Writing a sort comparison function, part 4: descending sort

devblogs.microsoft.com/oldnewthing/20220520-00

May 20, 2022

Raymond Chen

So far, we've written sort comparison functions on the assumption that we want to sort all keys ascending. But what if you want a mix of ascending and descending?

Unfortunately, `std::tuple` does straight lexicographic ordering, so all the elements are sorted ascending. To get it to do descending sort, you'll have to do something clever.

If you just want a straight descending sort across all columns, then you can flip the direction of the top-level comparison. For example, suppose we want to sort by height descending, then width descending:

```
void f(std::vector<T>& v)
{
    auto key = [](auto&& t) {
        return std::make_tuple(t.height, t.width);
    };
    std::sort(v.begin(), v.end(), [key](auto&& t, auto&& b) {
        // reversed comparison for descending sort
        return key(a) > key(b);
    });
}
```

However, it is more likely the case that you want a mix of ascending and descending. For example, you might want descending by height, then ascending by name.

Well, if the thing being sorted descending has a natural way of reversing the order, you can apply that reversal. For example, Boolean values can be `!`'ed, and signed integers can be negated, assuming they aren't the most-negative two's complement integer.

```
void f(std::vector<T>& v)
{
    auto key = [](auto&& t) {
        // ascending by height, descending by width
        return std::make_tuple(t.height, -t.width);
    };
    std::sort(v.begin(), v.end(), [key](auto&& t, auto&& b) {
        return key(a) < key(b);
    });
}
```

For unsigned integers, bitwise negation works at reversing the sort order. And in fact, for two's complement signed integers, bitwise negation will reverse the sort order as well. So we can use

```
void f(std::vector<T>& v)
{
    auto key = [](auto&& t) {
        // ascending by height, descending by width
        return std::make_tuple(t.height, ~t.width);
        //                               ^
    };
    std::sort(v.begin(), v.end(), [key](auto&& t, auto&& b) {
        return key(a) < key(b);
    });
}
```

However, for most types, there is no obvious "reversal" transformation on the data. You'll have to reverse the comparison itself.

```
std::weak_ordering
compare_3way_for_sorting(T const& a, T const& b)
{
    auto cmp = std::compare_weak_order_fallback(a.name, b.name);
    // descending by connector (note that "a" and "b" are reversed)
    if (cmp == 0) cmp = std::compare_weak_order_fallback(b.GetConnector(),
a.GetConnector());
    if (cmp == 0) cmp =
std::compare_weak_order_fallback(LookupManufacturingDate(a.part_number),

LookupManufacturingDate(b.part_number));
    return cmp;
}
```

It's important that you leave a comment explaining that the `a` and `b` are reversed: The reversal is easily overlooked, so somebody looking at the code may not realize that we are sorting descending by connector, and somebody copying the code may not realize it either. And for readers who do notice it, you need the comment so that they don't think the reversal is a bug and try to "fix" it.

I don't see any easy way to create a "reverse-comparison" wrapper, so I'll make my own:

```
template<typename T>
struct descending_compare
{
    descending_compare(T t) : value(std::move(t)) { }
    T value;

    auto operator<=>(descending_compare const& other) const {
        return std::compare_weak_order_fallback(other.comparand(), comparand());
    }
    std::unwrap_reference_t<T> const& comparand() const {
        return value;
    }
};
```

There are a few tricks here.

First, we implement only the spaceship operator, and let the compiler autogenerate the other comparison operators from it.

Second, we use `std::compare_weak_order_fallback` to generate the three-way comparison result, even if the wrapped type supports only the two-way comparison operators.

Third, we use `unwrap_reference_t` to pull the underlying type out of the value, in case the value is a `std::reference_wrapper`. To get to the innermost value, the compiler would have to apply two user-defined conversions: One from `descending_compare<reference_wrapper<T>>` to `reference_wrapper<T>`, and then another from `reference_wrapper<T>` to `T`. But the C++ language rule for conversion chains allows only one user-defined conversion in the chain.

We can wrap a value inside a `descending_compare` to add a descending key to the sort key:

```
void f(std::vector<T>& v)
{
    auto key = [](auto&& t) {
        // ascending by height, descending by area, descending by name
        return std::make_tuple(
            t.height,
            descending_compare(t.height * t.width),
            descending_compare(std::ref(t.name)));
    };
    std::sort(v.begin(), v.end(), [key](auto&& a, auto&& b) {
        return key(a) < key(b);
    });
}
```

Unfortunately, I still see `t.name`'s `<=>` operator being called twice, so this still isn't as good as I hoped. Maybe somebody can spot what I'm doing wrong. (I think it has to do with how I declared the spaceship operator. The compiler doesn't realize that the `descending_compare` is three-way comparable, so it falls back to doing two two-way comparisons.)

Bonus chatter: It's also easy to overlook the need to wrap the field in a `std::ref` to suppress a copy. We could have a separate `descending_field` wrapper for sorting fields descending.

```
template<typename T>
struct descending_field :
    dscending_compare<std::reference_wrapper<T const>>
{
    descending_field(T const& f) :
            descending_compare<std::reference_wrapper<T const>>(std::ref(f)) {}
};
```

Bonus bonus chatter: I don't know why I had to spell out the base class `descending_compare<...>`. I expected the injected class name to allow me to write just `descending_compare`, but it didn't work.

Raymond Chen

Follow