Document number:	P1243R3
Date:	2020, Jan. 9
Author:	Dan Raviv <dan.raviv@gmail.com>
Audience:	LWG

Rangify New Algorithms

I. Motivation and Scope

This paper complements P0896 by adding rangified overloads for some of the non-parallel additions to `<algorithm>` since C++14, from whence the Ranges TS took its algorithms: `for_each_n`, `clamp`, `sample`, `shift_left`, `shift_right`.

The paper does *not* provide rangified overloads for the rest of the additions to `<algorithm>` since C++14: `lexicographical_compare_three_way, search(`*range, searcher*`)`.

The paper's wording also integrates the changes in P1233R1 by Ashley Hedberg, Matt Calabrese and Bryce Adelstein Lelbach.

II. Impact On the Standard

This is a pure library extension of the Standard.

III. Proposed Wording

Header `<algorithm>` synopsis [algorithm.syn]

// [alg.foreach], for each

[…]

```
template<class InputIterator, class Size, class Function>
  constexpr InputIterator for_each_n(InputIterator first, Size n, Function
f);
template<class ExecutionPolicy, class ForwardIterator, class Size, class
Function>
  ForwardIterator for_each_n(ExecutionPolicy&& exec, // see
[algorithms.parallel.overloads]
                             ForwardIterator first, Size n, Function f);
namespace ranges {
  template<input_iterator I, class Proj = identity,
indirectly_unary_invocable<projected<I, Proj>> Fun>
    constexpr for_each_result<I, Fun>
      for_each_n(I first, iter_difference_t<I> n, Fun f, Proj proj = {});
}
```

[…]

```
// [alg.random.sample], sample
template<class PopulationIterator, class SampleIterator,
         class Distance, class UniformRandomBitGenerator>
  SampleIterator sample(PopulationIterator first, PopulationIterator last,
                        SampleIterator out, Distance n,
                        UniformRandomBitGenerator&& g);
namespace ranges {
  template<input_iterator I, sentinel_for<I> S, weakly_incrementable O, class
Gen>
    requires (forward_iterator<I> || random_access_iterator<O>) &&
      indirectly_copyable<I, O> &&
      uniform_random_bit_generator<remove_reference_t<Gen>>
    O sample(I first, S last, O out, iter_difference_t<I> n, Gen&& g);

  template<input_range R, weakly_incrementable O, class Gen>
    requires (forward_range<R> || random_access_iterator<O>) &&
      indirectly_copyable<iterator_t<R>, O> &&
      uniform_random_bit_generator<remove_reference_t<Gen>>
    O sample(R&& r, O out, range_difference_t<R> n, Gen&& g);
}
```

[…]

```
// [alg.shift], shift
template<class ForwardIterator>
  constexpr ForwardIterator
    shift_left(ForwardIterator first, ForwardIterator last,
               typename iterator_traits<ForwardIterator>::difference_type n);
template<class ExecutionPolicy, class ForwardIterator>
  ForwardIterator
    shift_left(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
               ForwardIterator first, ForwardIterator last,
               typename iterator_traits<ForwardIterator>::difference_type n);
namespace ranges {
  template<permutable I, sentinel_for<I> S>
    constexpr subrange<I>
      shift_left(I first, S last, iter_difference_t<I> n);
  template<forward_range R>
    requires permutable<iterator_t<R>>
    constexpr safe_subrange_t<R>
      shift_left(R&& r, range_difference_t<R> n);
}
template<class ForwardIterator>
  constexpr ForwardIterator
    shift_right(ForwardIterator first, ForwardIterator last,
                typename iterator_traits<ForwardIterator>::difference_type
n);
template<class ExecutionPolicy, class ForwardIterator>
  ForwardIterator
    shift_right(ExecutionPolicy&& exec, // see [algorithms.parallel.overloads]
                ForwardIterator first, ForwardIterator last,
                typename iterator_traits<ForwardIterator>::difference_type
n);
namespace ranges {
  template<permutable I, sentinel_for<I> S>
    constexpr subrange<I>
      shift_right(I first, S last, iter_difference_t<I> n);
  template<forward_range R>
    requires permutable<iterator_t<R>>
    constexpr safe_subrange_t<R>
      shift_right(R&& r, range_difference_t<R> n);
}
```

