Sorting by indices, part 2: The Schwartzian transform

devblogs.microsoft.com/oldnewthing/20170106-00

January 6, 2017

Raymond Chen

Before we dig into the Schwartzian transform, let's look at a more conventional generic way to sort by a key:

```
template<typename Iter, typename UnaryOperation, typename Compare>
void sort_by(Iter first, Iter last, UnaryOperation op, Compare comp)
{
  std::sort(first, last,
            [&](T& a, T& b) { return comp(op(a), op(b)); });
}
```

The idea here is that you give a unary operator `op` that produces a sort key, and we sort the items by that key according to the comparer. For example, you might say

```
std::vector<Person> v = ...;

// Sort by last name
sort_by(v.begin(), v.end(),
        [](const Person& p) { return p.LastName; },
        std::less<std::string>);
```

The first functional selects the thing we are sorting by (here, the last name), and the second functional selects how we are sorting (here, in ascending order).

This technique works okay if the unary operator (the key generator) is simple, such as the one we have here. But if generating the key is expensive, then we will want to cache the keys rather than evaluating them over and over. So let's do it:

```
template<typename Iter, typename UnaryOperation, typename Compare>
void sort_by_with_caching(Iter first, Iter last, UnaryOperation op, Compare comp)
{
 using Diff = std::iterator_traits<Iter>::difference_type;
 using T = std::iterator_traits<Iter>::value_type;
 using Key = decltype(op(std::declval<T>()));
 using Pair = std::pair<T, Key>;
 Diff length = std::distance(first, last);
 std::vector<Pair> pairs;
 pairs.reserve(length);
 std::transform(first, last, std::back_inserter(pairs),
                [&](T& t) { return std::make_pair(std::move(t), op(t)); });
 std::sort(pairs.begin(), pairs.end(),
           [&](const Pair& a, const Pair& b) { return comp(a.second, b.second); });
 std::transform(pairs.begin(), pairs.end(), first, [](Pair& p) { return
std::move(p.first); });
}
```

The above is the literal translation of the Schwartian transform (also known more conventionally as the decorate-sort-undecorate pattern) into C++. You augment each item to be sorted with its corresponding key.[1] You then sort by the key. And then you throw away the keys, leaving the original items.[1]

We use `std::move` to move the items out of the original collection into our temporary vector of pairs, then we sort the pairs by the key, and then we move the items from our pairs back to the original collection. The hope is that the object is efficiently movable, so these move operations are very inexpensive.

But maybe the objects being sorted isn't efficiently movable. Or maybe (horrors) the keys aren't efficiently movable. We can use the trick from the `sort_minimize_copies` function to sort the items with minimal moving.

```
template<typename Iter, typename UnaryOperation, typename Compare>
void sort_by_with_caching(Iter first, Iter last, UnaryOperation op, Compare comp)
{
 using Diff = std::iterator_traits<Iter>::difference_type;
 using T = std::iterator_traits<Iter>::value_type;
 using Key = decltype(op(std::declval<T>()));
 Diff length = std::distance(first, last);
 std::vector<Key> keys;
 keys.reserve(length);
 std::transform(first, last, std::back_inserter(keys),
                [&](T& t) { return op(t); });
 std::vector<Diff> indices(length);
 std::iota(indices.begin(), indices.end(), static_cast<Diff>(0));
 std::sort(indices.begin(), indices.end(),
           [&](Diff a, Diff b) { return comp(keys[a], keys[b]); });
 apply_permutation(first, last, indices.begin());
}

template<typename Iter, typename UnaryOperation>
void sort_by_with_caching(Iter first, Iter last, UnaryOperation op)
{
 sort_by_with_caching(first, last, op, std::less<>());
}
```

We create two helper arrays. One holds the keys corresponding to the items, and the other holds the indices. The keys are in a parallel array with the original collection and do not move during sorting. Instead, we sort the indices. Once we finish the sort, we apply the permutation to the original items to move them to their final positions.

Okay, so that's what I was trying to get at: Sorting a vector by a key, with caching. If there's already a standard function to do this, please let me know.[3]

[1] The algorithm does assume that the key can consistently be generated from the item, and in particular that it depends only on the item and not on the item with which it is being compared.

[2] If we wanted to show off `sort_by`, the call to `std::sort` could have been replaced with

```
 sort_by(pairs.begin(), pairs.end(),
         [](const Pair& p) { return p.second; }, comp);
```

[3] I would like to point out that I arrived at this particular algorithm only after going down a dead end of having only a parallel key array. The idea was that I would sort the items and keys together by using a proxy iterator that represented both the original item and its key. The thing I had trouble working out was how to structure the proxy iterator so that it knew when its contents had been moved out, so it could move the real objects. I probably could have gotten it to work eventually, but then I realized I could avoid the entire hassle by sorting indices instead.

Raymond Chen

Follow