How can I synthesize a C++20 three-way comparison from two-way comparisons?

devblogs.microsoft.com/oldnewthing/20220516-52

May 16, 2022

Raymond Chen

The C++20 three-way comparison operator <=> (commonly nicknamed the *spaceship operator* due to its appearance) compares two items and describes the result. It's called the three-way comparison because there are five possible results: *less*, *equal*, *equivalent*, *greater*, and *unordered*.

Yeah, the name is kind of weird.

It's called the three-way comparison because in other languages, the equivalent operator has three possible results: *less*, *equal*, and *greater*. C++20 expands the set of possible results but kept the old name. (Sound familiar?)

Ordering	Results				
strong ordering	less	equal	equivalent	greater	
weak ordering	less	equivalent		greater	
partial ordering	less	equivalent		greater	unordered

Each of the orderings can convert to the one below it, using the conversions given by the chart.

The strong ordering distinguishes between items being *equal* (identical and interchangeable) and *equivalent* (not interchangeable but close enough for some purpose). For example, two instances of the same string `"hello"` are equal, in that they represent the same string and are fully interchangeable. On the other hand, two people with the same security clearance are *equivalent* from a security perspective (they have access to the same things), but they are not *equal* (they are nevertheless different people).

When sorting, you are usually interested in equivalence, but when searching you might be interested in equality. (I'm looking for Bob, not just anybody with the same security clearance as Bob.)

Suppose you have an object from a class library that predates C++20 and doesn't support three-way comparison. You want your code to be able to take advantage of the three-way comparison should the library be updated but fall back to two-way comparison in the meantime. In other words, you want to take advantage of three-way comparison if available.

Fortunately, you don't have to write all that SFINAE nonsense, because somebody else has done it for you: `std::tuple`.

Tuples have the bonus property of supporting the three-way comparison operator, even if the underlying types do not. In the case where they do not, they will synthesize a three-way comparison from the two-way comparisons.

if `a < b`	return `less`
else if `a > b`	return `greater`
otherwise	return `equivalent`

So we can just wrap the objects inside a `std::tuple` and compare the tuples. To avoid unnecessary copies, we can wrap them as references, or use `forward_as_tuple` which always uses references.

```
std::weak_ordering
compare_3way_via_tuple(T const& a, T const& b)
{
    return std::forward_as_tuple(a) <=>
           std::forward_as_tuple(b);
}
```

It turns out that there's already a pre-made function that does something very similar: `std::compare_weak_order_fallback` also synthesize a missing three-way comparison, but it uses a different algorithm from tuples:

if `a == b`	return `equivalent`
else if `a < b`	return `less`
otherwise	return `greater`

Tuples use a different algorithm from `std::compare_weak_order_fallback`. Which one is better? Why are they different?

I suspect that tuples use a different algorithm because tuple ordering comes from C++11, which predates three-way comparison. Back in those days, the comparison operators was used mostly for sorting and other ordered-sequence type algorithms. And those algorithms require only that the objects support the `<` operator. Therefore, tuples have to make do with only the `<` operator.

On the other hand, `std::compare_weak_order_fallback` was born into the world of three-way comparisons, so it has more liberty to take dependencies on things beyond just the `<` operator.

If you know that the underlying object supports `==`, then my guess is that `std::compare_weak_order_fallback` is better, because `==` testing tends to be faster than `<` testing. For example, comparing two strings for equality can short-circuit if the strings are different lengths. This shortcut is not available for less-than comparison.

Raymond Chen

Follow