Objects As Keys In Hash Tables, Reading

There's something we take for granted in all of this -- something that's just always there, and we don't even think about it. But it's not *always* there...

How about a scheduling application -- one that tracks events by their time of day. For example:

Time of day	Event
7:00 am	Breakfast
12:30 pm	Lunch
6:30 pm	Dinner

For this we'll need: `HashTable<Time, string> eventCalendar(hashCode);`. Try that, and you'll see that every statement in the template that compares a **typename K** to another **typename K** with equals-equals all of a sudden results in a compiler error. That's because it does not know how to compare Time objects. When we were using ints and strings, no problem, because =='s defined for those. But not for Time.

As it was for our AssociativeArray, the solution is to write an **operator==** function to compare Time objects, but since we already have a hash code function, we can use it like this:

```
struct Time
{
  int h; // hour
  int m; // minutes
  int s; // seconds
};
int hashCode(const Time& t){return 3600 * t.h + 60 * t.m + t.s;}
bool operator==(const Time& a, const Time& b){return hashCode(a) == hashCode(b);}
```

The C++ STL map

The STL's associative array is called "map". It's in the "map" library, and it works a lot like our AssociativeArray and HashTable templates. But one big difference is that it does not use *equals-equals* for its key comparisons. It uses *less-than* instead. Also it does not need a hashing function, because it uses a whole different way to store its key-value pairs than we do. It's really a simple change -- like this:

```
struct Time
{
  int h; // hour
  int m; // minutes
  int s; // seconds
};
int hashCode(const Time& t){return 3600 * t.h + 60 * t.m + t.s;}
bool operator<(const Time& a, const Time& b){return hashCode(a) < hashCode(b);}
```