Hash Codes, Reading

Here's how the numeric parameter **int index** works in the square bracket functions -- it's used to calculate the offset from the start of the **typename V** data array "values" like this: **index*sizeof(V)**. That is:

$$f(index) = index \cdot sizeof(V)$$

That's a function that returns the offset from the beginning of the array **as an int**. For the associative array with an ADT parameter **const K& key**, we need something similar -- something with this form:

$$f(key) = \ldots \cdot sizeof(Node)$$

This should *also* return an int. Since we don't know what "key" is, because it's an ADT, we cannot write the rest of this function in the template. That's a problem we'll put off until later, but for now let's just explore **ways that such a function could even be written**.

The "Desired Index"

Let's say that the **typename K** is a C++ string. How could a string value be turned onto a whole number int? You're probably already thinking of something which involves the ASCII values associated with the chars in the string. If so, you're on the right track! For example, the three ASCII codes in "Joe" are 74 (J), 111 (o), and 101 (e). We could combine them in various clever ways, but the easiest is just to add them: 286. So **f("Joe")=286·sizeof(Node)** is the offset in the **Node* data;** array. Wow -- we have successfully turned a string key into an offset, and "286" looks like a numeric index!

We'll call this the "desired index". It's the index in the **Node* data;** array that any value for the key "Joe" would want to be stored. You can see the advantage -- rather than looking through the whole array for an in-use node whose key matches "Joe". That's back to O(1), which was our quest! (And yes, there are going to be some issues with this idea, and it's good that you are already thinking about them, but be assured -- we'll find ways to deal with them by the end of this module.)

A Hash Code Function

Let's start with the easy part -- a function that takes a key as its input and returns an int. For the above example with a C++ string as a key, it could be something like this:

```
int hashCode(const string& key)
{
  int result = 0;
  for (int i = 0; i < key.length(); i++)
    result += key.at(i);
  return result;
}
```

Note that this is *not* templated, because (1) the data structure template is not supposed to know such details about the key (in this case, that there is a string::at function), and (2) this will have to go in the main program's CPP.

In general for any main program that uses this templated data structure, it's going to have to have this kind of function, called the "hash code function", designed by the programmer to convert whatever is the key into an int. For example, with Student objects as the key, the function might be:

```
int hashCode(const Student& key)
{
  return key.studentID;
}
```

Getting The Hash Code Function To The Template

Getting The Hash Code Function To The Template

So just how *do* we get this function to the template? Recall that *variables* can be **passed by reference** in parameter lists, which effectively tells the receiving function where that variable is located in memory, so that the function can have direct access to it through an aliased name. There's old-style C pointer syntax for that, and new-style reference variable syntax for that too. Both send the *memory location* of the specified data type to the function, and *not* a copy of its value.

Well, the C++ language uses the exact same technology to **share the location of *functions*** through the parameter list. Yes, functions have locations in memory just like everything else! The receiving function uses an aliased name for the shared function. So here's what we'll do -- we'll write our hash code function in the main program's CPP, then we'll get its memory location (how?). In a call to a template public member function, we'll send that memory location to the template (*really?* how?). The template will remember that location in a private data member (*what? how?*), and use that in its functions instead of using the O(n), searching for-loops.

Ok, so there's a lot of that plan that we just don't know how to do. First, what is the "public member function" that we're supposed to call, and where does the hash code function name go in its parameter list, and how do we put it there so that it's seen as its memory location? Since it's information the template will need for its whole existence, and it's not going to change, why not just use the constructor? That's what we'll do. For that we need to write the main constructor's prototype, and for that we need a template to put it in, so let's get that done:

```
template <typename K, typename V, int CAP>
class HashTable
{
  ...
  public:
  HashTable(int(*)(const K&)=0); // =0 because we still need a "default constructor"
  ...
};
```

Actually this forces us to come up with a name for the template ("HashTable") and to make a decision about the underlying data structure (a static array, hence the **int CAP** for capacity). "Hash table" is the common way to refer to this data structure. And since we already know that we're trying to avoid for-loops, linked lists are out, so the choice is between static and dynamic arrays. To see things simple, let's go with static and avoid issues of capacity adjustments and dynamic memory management.

Anyhow, look at the constructor's parameter's data type, "**int(*)(const K&)**". That's the data type specification for a *pointer* that stores the memory location of an *int-returning* function that has *one parameter*: a constant reference to a variable of ***typename K***. Funny looking syntax, but in its defense, it really does have a lot to "say"! It packs a lot of information into itself -- the return type *and* the parameter list.

*(The **=0** in the parameter list is just like **Array(int=2);** from the dynamic array -- there needs to be some default value for the parameter, or else there will be certain limitations in how we can use this template.)*

How to get the hash code function into the HashTable declaration is actually the easy part, because just as it is with arrays, the name of the array or function serves as its memory location -- no leading ampersand syntax to deal with. So:

```
  HashTable<string, unsigned long long, 100> phoneBook(hashCode);
```

...sends the information to the template. Now all we need is a way for the template to remember! For that we need a private data member of the same data type as the constructor's parameter. You might expect that it would look like this:

```
template <typename K, typename V, int CAP>
class HashTable
{
  int(*)(const K&) hashCode; // alias for hash code function
```

...but it doesn't! The syntax for declaring a pointer to a function is like this:

```
  int(*hashCode)(const K&); // alias for hash code function
```

The pointer name, unlike *any* other declaration in C++, is stuck inside the data type's specification. Weird...