Add std::apply_permutation algorithm

Document #: P0000
Date: 2019-09-06
Project: Programming Language C++
Audience: LEWGI
Reply-to: Alexander Zaitsev <zamazan4ik@tut.by, zamazan4ik@gmail.com>

1 Revision history

- R0 – Initial draft

2 Issues

- `apply_reverse_permutation` name is a little bit ugly. Suggestions are welcomed.

3 Motivation

4 Proposed wording

Add to [**alg.modifying.operations**] **25.6** new subsection [**alg.permute**] **Permute 25.6.15** after [**alg.shift**] **25.6.14**:

```
template<class RandomAccessIterator1, class RandomAccessIterator2>
  constexpr void apply_permutation(RandomAccessIterator1 item_first,
                                   RandomAccessIterator1 item_last,
                                   RandomAccessIterator2 ind_first,
                                   RandomAccessIterator2 ind_last);
template<class ExecutionPolicy, class RandomAccessIterator1, class RandomAccessIterator2>
  constexpr void apply_permutation(ExecutionPolicy&& exec,
                                   RandomAccessIterator1 item_first,
                                   RandomAccessIterator1 item_last,
                                   RandomAccessIterator2 ind_first,
                                   RandomAccessIterator2 ind_last);

template<random_access_iterator I1, random_access_iterator I2, sentinel_for<I1> S1,
         sentinel_for<I2> S2, class Proj = identity>
  constexpr I1 ranges::apply_permutation(I1 item_first, S1 item_last, I2 ind_first, S2 ind_last,
                                         Proj proj = {});
template<random_access_range R1, random_access_range R2, class Proj = identity>
```

```
  constexpr safe_iterator_t<R1> ranges::apply_permutation(R1&& r1, R2&& r2, Proj proj = {});
```

Let proj be identity for the overloads with no parameter named proj.

Effects: Reorders the elements in the range [item_first, item_last) according to the order sequence [ind_first, ind_last). Every value in order sequence means where the item comes from.

Requires: For the overloads in namespace std, RandomAccessIterator1 shall meet the Cpp17ValueSwappable requirements. Order sequence needs to be exactly a permutation of the sequence [0, 1, ... , N], where N is the biggest index in the item sequence (zero-indexed).

Returns: item_last, for the overloads in namespace ranges.

Complexity: Linear.

Note: Order sequense gets permuted.

```
template<class RandomAccessIterator1, class RandomAccessIterator2>
  constexpr void apply_reverse_permutation(RandomAccessIterator1 item_first,
                                           RandomAccessIterator1 item_last,
                                           RandomAccessIterator2 ind_first,
                                           RandomAccessIterator2 ind_last);
template<class ExecutionPolicy, class RandomAccessIterator1, class RandomAccessIterator2>
  constexpr void apply_permutation(ExecutionPolicy&& exec,
                                   RandomAccessIterator1 item_first,
                                   RandomAccessIterator1 item_last,
                                   RandomAccessIterator2 ind_first,
                                   RandomAccessIterator2 ind_last);

template<random_access_iterator I1, random_access_iterator I2,
         sentinel_for<I1> S1, sentinel_for<I2> S2, class Proj = identity>
  constexpr I1 ranges::apply_reverse_permutation(I1 item_first, S1 item_last,
                                                 I2 ind_first, S2 ind_last, Proj proj = {});
template<random_access_range R1, random_access_range R2, class Proj = identity>
  constexpr safe_iterator_t<R1> ranges::apply_reverse_permutation(R1&& r1, R2&& r2,
                                                                  Proj proj = {});
```

Let proj be identity for the overloads with no parameter named proj.

Effects: Reorders the elements in the range [item_first, item_last) according to the order sequence [ind_first, ind_last). Every value in order sequence means where the item goes to.

Requires: For the overloads in namespace std, RandomAccessIterator1 shall meet the Cpp17ValueSwappable requirements. Order sequence needs to be exactly a permutation of the sequence [0, 1, ... , N], where N is the biggest index in the item sequence (zero-indexed).

Returns: item_last, for the overloads in namespace ranges.

Complexity: Linear.

Note: Order sequense gets permuted.

5 Examples

Given the containers: `std::vector<int> emp_vec, emp_order, one{1}, one_order{0}, two{1,2}, two_order{1,0}, vec{1, 2, 3, 4, 5}, order{4, 2, 3, 1, 0}`, then

```
apply_permutation(emp_vec, emp_order))  // no changes
apply_reverse_permutation(emp_vec, emp_order))  // no changes
apply_permutation(one, one_order)  // no changes
apply_reverse_permutation(one, one_order)  // no changes
apply_permutation(two, two_order)  // two:2,1
apply_reverse_permutation(two, two_order)  // two:2,1
apply_permutation(vec, order)  // vec:5, 3, 4, 2, 1
apply_reverse_permutation(vec, order)  // vec:5, 4, 2, 3, 1
```

6 Possible implementation

Possible implementation (without version with Execution policy) can be found in Boost.Algorithm: GitHub. Documentation can be found here: Boost. Available in Boost.Algorithm since Boost 1.68. Original implementation and ideas from Microsoft blog: 1, 2, 3, 4, 5, 6.