The following is an excerpt from Scott Meyers' new book, *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*.

Item 21: Always have comparison functions return false for equal values.

Let me show you something kind of cool. Create a set where less_equal is the comparison type, then insert 10 into the set:

```
set<int, less_equal<int> > s;          // s is sorted by "<="
s.insert(10);                          // insert the value 10
```

Now try inserting 10 again:

```
s.insert(10);
```

For this call to insert, the set has to figure out whether 10 is already present. We know that it is, but the set is dumb as toast, so it has to check. To make it easier to understand what happens when the set does this, we'll call the 10 that was initially inserted 10_A and the 10 that we're trying to insert 10_B.

The set runs through its internal data structures looking for the place to insert 10_B. It ultimately has to check 10_B to see if it's the same as 10_A. The definition of "the same" for associative containers is equivalence (see Item 19), so the set tests to see whether 10_B is equivalent to 10_A. When performing this test, it naturally uses the set's comparison function. In this example, that's operator<=, because we specified less_equal as the set's comparison function, and less_equal means operator<=. The set thus checks to see whether this expression is true:

```
!(10A <= 10B) && !(10B <= 10A)         // test 10A and 10B for equivalence
```

Well, 10_A and 10_B are both 10, so it's clearly true that $10_A <= 10_B$. Equally clearly, $10_B <= 10_A$. The above expression thus simplifies to

```
!(true) && !(true)
```

and that simplifies to

```
false && false
```

which is simply false. That is, the set concludes that 10_A and 10_B are *not* equivalent, hence *not* the same, and it thus goes about inserting 10_B into the container alongside 10_A. Technically, this action yields undefined behavior, but the nearly universal outcome is that the set ends up with *two* copies of the value 10, and that means it's not a set any longer. By using less_equal as our comparison type, we've corrupted the container! Furthermore, *any* comparison function where

equal values return true will do the same thing. Equal values are, by definition, *not* equivalent! Isn't that cool?

Okay, maybe your definition of cool isn't the same as mine. Even so, you'll still want to make sure that the comparison functions you use for associative containers always return false for equal values. You'll need to be vigilant, however. It's surprisingly easy to run afoul of this constraint.

For example, Item 20 describes how to write a comparison function for containers of string* pointers such that the container sorts its contents by the values of the strings instead of the values of the pointers. That comparison function sorts them in ascending order, but let's suppose you're in need of a comparison function for a container of string* pointers that sorts in descending order. The natural thing to do is to grab the existing code and modify it. If you're not careful, you might come up with this, where I've highlighted the changes to the code in Item 20:

```
struct StringPtrGreater:                          // highlights show how
  public binary_function<const string*,           // this code was changed
                         const string*,           // from page 89. Beware,
                         bool> {                  // this code is flawed!
  bool operator()(const string *ps1, const string *ps2) const
  {
    return !( *ps1 < *ps2);                       // just negate the old test;
  }                                               // this is incorrect!
};
```

The idea here is to reverse the sort order by negating the test inside the comparison function. Unfortunately, negating "<" doesn't give you ">" (which is what you want), it gives you ">=". And you now understand that ">=", because it will return true for equal values, is an invalid comparison function for associative containers.

The comparison type you really want is this one:

```
struct StringPtrGreater:                          // this is a valid
  public binary_function<const string*,           // comparison type for
                         const string*,           // associative containers
                         bool> {
  bool operator()(const string *ps1, const string *ps2) const
  {
    return *ps2 < *ps1;                           // return whether *ps2
  }                                               // precedes *ps1 (i.e., swap
                                                  // the order of the
};                                                // operands)
```

To avoid falling into this trap, all you need to remember is that the return value of a comparison function indicates whether one value precedes another in the sort order defined by that function. Equal values never precede one another, so comparison functions should always return false for equal values.

Sigh.

I know what you're thinking. You're thinking, "Sure, that makes sense for set and map, because those containers can't hold duplicates. But what about multiset and multimap? Those containers may contain duplicates, so what do I care if the container thinks that two objects of equal value aren't equivalent? It will store them both, which is what the multi containers are supposed to do. No problem, right?"

Wrong. To see why, let's go back to the original example, but this time we'll use a multiset:

```
multiset<int, less_equal<int> > s;          // s is still sorted by "<="
s.insert(10);                               // insert 10A
s.insert(10);                               // insert 10B
```

s now has two copies of 10 in it, so we'd expect that if we do an equal_range on it, we'll get back a pair of iterators that define a range containing both copies. But that's not possible. equal_range, its name notwithstanding, doesn't identify a range of equal values, it identifies a range of *equivalent* values. In this example, s's comparison function says that 10_A and 10_B are not equivalent, so there's no way that both can be in the range identified by equal_range.

You see? Unless your comparison functions always return false for equal values, you break all standard associative containers, regardless of whether they are allowed to store duplicates.

Technically speaking, comparison functions used to sort associative containers must define a "strict weak ordering" over the objects they compare. (Comparison functions passed to algorithms like sort (see Item 31) are similarly constrained.) If you're interested in the details of what it means to be a strict weak ordering, you can find them in many comprehensive STL references, including Josuttis' *The C++ Standard Library* [3], Austern's *Generic Programming and the STL* [4], and the SGI STL Web Site [21]. I've never found the details terribly illuminating, but one of the requirements of a strict weak ordering bears directly on this Item. That requirement is that any function defining a strict weak ordering must return false if it's passed two copies of the same value.

Hey! That *is* this Item!