Document number: P1243R0
Date: 2018, Oct. 7
Author: Dan Raviv <dan.raviv@gmail.com>
Audience: LEWG, 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 additions are straightforward and require no new concepts or classes. The motivation is to increase the chance of the proposal going through and the new rangified overloads to make it into C++20 in the San Diego meeting: the required LEWG review is expected to be minimal, and hopefully LWG would have enough time to go over the wording of the new overloads, similar to the wording review of P0896.

This paper does *not* provide rangified overloads for the rest of the additions to `<algorithm>` since C++14: `compare_3way`, `lexicographical_compare_3way`, `search(`*range, searcher*`)`, as those probably do require adding new concepts and classes and would require non-trivial LEWG design review. The rangified overloads for these will be provided in a separate proposal.

II. Impact On the Standard

This is a pure library extension of the Standard (after P0896 is merged).

III. Proposed Wording

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

```
// 24.6.4, 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<InputIterator I, Sentinel<I> S, class Proj = identity,
IndirectUnaryInvocable<projected<I, Proj>> Fun>
    constexpr for_each_result<I, Fun>
      for_each_n(I first, iter_difference_t<I> n, Fun f, Proj proj = Proj{});
  template<InputRange Rng, class Proj = identity,
IndirectlyUnaryInvocable<projected<iterator_t<Rng>, Proj>> Fun>
    constexpr for_each_result<safe_iterator_t<Rng>, Fun>
      for_each_n(I first, iter_difference_t<iterator_t<Rng>> n, Fun f, Proj
proj = Proj{});
}
```

// 24.6.12, 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<class I, class O>
  using sample_result = copy_result<I, O>;

  template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O, class Gen>
    requires (ForwardIterator<I> || RandomAccessIterator<O>) &&
      IndirectlyCopyable<I, O> &&
      UniformRandomBitGenerator<remove_reference_t<Gen>>
    sample_result<I, O>
      sample(I first, S last, O out, iter_difference_t<I> n, Gen&& g);
  template<InputRange Rng, WeaklyIncrementable O, class Gen>
    requires (ForwardRange<Rng> || RandomAccessIterator<O>) &&
      IndirectlyCopyable<iterator_t<Rng>, O> &&
      UniformRandomBitGenerator<remove_reference_t<Gen>>
    sample_result<I, O>
      sample(Rng&& rng, O out, iter_difference_t<iterator_t<Rng>> n, Gen&&
g);
```

// 24.6.14, 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<I> S>
    constexpr subrange<I>
      shift_left(I first, S last, iter_difference_t<I> n);
  template<ForwardRange Rng>
    requires Permutable<iterator_t<Rng>>
    constexpr safe_subrange_t<Rng>
      shift_left(Rng&& rng, iter_difference_t<iterator_t<Rng>> 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<I> S>
    constexpr subrange<I>
      shift_right(I first, S last, iter_difference_t<I> n);
  template<ForwardRange Rng>
    requires Permutable<iterator_t<Rng>>
    constexpr safe_subrange_t<Rng>
      shift_right(Rng&& rng, iter_difference_t<iterator_t<Rng>> n);
}
```

// 24.7.9, 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,
    IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less<>>
    constexpr const T& clamp(const T& v, const T& lo, const T& hi, Comp comp
= Comp{}, Proj proj = Proj{});
}
```

24.6.4 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.

```
namespace ranges {
  template<InputIterator I, Sentinel<I> S, class Proj = identity,
IndirectUnaryInvocable<projected<I, Proj>> Fun>
    constexpr for_each_result<I, Fun>
      for_each_n(I first, iter_difference_t<I> n, Fun f, Proj proj = Proj{});
  template<InputRange Rng, class Proj = identity,
IndirectlyUnaryInvocable<projected<iterator_t<Rng>, Proj>> Fun>
```

```
    constexpr for_each_result<safe_iterator_t<Rng>, Fun>
      for_each_n(I first, iter_difference_t<iterator_t<Rng>> n, Fun f, Proj
proj = Proj{});
}
```

Requires: `n >= 0`.

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.

24.6.12 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);
namespace ranges {
 template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O, class Gen>
    requires (ForwardIterator<I> || RandomAccessIterator<O>) &&
      IndirectlyCopyable<I, O> &&
      UniformRandomBitGenerator<remove_reference_t<Gen>>
    sample_result<I, O>
      sample(I first, S last, O out, iter_difference_t<I> n, Gen&& g);
  template<InputRange Rng, WeaklyIncrementable O, class Gen>
    requires (ForwardRange<Rng> || RandomAccessIterator<O>) &&
      IndirectlyCopyable<iterator_t<Rng>, O> &&
      UniformRandomBitGenerator<remove_reference_t<Gen>>
    sample_result<iterator_t<Rng>, O>
      sample(Rng&& rng, O out, iter_difference_t<iterator_t<Rng>> n, Gen&&
g);
}
```

Requires:

— `out` shall not be in the range `[first, last)`.