[…]

```
// [alg.clamp], bounded value
template<class T>
  constexpr const T& clamp(const T& v, const T& lo, const T& hi);
template<class T, class Compare>
  constexpr const T& clamp(const T& v, const T& lo, const T& hi, Compare
comp);
namespace ranges {
  template<class T, class Proj = identity,
indirect_strict_weak_order<projected<const T*, Proj>> Comp = ranges::less>
    constexpr const T& clamp(const T& v, const T& lo, const T& hi, Comp comp
= {}, Proj proj = {});
}
```

For each [alg.foreach]

[…]

Remarks: If `f` returns a result, the result is ignored. Implementations do not have the freedom granted under [algorithms.parallel.exec] to make arbitrary copies of elements from the input sequence.

```
template<input_iterator I, class Proj = identity,
indirectly_unary_invocable<projected<I, Proj>> Fun>
  constexpr ranges::for_each_result<I, Fun>
    ranges::for_each_n(I first, iter_difference_t<I> n, Fun f, Proj proj =
{});
```

Preconditions: `n >= 0` is `true`.

Effects: Calls `invoke(f, invoke(proj, *i))` for every iterator `i` in the range `[first, first + n)` in order. [*Note:* If the result of `invoke(proj, *i)` is a mutable reference, `f` may apply non-constant functions. — *end note*]

Returns: `{first + n, std::move(f)}`.

Remarks: If f returns a result, the result is ignored.

[*Note*: The overload in namespace ranges requires Fun to model copy_constructible. *—end note*]

Sample [alg.random.sample]

```
template<class PopulationIterator, class SampleIterator,
         class Distance, class UniformRandomBitGenerator>
  SampleIterator sample(PopulationIterator first, PopulationIterator last,
                        SampleIterator out, Distance n,
                        UniformRandomBitGenerator&& g);
```

```
template<input_iterator I, sentinel_for<I> S, weakly_incrementable O, class
Gen>
  requires (forward_iterator<I> || random_access_iterator<O>) &&
    indirectly_copyable<I, O> &&
    uniform_random_bit_generator<remove_reference_t<Gen>>
  O ranges::sample(I first, S last, O out, iter_difference_t<I> n, Gen&& g);
template<input_range R, weakly_incrementable O, class Gen>
  requires (forward_range<R> || random_access_iterator<O>) &&
    indirectly_copyable<iterator_t<R>, O> &&
    uniform_random_bit_generator<remove_reference_t<Gen>>
  O ranges::sample(R&& r, O out, range_difference_t<R> n, Gen&& g);
```

Mandates: Distance is an integer type. For the overload in namespace std, *first is writable ([iterator.requirements.general]) to out.

Preconditions:
out is not in the range [first, last).
For the overload in namespace std:
— PopulationIterator meets the *Cpp17InputIterator* requirements ([input.iterators]).
— SampleIterator meets the *Cpp17OutputIterator* requirements ([output.iterators]).
— SampleIterator meets the *Cpp17RandomAccessIterator* requirements ([random.access.iterators]) unless PopulationIterator satisfies the *Cpp17ForwardIterator* requirements ([forward.iterators]).
— remove_reference_t<UniformRandomBitGenerator> meets the requirements of a uniform random bit generator type ([rand.req.urng]).
~~— out is not in the range [first, last).~~

[…]

Remarks:

— For the overload in namespace `std`, ~~S~~stable if and only if `PopulationIterator` meets the *Cpp17ForwardIterator* requirements. For the first overload in namespace `ranges`, stable if and only if `I` models `forward_iterator`.
— To the extent that the implementation of this function makes use of random numbers, the object `g` ~~shall~~ serves as the implementation's source of randomness.

Shift [alg.shift]

```
template<class ForwardIterator>
  constexpr ForwardIterator
    shift_left(ForwardIterator first, ForwardIterator last,
               typename iterator_traits<ForwardIterator>::difference_type n);
template<class ExecutionPolicy, class ForwardIterator>
  ForwardIterator
    shift_left(ExecutionPolicy&& exec, ForwardIterator first,
               ForwardIterator last,
               typename iterator_traits<ForwardIterator>::difference_type n);
template<permutable I, sentinel_for<I> S>
  constexpr subrange<I>
    ranges::shift_left(I first, S last, iter_difference_t<I> n);
template<forward_range R>
  requires permutable<iterator_t<R>>
  constexpr safe_subrange_t<R>
    ranges::shift_left(R&& r, range_difference_t<R> n);
```

Preconditions: `n >= 0` is true. For the overloads in namespace `std`, t~~T~~he type of `*first` meets the *Cpp17MoveAssignable* requirements ([tab:cpp17.moveassignable]).
Effects: If `n` ~~<~~`== 0` or `n >= last - first`, does nothing. Otherwise, moves the element from position `first + n + i` into position `first + i` for each non-negative integer `i < (last - first) - n`. ~~In the first overload case~~For the overloads with no `ExecutionPolicy`, does so in order starting from `i = 0` and proceeding to `i = (last - first) - n - 1`.

