Objects As Keys In Associative Arrays, Reading

Actually, C++ strings are objects, and our examples so far have had strings as keys. So we might be lulled into thinking that all this works for objects. But not so fast! It just so *happens* that all this works for strings because strings implement certain actions and behaviors that *allow* them to be used as keys.

Remember abstract data types (ADTs)? The key is an abstraction -- it's not a real data type in the template coding. In our implementation we assume these things about **typename K**:

1. it either has no constructors *or* it has a default constructor,
2. it supports equals-equals comparisons.

Take the following object type, for example:

```
struct Time
{
  int hour, minute;
  Time(int h, int m){hour = h; minute = m;}
};
```

The constructor *forces* Time objects to be created with a specific hour and minute specified, or else it will not compile.

```
  ~~Time time;~~ // will not compile
  ~~Time times[100];~~ // will not compile
  ~~Time* times = new Time[n];~~ // will not compile
  Time noon(12, 0); // will compile
```

This means that the **new Node** statements would all fail to compile when the template gets applied to Time objects as keys, because each Node has to create a Time object as its key, expecting there to be a default (or no) constructor.

Likewise if Time was the data type for the value, **V dummy;** would fail. And the solution is *not* to go into the template and replace **V dummy;** with **V dummy(0, 0);**. Remember that **typename V** is an abstraction -- putting the zeros there means that you know something about the data type that the template is supposed to know.

A solution is to either write another constructor *or* assign a default value to each parameter in the prototype (or inline function):

```
struct Time
{
  int hour, minute;
  Time(int h=0, int m=0){hour = h; minute = m;}
};
```

That solves one problem, but it *still* won't work because of this in the for-loops: **if (data[i].key == key)** and **if (p->key == key)**.   The solution is this:

```
struct Time
{
  int hour, minute;
  Time(int h=0, int m=0){hour = h; minute = m;}
};
bool operator==(const Time& a, const Time& b){return 60 * a.hour + a.minute == 60 * b.hour + b.minute;}
```

DVC Schedule History Revisited

The DVC schedule history solution could use an associative array to track the subject-count pairs -- for example, {COMSC, 1451}. Its declaration would

The DVC schedule history solution could use an associative array to track the subject-count pairs -- for example, {COMSC, 1481}. Its declaration would be something like:

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

Then if COMSC has been seen before, you would just do **subjectCounts["COMSC"]++;**. But it no key yet exists, you'd do **subjectCounts["COMSC"]=0;**.

Applying it to duplicate checking, where we maintain a separate list for each term, the data structure we could use for the list of seen sections could be a **DynamicArray<string>**. Or to avoid having to track size with a separate int, you could do **AssociativeArray<string, bool>** where the **bool** gets set to true. Using that as the "value" part of a "key-value" pair, where the "key" is the subject (like COMSC), that data structure could be something like this:

```
AssociativeArray<string, AssociativeArray<string, bool> > alreadySeen;
```

Yes, that's a data structure object *inside* a data structure without the need for a separate struct like we used in the "big data" Module 5. Note one odd coding issue -- it's the back-to-back less-thans that end the angle-bracket container. There's a space between them, because without that space, >> is a *stream extraction operator!* In any case, here's our new "cast of characters":

- **alreadySeen[term]** is the AssociativeArray of already seen sections for the term. It's key-value pairs are the section number and the Boolean "true". If that term's not been seen yet, **alreadySeen[term]** creates it with an empty AssociativeArray<string, bool> for tracking the seen sections.
- **alreadySeen[term].containsKey[section]** not only creates all entries in the data structures, it also returns true or false if the section has been seen for that term. Usually this will be false, but if it's true, it's a duplicate.
- **alreadySeen[term][section]=true;** marks the term and section as having been seen.

See if you can figure out the simple if-else-statement that (1) checks if the term and section are already seen, and if not (2) marks them as already seem.

operator==, v.1

The example above has this stand-alone function:

```
bool operator==(const Time& a, const Time& b){return 60 * a.hour + a.minute == 60 * b.hour + b.minute;}
```

If it had been too complicated to write all on one line, we could have written like this:

```
bool operator==(const Time& a, const Time& b)
{
  int aTime = 60 * a.hour + a.minute;
  int bTime = 60 * b.hour + b.minute;
  if (aTime == bTime) return true;
  return false;
}
```

A prototype is not really necessary, so we're not using them in the examples in these modules.

operator ==, v.2

But wait -- there's more! The Time class is rather unique, because different combinations of hour, minute, and second can be equal. {1, 0, 0} equals {0, 0, 3600} equals {0, 60, 0}, for example.

For something like `struct Key {string term; string section;};` in the DVC Schedule app, you have to match each attribute *exactly*. Like this:

```
bool operator==(const Key& a, const Key& b)
{
  if (a.term != b.term) return false;
  if (a.section != b.section) return false;
  return true;
}
```