Applying a permutation to a vector, part 1

devblogs.microsoft.com/oldnewthing/20170102-00

January 2, 2017

Raymond Chen

Suppose you have a vector `indices` of *N* integers that is a permutation of the numbers 0 through *N* – 1. Suppose you also have a vector `v` of *N* objects. The mission is to apply the permutation to the vector. If we let `v2` represent the contents of the vector at the end of the operation, the requirement is that `v2[i] = v[indices[i]]` for all `i`.

For example, if the objects are strings, and the vector of objects is `{ "A", "B", "C" }` and the vector of integers is `{ 1, 2, 0 }`, then this means the output vector is `{ "B", "C", "A" }`.

This sounds like something that would be in the standard library, but I couldn't find it, so I guess we'll have to write it ourselves. (If you find it in the standard library, let me know!)

Let's start with the easy version: You don't update the vector in place. Instead, you produce a new vector. Solving this version is fairly straightforward, because all you have to do is write the problem statement!

```
template<typename T>
std::vector<T>
apply_permutation(
    const std::vector<T>& v,
    const std::vector<int>& indices)
{
 std::vector<T> v2(v.size());
 for (size_t i = 0; i < v.size(); i++) {
  v2[i] = v[indices[i]];
 }
 return v2;
}
```

Now we get to the analysis, which is where the fun is.

The above algorithm assumes that `T` is default-constructible, because we create a vector `v2` of default-constructed `T` objects, and then replace them one by one with copies of the real `T` objects from `v`. I wrote it that way to make the solution match the problem

statement, making its correctness easy to verify by inspection. You can even write a unit test for this function, so that you can verify that the transformations we are going to perform don't break the fundamental algorithm.

First, let's get rid of the requirement that there be a default constructor for `T`. All we'll require is that `T` be copy-constructible.

```
template<typename T>
std::vector<T>
apply_permutation(
    const std::vector<T>& v,
    const std::vector<int>& indices)
{
 std::vector<T> v2;
 v2.reserve(v.size());
 for (size_t i = 0; i < v.size(); i++) {
  v2.push_back(v[indices[i]]);
 }
 return v2;
}
```

But still, we're copying the elements around. Let's refine the problem statement so that instead of returning a new vector, we mutate the vector in place. That way, it'll work with movable objects, too.[1]

```
template<typename T>
void
apply_permutation(
    std::vector<T>& v,
    const std::vector<int>& indices)
{
 std::vector<T> v2;
 v2.reserve(v.size());
 for (size_t i = 0; i < v.size(); i++) {
  v2.push_back(std::move(v[indices[i]]));
 }
 v = std::move(v2);
}
```

This version is basically the same thing as before, except that we move the objects around instead of copying them. But this kind of misses the point of the exercise: We didn't really update the vector in place. We created a second vector and then swapped it in. Can we do it without having to create a second vector? After all, that vector might be large (either because `T` is large or N is large, or both). Actually, let's go even further: Can we do it in $O(1)$ space?

At this point, I needed to pull out a piece of paper and do some sketching. My first idea[2] led nowhere, but eventually I found something that worked: The idea is that we look at `indices[0]` to see what item should be moved into position zero. We move that item in,

but we save the item that was originally there. Then we look at what item should be moved into the place that was just vacated, and move that item into that place. Repeat until the item that was originally in position zero is the one that gets placed.

This completes one cycle, but there could be multiple cycles in the permutation, so we need to look for the next slot that hasn't yet been processed. We'll have to find a place to keep track of which slots have already been processed, and we cannot allocate a vector for it, because that would be $O(N)$ in space.

So we pull a trick: We use the indices vector to keep track of itself.

```
template<typename T>
void
apply_permutation(
    std::vector<T>& v,
    std::vector<int>& indices)
{
 for (size_t i = 0; i < indices.size(); i++) {
  T t{std::move(v[i])};
  auto current = i;
  while (i != indices[current]) {
   auto next = indices[current];
   v[current] = std::move(v[next]);
   indices[current] = current;
   current = next;
  }
  v[current] = std::move(t);
  indices[current] = current;
 }
}
```

Okay, what just happened here?

We start by "picking up" the `i`'th element (into a temporary variable `t`) so we can create a hole into which the next item will be moved. Then we walk along the cycle using `current` to keep track of where the hole is, and `next` to keep track of the item that will move into the hole. If the next item is not the one in our hand, then move it in, and update `current` because the hole has moved to where the next item was. When we finally cycle back and discover that the hole should be filled with the item in our hand, we put it there.

The way we remember that this slot has been filled with its final item is by setting its index to a sentinel value. One option would be to use −1, since that is an invalid index, and we can filter it out in our loop.

But instead, I set the index to itself. That makes the item look like a single-element cycle, and processing a single-element cycle has no effect, since nothing moves anywhere. This avoids having an explicit sentinel test. We merely chose our sentinel to be a value that already does

nothing.

Now, one thing about this algorithm is that processing a single-element cycle does require us to pick up the element, and then put it back down. That's two move operations that could be avoided. Fixing that is easy: Don't pick up the item until we know that we will need to vacate the space.

```
template<typename T>
void
apply_permutation(
    std::vector<T>& v,
    std::vector<int>& indices)
{
 for (size_t i = 0; i < indices.size(); i++) {
  if (i != indices[current]) {
   T t{std::move(v[i])};
   auto current = i;
   while (i != indices[current]) {
    auto next = indices[current];
    v[current] = std::move(v[next]);
    indices[current] = current;
    current = next;
   }
   v[current] = std::move(t);
   indices[current] = current;
  }
 }
}
```

Another trick we can use to avoid having to have a temporary item `t` is realizing that instead of holding the item in your hand, you can just put the item *in the hole*.

```
template<typename T>
void
apply_permutation(
    std::vector<T>& v,
    std::vector<int>& indices)
{
 using std::swap; // to permit Koenig lookup
 for (size_t i = 0; i < indices.size(); i++) {
  auto current = i;
  while (i != indices[current]) {
   auto next = indices[current];
   swap(v[current], v[next]);
   indices[current] = current;
   current = next;
  }
  indices[current] = current;
 }
}
```

Note that since the item that used to be in our hand is now in the hole, we don't need the manual last step of moving the item from our hand to the hole. It's already in the hole!

Okay, so is it better to move or to swap? This is an interesting question. We'll pick this up next time.

[1] Note that by making this operate on movable objects, we lose the ability to operate on objects which are not MoveAssignable. This is not a significant loss of amenity, however, because most of the algorithms in the standard library which rearrange elements within a container already require MoveAssignable, and many of them also require Swappable.

[2] My first idea was to start with the first position, swap the correct item into that first position, and update the indices so that what remained was a permutation of the remaining items that preserved the intended final result. I could then repeat the process with each subsequent position, and when I had finished walking through the entire vector, every item would be in the correct final position. I figured I could use the last position as a sort of scratchpad, similar to the "hole" we ended up using in our final algorithm. I struggled for a while trying to find the correct cleverly-chosen two-way or three-way swap among the current position, the last position, and *somebody* that would get me to the desired exit condition. I got hung up on this line of investigation and had to walk away from the problem for a little while, and then come back to it with a fresh mind.

I mention this because a lot of the time, these articles which explain how to solve a problem don't talk much about the attempted solutions that didn't work. Instead, they give the impression that the author got it right the first time. In reality, the author got it wrong the first few times, too. The reason you don't see a lot of writing about the failures is that the usual order of operations is (1) find solution, (2) write about it. You usually find the solution first, and only then do you start writing about it. As a result, you don't have very good documentation for your failures.

Raymond Chen

Follow