Writing a sort comparison function, part 2: avoid unnecessary expense

devblogs.microsoft.com/oldnewthing/20220518-00

May 18, 2022

Raymond Chen

Last time, we wrote a basic multi-level sort. I reiterate that the best way to do this is not to write your own multi-level comparison function but rather to rely on `std::pair` or `std::tuple` to do the work for you.

It may be that calculating one of the secondary keys is expensive, and you don't want to do it unless it turns out to be necessary. In that case, you'll have to break down the comparison manually into components. But don't try to be clever about it. Just write the most obvious version:

```
// three-way comparison
int compare_3way_for_sorting(T const& a, T const& b)
{
    // First compare by name
    if (a.name < b.name) return -1;
    if (a.name > b.name) return +1;

    // Names are equal, check connector names
    auto&& a_connector = a.GetConnector();
    auto&& b_connector = b.GetConnector();

    if (a_connector.name < b_connector.name) return -1;
    if (a_connector.name > b_connector.name) return +1;

    // Names and connector names are equal,
    // check manufacturing date
    auto&& a_date = LookupManufacturingDate(a.part_number);
    auto&& b_date = LookupManufacturingDate(b.part_number);

    if (a_date < b_date) return -1;
    if (a_date > b_date) return +1;

    // All keys match
    return 0;
}

// less-than comparison
bool compare_less_for_sorting(T const& a, T const& b)
{
    // First compare by name
    if (a.name < b.name) return true;
    if (a.name > b.name) return false;

    // Names are equal, check connector names
    auto&& a_connector = a.GetConnector();
    auto&& b_connector = b.GetConnector();

    if (a_connector.name < b_connector.name) return true;
    if (a_connector.name > b_connector.name) return false;

    // Names and connector names are equal,
    // check manufacturing date
    auto&& a_date = LookupManufacturingDate(a.part_number);
    auto&& b_date = LookupManufacturingDate(b.part_number);

    if (a_date < b_date) return true;
    if (a_date > b_date) return false;

    // All keys match
    return false;
}
```

I've seen code that tried to do the multi-level comparison manually, but they were too clever and tried to cram it all into one line for style points, but messed it up. Resist the temptation to earn style points. Write the simplest, most straightforward code. Not only is it easier for humans to understand, it's also easier for the compiler to understand.

Next time, we'll look at how to do this with spaceships.

Bonus chatter: I considered embracing the "Don't do this, just use tuples" principle by creating a delayed-comparison wrapper:

```
template<typename Lambda>
struct defer_comparison
{
    defer_comparison(Lambda lambda) : key(std::move(lambda)){}
    Lambda key;

    auto operator<=>(defer_comparison const& other) const
        { return compare_3way(key(), other.key() ); }
};

auto key(T const& t)
{
    return std::make_tuple(std::ref(t.name),
                           defer_comparison([&] { return t.GetConnector(); }),
                           defer_comparison([&] { return
LookupManufacturingDate(t.part_number); }));
}

std::weak_ordering
compare_3way_for_sorting(T const& a, T const& b)
{
    return key(a) <=> key(b);
}

bool compare_less_for_sorting(T const& a, T const& b)
{
    return key(a) < key(b);
}
```

However, this generates unnecessary calls to `GetConnector()` and `Lookup-ManufacturingsDate()` because it breaks down as

```
bool compare_less_for_sorting(T const& a, T const& b)
{
    auto a_tuple = key(a);
    auto b_tuple = key(b);

    if (std::get<0>(a) < std::get<0>(b)) return true;
    if (std::get<0>(a) > std::get<0>(b)) return false;

    if (std::get<1>(a) < std::get<1>(b)) return true;
    if (std::get<1>(a) > std::get<1>(b)) return false;

    if (std::get<2>(a) < std::get<2>(b)) return true;
    if (std::get<2>(a) > std::get<2>(b)) return false;

    return false;
}
```

In our case, `std::get<1>()` returns a `defer_comparison`, so we end up comparing the same two `defer_comparison` objects twice.

We could try to solve this by using a `std::async` with deferred execution to memoize the result of the lambda, but this introduces extra memory allocations and virtual function tables and memory barriers, so it feels like we'd be heading in the wrong direction.

Raymond Chen

Follow