Applying a permutation to a vector, part 5

devblogs.microsoft.com/oldnewthing/20170110-00

January 10, 2017

Raymond Chen

Our `apply_permutation` function assumes that the integers form a valid permutation. Let's add error checking.

There are two ways the integers could fail to be a permutation: One is that the collection includes a value that is out of range. The other problem case is that all the values are in range, but a value appears more than once. We can detect that when we encounter a single-element cycle when we expected a longer cycle. (Another way of looking at it is that we detect the error when we discover that we're about to move an item for the second time, because the permutation application algorithm is supposed to move each item only once.)

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

(I added the `typename` keyword at the suggestion of commenter ildjarn. And I used `std::distance` to calculate the distance between two iterators. The second change was not technically necessary because `std::distance` is defined as subtraction when the iterators are random-access, but if you're going to go with the standard library, you may as well go all the way, right?)

I switched to the swapping version of the algorithm because that allows me to ensure a useful exit condition in the case of exception: If an exception occurs, the elements in `[first, last)` have been permuted in an unspecified manner. Even though the resulting order is unspecified, you at least know that no items were lost. It's the same set of items, just in some other order. The indices, on the other hand, are left in an unspecified state. They won't be a permutation of the original indices.

But wait, we can even restore the `indices` to a permutation of their former selves:[1] We can take the duplicate index and drop it back into `indices[i]`. That entry optimistically was set to the value we expected to find when we reached the end of the cycle. If we never find that value, then we can put the value we actually found into that slot, thereby correcting our optimistic assumption.

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

[1] This is valuable because it improves post-mortem debuggability: You can inspect the `indices` to look for the out-of-range or duplicate index.

Raymond Chen

Follow