For the overloads in namespace `std`:

— `PopulationIterator` shall ~~satisfy~~meet the *Cpp17InputIterator* requirements (22.2.3).

— `SampleIterator` shall ~~satisfy~~meet the *Cpp17OutputIterator* requirements (22.2.4).

— `SampleIterator` shall ~~satisfy~~meet the *Cpp17RandomAccessIterator* requirements (22.2.7) unless `PopulationIterator` satisfies the

Cpp17ForwardIterator requirements (22.2.5).
— `PopulationIterator`'s value type shall be writable (22.2.1) to `out`.
— `Distance` shall be an integer type.
— `remove_reference_t<UniformRandomBitGenerator>` shall ~~satisfy the requirements of a~~meet the uniform random bit generator ~~type~~requirements (24.7.2.3) ~~whose return type is convertible to Distance~~.
~~— out shall not be in the range [first, last).~~

Effects: Copies `min(last - first, n)` elements (the *sample*) from `[first, last)` (the *population*) to `out` such that each possible sample has equal probability of appearance. [*Note*: Algorithms that obtain such effects include *selection sampling* and *reservoir sampling*. —*end note*]

Returns: Let SAMPLE_END be t~~T~~he end of the resulting sample range. Returns
- SAMPLE_END for the overloads in namespace `std`, or
- {`last`, SAMPLE_END} for the overloads in namespace `ranges`.

Complexity: O(`last - first`).

Remarks:
— For the overloads in namespace `std`, ~~S~~stable if and only if `PopulationIterator` satisfies the *Cpp17ForwardIterator* requirements. For the overloads in namespace `ranges`, stable if and only if `I` models `ForwardIterator`.
— To the extent that the implementation of this function makes use of random numbers, the object referenced by `g` shall serve as the implementation's source of randomness.

24.6.14 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);
namespace ranges {
  template<Permutable I, Sentinel<I> S>
    constexpr subrange<I>
      shift_left(I first, S last, iter_difference_t<I> n);
  template<ForwardRange Rng>
```

```
    requires Permutable<iterator_t<Rng>>
    constexpr safe_subrange_t<Rng>
      shift_left(Rng&& rng, iter_difference_t<iterator_t<Rng>> n);
}
```

Requires: For the overloads in namespace `std`, t~~T~~he type of `*first` shall meetsatisfy the *Cpp17MoveAssignable* requirements ([tab: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);
namespace ranges {
  template<Permutable I, Sentinel<I> S>
   constexpr subrange<I>
      shift_right(I first, S last, iter_difference_t<I> n);
  template<ForwardRange Rng>
    requires Permutable<iterator_t<Rng>>
    constexpr safe_subrange_t<Rng>
      shift_right(Rng&& rng, iter_difference_t<iterator_t<Rng>> n);
}
```

Requires: For the overloads in namespace `std`, t~~T~~he type of `*first` shall meetsatisfy the *Cpp17MoveAssignable* requirements~~.~~ ([tab:moveassignable]), and `ForwardIterator` shall meet the *Cpp17BidirectionalIterator* ~~requirements~~

([bidirectional.iterators]) or the *Cpp17ValueSwappable* ([swappable.requirements]) 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~~satisfies~~ 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 `BidirectionalIterator`, 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.

24.7.9 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);
namespace ranges {
  template<class T, class Proj = identity,
    IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less<>>
    constexpr const T& clamp(const T& v, const T& lo, const T& hi, Comp comp
= Comp{}, Proj proj = Proj{});
}
```

Requires: The value of `lo` shall be no greater than `hi`. For the first form, type `T` shall be *Cpp17LessThanComparable* ([tab:lessthancomparable]).

Returns: `lo` if `v` is less than `lo`, `hi` if `hi` is less than `v`, otherwise `v`.

[*Note*: If `NaN` is avoided, `T` can be a floating-point type. —*end note*]

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

IV. Revision History

- R0, 7.10.18 - Initial revision

V. Acknowledgements

- Special thanks to Casey Carter for his guidance.