Writing a sort comparison function, part 3: spaceships

devblogs.microsoft.com/oldnewthing/20220519-00

May 19, 2022

Raymond Chen

Last time, we wrote a multi-level sort with deferred calculation of secondary keys. Having to compare everything twice got cumbersome. We can do better with the C++20 spaceship operator.

```
// three-way comparison
std::weak_ordering
compare_3way_for_sorting(T const& a, T const& b)
{
    // First compare by name
    std::weak_ordering cmp = a.name <=> b.name;
    if (cmp != 0) return cmp;

    // Names are equal, check connector names
    cmp = a.GetConnector() <=> b.GetConnector();
    if (cmp != 0) return cmp;

    // Names and connector names are equal,
    // manufacturing date is the last check.
    cmp = LookupManufacturingDate(a.part_number) <=>
          LookupManufacturingDate(b.part_number);
    return cmp;
}

// less-than comparison
bool compare_less_for_sorting(T const& a, T const& b)
{
    // First compare by name
    std::weak_ordering cmp = a.name <=> b.name;
    if (cmp != 0) return cmp < 0;

    // Names are equal, check connector names
    cmp = a.GetConnector() <=> b.GetConnector();
    if (cmp != 0) return cmp < 0;

    // Names and connector names are equal,
    // manufacturing date is the last check.
    cmp = LookupManufacturingDate(a.part_number) <=>
          LookupManufacturingDate(b.part_number);
    return cmp < 0;
}
```

Another way of writing this is

```
// three-way comparison
std::weak_ordering
compare_3way_for_sorting(T const& a, T const& b)
{
    std::weak_ordering cmp = a.name <=> b.name;
    if (cmp == 0) cmp = a.GetConnector() <=> b.GetConnector();
    if (cmp == 0) cmp = LookupManufacturingDate(a.part_number) <=>
                        LookupManufacturingDate(b.part_number);
    return cmp;
}

// less-than comparison
bool compare_less_for_sorting(T const& a, T const& b)
{
    // First compare by name
    std::weak_ordering cmp = a.name <=> b.name;
    if (cmp == 0) cmp = a.GetConnector() <=> b.GetConnector();
    if (cmp == 0) cmp = LookupManufacturingDate(a.part_number) <=>
                        LookupManufacturingDate(b.part_number);
    return cmp < 0;
}
```

And then we can use the `std::compare_weak_order_fallback` function to synthesize a missing three-way comparison:

```
std::weak_ordering
compare_3way_for_sorting(T const& a, T const& b)
{
    auto cmp = std::compare_weak_order_fallback(a.name, b.name);
    if (cmp == 0) cmp = std::compare_weak_order_fallback(a.GetConnector(),
b.GetConnector());
    if (cmp == 0) cmp =
std::compare_weak_order_fallback(LookupManufacturingDate(a.part_number),

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

// less-than comparison
bool compare_less_for_sorting(T const& a, T const& b)
{
    auto cmp = std::compare_weak_order_fallback(a.name, b.name);
    if (cmp == 0) cmp = std::compare_weak_order_fallback(a.GetConnector(),
b.GetConnector());
    if (cmp == 0) cmp =
std::compare_weak_order_fallback(LookupManufacturingDate(a.part_number),

LookupManufacturingDate(b.part_number));
    return cmp < 0;
}
```

Next time, we'll look at descending sorts.

Bonus chatter: There is high copy/pasta error risk because the left and right of the `<=>` operator differ only by the choice of `a` or `b`. Maybe we could use this helper:

```
template<typename T, typename U = T>
struct successive_comparisons
{
    successive_comparisons(T&& left, U&& right) :
        a(left), b(right) {}
    T&& a;
    U&& b;

    template<typename Lambda>
    auto compare(Lambda&& lambda)
    { return lambda(a) <=> lambda(b); }
};

// I don't know why these deduction guides are necessary, but
// it doesn't work if I omit them.
template<typename T, typename U = T>
successive_comparisons(T&, U&) -> successive_comparisons<T&, U&>;

template<typename T, typename U = T>
successive_comparisons(T&&, U&) -> successive_comparisons<T&, U>;

template<typename T, typename U = T>
successive_comparisons(T&, U&&) -> successive_comparisons<T, U&>;

template<typename T, typename U = T>
successive_comparisons(T&&, U&&) -> successive_comparisons<T, U>;

// Avoid copy/pasta risk.
std::weak_ordering
compare_3way_for_sorting(T const& a, T const& b)
{
    auto s = successive_comparisons(a, b);

    std::weak_ordering
        cmp = s.compare([](auto&& v) { return v.name; });
    if (cmp == 0)
        cmp = s.compare([](auto&& v) { return v.GetConnector(); });
    if (cmp == 0)
        cmp = s.compare([](auto&& v) { return LookupManufacturingDate(v.part_number);
});
    return cmp;
}
```

Unfortunately, this copies the `v.name` unnecessarily since the deduced lambda return type is as if by `auto`, which is a copy, not a reference. To avoid the copy, you must ask for a reference return, which is either by naming the type explicitly or by using `decltype(auto)` with a parenthesized return value.

```
std::weak_ordering
compare_3way_for_sorting(T const& a, T const& b)
{
    auto s = successive_comparisons(a, b);

    std::weak_ordering
        cmp = s.compare([](auto&& v) -> decltype(auto) { return (v.name); });
    if (cmp == 0)
        cmp = s.compare([](auto&& v) { return v.GetConnector(); });
    if (cmp == 0)
        cmp = s.compare([](auto&& v) { return LookupManufacturingDate(v.part_number);
});
    return cmp;
}
```

I'm not sure this is an improvement. It just replaces one copy/pasta risk for another.

Also, this helper class prevents you from reusing intermediates from previous comparison steps.

```
std::weak_ordering
compare_3way_for_sorting(T const& a, T const& b)
{
    auto s = successive_comparisons(a, b);

    std::weak_ordering
        cmp = s.compare([](auto&& v) -> decltype(auto) { return (v.name); });
    if (cmp == 0)
        cmp = s.compare([](auto&& v) { return v.GetConnector(); });
    if (cmp == 0)
        cmp = s.compare([](auto&& v) { return v.GetConnector().InstallDate(); });
    return cmp;
}
```

We have to call `GetConnector()` a second time in order to look up its installation date, even though we had already done so earlier. With the explicit cascading version, we could reuse the results of the previous lookup:

```
std::weak_ordering
compare_3way_for_sorting(T const& a, T const& b)
{
    std::weak_ordering cmp = a.name <=> b.name;
    if (cmp != 0) return cmp;

    auto connectorA = a.GetConnector();
    auto connectorB = b.GetConnector();
    cmp = connectorA <=> connectorB;
    if (cmp != 0) return cmp;

    return connectorA.InstallDate() <=> connectorB.InstallDate();
}
```

Raymond Chen

Follow