Applying a permutation to a vector, part 2

devblogs.microsoft.com/oldnewthing/20170103-00

January 3, 2017

Raymond Chen

We left off our study of the `apply_permutation` function by wondering which version is bettern: the moving version of the swapping version. I'm not certain I have the answer, but here's my analysis.

The first observation is that the standard swap function performs three move operations. It basically goes like this:

```
template<class T>
void std::swap(T& a, T& b)
{
 T t{std::move(a));
 a = std::move(b);
 b = std::move(t);
}
```

So if you're counting move operations, you need to count a swap as three moves.

But wait, you say. It is legal for types to provide a custom swap operation. However, even those custom swap operations are still going to perform three move operations.[1] The customization is just to reduce the memory requirements. While the standard swap will move the entire instance into a temporary, a custom swap will move individual members.

```
struct sample
{
 std::string x, y, z;
};
```

In the above example, assuming the obvious definition of the move assignment operator, the standard swap would move all three strings from the first `sample` into a temporary `sample`, then move the three strings from the second `sample` into the first, and then move the three strings from the temporary `sample` into the second. But a custom swap would look like this:

```
void swap(sample& a, sample& b)
{
 swap(a.x, b.x);
 swap(a.y, b.y);
 swap(a.z, b.z);
}
```

This version performs three swaps consecutively. The total number of move operations is the same; they just happen in a different order.

standard swap	custom swap

`t.x = std::move(a.x);`
`t.y = std::move(a.y);`
`t.z = std::move(a.z);`
`a.x = std::move(b.x);`
`a.y = std::move(b.y);`
`a.z = std::move(b.z);`
`b.x = std::move(t.x);`
`b.y = std::move(t.y);`
`b.z = std::move(t.z);`
	`t   = std::move(a.x);`
	`a.x = std::move(b.x);`
	`b.x = std::move(t);`
	`t   = std::move(a.y);`
	`a.y = std::move(b.y);`
	`b.y = std::move(t);`
	`t   = std::move(a.z);`
	`a.z = std::move(b.z);`
	`b.z = std::move(t);`

The member-by-member swap of the custom swap function will probably exhibit better locality than the full-class swap used by the standard swap. The member-by-member swap also requires fewer temporary resources than the full-class swap (here: one string compared to three).

Okay, so either way, a swap costs three moves. Therefore, if we are just counting moves, the swapping version of `apply_permutation` performs almost three times as many move operations as the explicit-temporary version.

The counter-argument to “too many move operations” is that move operations are relatively inexpensive. A typical move operation transfers ownership of resources from one instance to another. No new allocation needs to be done; the existing allocation just needs to be transferred across. So counting your move operations is like counting pennies: Even if you manage to save a hundred of them, that’s still only one dollar.

But I think the winning argument for moving rather than swapping is the copyable-but-not-movable object. If the object doesn’t have a move constructor/move assignment operator, but it does have a copy constructor/copy assignment operator, then the algorithm will still work, but it will fall back to using the copy operation in the absence of a move operation.

And copy operations are not cheap.

So now instead of saving pennies, we are saving dollars, and those dollars quickly add up. So this argues in favor of the moving version rather than the swapping version.[2]

Like I said at the start, this is my analysis. It could be wrong. Let me know.

Next time, we'll look at how this function could be generalized.

[1] Yes, there may be super-optimized custom swaps that are actually perform less work than the standard swap, but I think those types of custom swaps are relatively uncommon.

[2] On the third hand (fourth, fifth? how many am I up to?) if the object is copyable but not movable, but it also has a custom swap function, then that swap function is going to be much less expensive than copying. (Because the custom swap function is going to exchange contents rather than making three expensive copies.) You'll encounter objects of this ilk if they predate C++11, since it is C++11 that introduced the concept of movability. So now, if you have an object that is copyable, efficiently swappable, and not movable, you are better off using the swapping version again. Another case where the swapping version may be better is if the vector uses a proxy iterator, such as `vector<bool>`.

So now I'm not sure any more. Maybe the way to go is to do compile-time detection of whether the object has a custom swap function. If so, then use the swapping version. If not, then use the moving version.

Raymond Chen

Follow