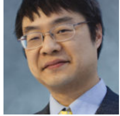
Applying a permutation to a vector, part 6

January 11, 2017



Raymond Chen

I left an exercise to write a function `apply_reverse_permutation` in which each element in the `indices` represents where the element should move *to* rather than where it comes *from*.

This exercise is easier than the forward permutation case[1] because we can maintain a very simple invariant: At all times, the state of the input variables describe the same result. All we do is take a step closer to that result at each swap.

```
template<typename Iter1, typename Iter2>
void
apply_reverse_permutation(
    Iter1 first,
    Iter1 last,
    Iter2 indices)
{
 using std::swap;
 using T = typename std::iterator_traits<Iter1>::value_type;
 using Diff = typename std::iterator_traits<Iter2>::value_type;
 Diff length = std::distance(first, last);
 for (Diff i = 0; i < length; i++) {
  while (i != indices[i]) {
   Diff next = indices[i];
   swap(first[i], first[next]);
   swap(indices[i], indices[next]);
  }
 }
}
```

The idea here is that we have a "working position" that starts at the beginning of the collection. We study the element in the working position and move it to its final destination by swapping it with whatever happens to be there right now. We also swap the bookkeeping so that we also remember where that swapped element needs to go eventually. As a result of the swap, you now have a new element in the working position. Repeat as long as element in the working position is not in the correct position. If you have a proper permutation, then

eventually you will reach the end of the cycle and the element in the working position is one that wants to be there. You can now move the working position to the next position until you have moved through the entire collection.

The complexity of this algorithm is $O(N)$ because each swap operation moves one element to its final destination, and no element appears on the left hand side of a swap operation more than once. (An element may end up swapping at most twice. Once when it swaps out of its old position into the working position, and again when it swaps out of the working position to its final position.)

Fans of tail recursion can rewrite this function tail-recursively, which might be instructive. (Or it might not. At least it'll be fun, if rewriting functions to be tail-recursive is your idea of fun.)

As before, we can add error checking while preserving the same useful exit conditions: If an error occurs, the original collection and the indices are permuted in an unspecified way.

As before, there are two error cases. One is that an index is out of range. That's easy to check. The other is that an index appears more than once. This could be detected in a number of ways. One way is to detect that we have swapped more than `length` items through the working position, because the length of a cycle in a permutation cannot exceed the number of elements. But I'm going to use the same technique we used for the forward permutation: When we realize that we are about to swap an element that has already been swapped into position. In other words, if the element at the destination already thinks that it's at the correct destination, then we have an error because two elements both want to go to the same destination.

```
template<typename Iter1, typename Iter2>
void
apply_reverse_permutation(
    Iter1 first,
    Iter1 last,
    Iter2 indices)
{
 using T = typename std::iterator_traits<Iter1>::value_type;
 using Diff = typename std::iterator_traits<Iter2>::value_type;
 Diff length = std::distance(first, last);
 for (Diff i = 0; i < length; i++) {
  while (i != indices[i]) {
   Diff next = indices[i];
   if (next < 0 || next >= length) {
    throw std::range_error("Invalid index in permutation");
   }
   if (next == indices[next]) {
    throw std::range_error("Not a permutation");
   }
   swap(first[i], first[next]);
   swap(indices[i], indices[next]);
  }
 }
}
```

[1] This is different from the forward permutation, where the work of rotating the elements through a cycle leaves the inputs in a temporarily unstable state. We saw last time that before we could report an error, we had to restore some state before reporting an error, and that state that we restored didn't even correspond meaningfully to the intermediate state. It merely corresponded to the original state in a very weakly-specified way.

Raymond Chen

Follow