Returns: Let *NEW_LAST* be `first + (last - first - n)` if ~~n is positive and~~ `n < last - first`, otherwise `first` ~~if n is positive, otherwise `last`.~~:

- *NEW_LAST* for the overloads in namespace `std`, or

- `{first, `*NEW_LAST*`}` for the overloads in namespace `ranges`.

Complexity: At most `(last - first) - n` assignments.

```
template<class ForwardIterator>
  constexpr ForwardIterator
    shift_right(ForwardIterator first, ForwardIterator last,
                typename iterator_traits<ForwardIterator>::difference_type
n);
template<class ExecutionPolicy, class ForwardIterator>
  ForwardIterator
    shift_right(ExecutionPolicy&& exec, ForwardIterator first,
                ForwardIterator last,
                typename iterator_traits<ForwardIterator>::difference_type
n);
```

```
template<permutable I, sentinel_for<I> S>
  constexpr subrange<I>
    ranges::shift_right(I first, S last, iter_difference_t<I> n);
template<forward_range R>
  requires permutable<iterator_t<R>>
  constexpr safe_subrange_t<R>
    ranges::shift_right(R&& r, range_difference_t<R> n);
```

Preconditions: `n >= 0` is true. For the overloads in namespace `std`, t~~T~~he type of `*first` meets the *Cpp17MoveAssignable* requirements~~.~~ ([tab:cpp17.moveassignable]) and `ForwardIterator` meets the *Cpp17BidirectionalIterator* requirements ([bidirectional.iterators]) or the *Cpp17ValueSwappable* requirements ([swappable.requirements]).

Effects: If `n` ~~`<`~~`== 0` or `n >= last - first`, does nothing. Otherwise, moves the element from position `first + i` into position `first + n + i` for each non-negative integer `i < (last - first) - n`. ~~In the first overload case, if~~ Does so in order starting from `i = (last - first) - n - 1` and proceeding to `i = 0` if:

- `ForwardIterator` meets the *Cpp17BidirectionalIterator* requirements ([bidirectional.iterators]), ~~does so in order starting from i = (last – first) – n – 1 and proceeding to i = 0.~~for the overload in namespace `std` with no `ExecutionPolicy`, or
- `decltype(first)` models `bidirectional_iterator`, for the overloads in namespace `ranges`.

Returns: Let *NEW_FIRST* be `first + n` if ~~n is positive and~~ `n < last - first`, otherwise `last`~~ if n is positive, otherwise `first`.~~:

- *NEW_FIRST* for the overloads in namespace `std`, or
- `{`*NEW_FIRST*`, last}` for the overloads in namespace `ranges`.

Complexity: At most `(last - first) - n` assignments or swaps.

Bounded value [alg.clamp]

```
template<class T>
  constexpr const T& clamp(const T& v, const T& lo, const T& hi);
template<class T, class Compare>
  constexpr const T& clamp(const T& v, const T& lo, const T& hi, Compare
comp);
template<class T, class Proj = identity,
indirect_strict_weak_order<projected<const T*, Proj>> Comp = ranges::less>
  constexpr const T& ranges::clamp(const T& v, const T& lo, const T& hi, Comp
comp = {}, Proj proj = {});
```

[…]

Complexity: At most two comparisons and three applications of any projection.

IV. Revision History

- R3, 9.1.20 - Wording changes following Cologne and Belfast reviews as well as a review by the forming Israeli committee. Rebased on N4842.
- R2, 9.3.19 - Wording fixes and improvements following LWG review. Integrated P1233 wording changes.
- R1, 8.11.18 - Remove overload of `for_each_n` taking a range parameter following LEWG guidance.
- R0, 7.10.18 - Initial revision

V. Acknowledgements

- Special thanks to Casey Carter for his guidance.
- My gratitude to the forming Israeli committee for their review and comments.