Writing a sort comparison function, part 1: basics

devblogs.microsoft.com/oldnewthing/20220517-00

May 17, 2022

Raymond Chen

I've noted in the past that a sort comparison function must follow certain rules, and if you violate those rules, very strange things happen. So what are some patterns for writing sort comparison functions that don't break the rules?

Most of the time, sorting can be reduced to key comparison:[1] From each element being sorted, you generate a *sort key*, and those sort keys are compared against each other.[2]

Related reading: Sorting by indices, part 2: The Schwartzian transform.

The easiest way to write a sort comparison function is to generate the sort keys, and then compare the keys.

```
// Assume some function called "key" that produces
// the sort key from an object.

// three-way comparison
int compare_3way_for_sorting(T const& a, T const& b)
{
    int a_key = key(a);
    ant b_key = key(b);

    if (a_key < b_key) return -1;
    if (a_key > b_key) return +1;
    return 0;
}

// less-than comparison
bool compare_less_for_sorting(T const& a, T const& b)
{
    return key(a) < key(b);
}
```

This reduces the problem to writing a sort key. Here's an example:

```
// Key generator: Sort by total cost (price + tax)
auto key(T const& t)
{
    return t.price + t.tax;
}
```

For a multi-level sort, you can return a `std::pair` or `std::tuple` with the most significant key as the first element.

```
// Key generator: Sort by length, then by width
auto key(T const& t)
{
    return std::make_pair(t.length, t.width);
}

// Key generator: Sort by length, then by width,
// then by weight
auto key(T const& t)
{
    return std::make_tuple(t.size, t.width, t.weight);
}
```

The `make_tuple` function copies its arguments, which you may want to avoid if the members are expensive to copy. One solution is to wrap the member in a `std::ref`:

```
// Key generator: Sort by name, then age
auto key(T const& t)
{
    return std::make_tuple(std::ref(t.name), t.age);
}
```

Another is to use `std::forward_as_tuple` to make everything a reference.

```
// Key generator: Sort by name, then age
auto key(T const& t)
{
    return std::forward_as_tuple(t.name, t.age);
}
```

The `std::forward_as_tuple` approach is dangerous, however, if a portion of the key comes from a temporary:

```
// Key generator: Sort by name, then area
auto key(T const& t)
{
    // Don't do this!
    return std::forward_as_tuple(t.name, t.height * t.width);
}
```

We saw some time ago that `std::forward_as_tuple` captures everything as a reference, so it will capture the second component as a reference to a temporary integer, and that temporary integer will be destroyed at the end of the full expression, leaving the reference dangling.

If you inline the call to `std::forward_as_tuple` into `compare_less_for_sorting`, then the temporary will survive past the comparison:

```
bool compare_less_for_sorting(T const& a, T const& b)
{
    return std::forward_as_tuple(a.name, a.height * a.width) <
    return std::forward_as_tuple(b.name, b.height * b.width);
}
```

But this is itself a violation of the principle of Don't Repeat Yourself (DRY). So I guess that means `make_tuple` with judicious use of `std::ref` is the way to go.

To avoid making the key generator visible outside the scope of the comparison function, you can declare it as a local captureless lambda:

```
bool compare_less_for_sorting(T const& a, T const& b)
{
    auto key = [](auto&& t) {
        return std::make_tuple(std::ref(t.name), t.height * t.width);
    };
    return key(a) < key(b);
}
```

And then you can lambda-ize the comparison function, too:

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

Note that even if you limit your use of `forward_as_tuple` to references to members of the objects being sorted, you have to consume the forwarded tuple immediately. Sorting may move the object, and once that happens, your forwarded references become invalid.

Next time, we'll look at what we could do if one of the sort keys is expensive to calculate.

[1] And in fact, if your sorting does *not* reduce to key comparison, then it's a clue that your sort function might not be valid.

[2] Indeed, in the Python programming language, this is the *only* way to sort. The ability to provide a custom comparison function was removed in Python 3. Instead, you pass a function that produces a sort key, and the sort occurs on the keys.

Raymond Chen

Follow