}

template <typename Key, typename Value>
Value HashMap<Key, Value>::get(const Key& key) const {
   const node *found = findNode(key);
   return found == NULL ? Value() : found->value;
}

template <typename Key, typename Value>
Value& HashMap<Key, Value>::operator[](const Key& key) {
   return ensureNodeExists(key)->value;
}

template <typename Key, typename Value>
int HashMap<Key, Value>::hash(const Key& key) const {
```

implementation omitted, as it uses lots of specialized blocks of code, depending on whether or not **Key**—*the type being hashed to a number between 0 and* **numBuckets - 1**, *inclusive—is* **int**, **unsigned long long**, **char**, **double**, **std::string**, *etc.*

```
}

template <typename Key, typename Value>
typename HashMap<Key, Value>::node *
HashMap<Key, Value>::ensureNodeExists(const Key& key) {
   int hashcode = hash(key);
   node *bucket = buckets[hashcode];
   node *found = const_cast<node *>(findNode(key));
   if (found == NULL) {
      found = new node;
      found->key = key;
      found->value = Value();
      found->next = bucket;
      buckets[hashcode] = found;
      count++;
   }

   return found;
}

template <typename Key, typename Value>
const typename HashMap<Key, Value>::node *
HashMap<Key, Value>::findNode(const Key& key) const {
   int hashcode = hash(key);
   const node *curr = buckets[hashcode];
   while (curr != NULL && !(curr->key == key)) {
      curr = curr->next;
   }
   return curr;
}
```