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]

```cpp
// [alg.foreach], for each

// [alg.foreach], for each

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 = {});

  // [algorithms.parallel.overloads]
  ForwardIterator for_each_n(ExecutionPolicy&& exec, // see
    ForwardIterator first, Size n, Function f);
}
```
// [alg.random.sample], sample

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

```cpp
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);
}
```

[...]

// [alg.shift], shift

```cpp
template<class ForwardIterator>
    constexpr ForwardIterator 
    shift_left(ForwardIterator first, ForwardIterator last, 
                typename iterator_traits<ForwardIterator>::difference_type n);
```

```cpp
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);
}
```

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

```cpp
namespace ranges {
    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_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
namespace ranges {
    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& 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>
ranges::for_each::for_each_result<I, Fun>
    ranges::for_each_n(I first, iter_difference_t<I> n, Fun f, Proj proj = {});

Preconditions: \( n \geq 0 \) is true.

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

Returns: \([\text{first} + n, \text{std::move}(f)\}).
Remarks: If \( \mathbf{f} \) returns a result, the result is ignored.

[Note: The overload in namespace \texttt{ranges} requires \texttt{Fun} to model \texttt{copy_constructible}. —end note]

Sample [\texttt{alg.random.sample}]

\[
\begin{aligned}
\text{template}<&\text{class PopulationIterator, class SampleIterator,} \\
&\text{class Distance, class UniformRandomBitGenerator}> \\
\text{SampleIterator sample(PopulationIterator first, PopulationIterator last,} \\
\text{SampleIterator out, Distance n,} \\
\text{UniformRandomBitGenerator&& g});
\end{aligned}
\]

\[
\begin{aligned}
\text{template}<&\text{input_iterator I, sentinel_for<I> S, weakly_incrementable O, class} \\
&\text{Gen}> \\
\text{requires (forward_iterator<I> || random_access_iterator<O>) &&} \\
\text{indirectly_copyable<I, O> &&} \\
\text{uniform_random_bit_generator<remove_reference_t<Gen>>} \\
\text{O ranges::sample(I first, S last, O out, iter_difference_t<I> n, Gen&& g)};
\end{aligned}
\]

\[
\begin{aligned}
\text{template}<&\text{input_range R, weakly_incrementable O, class} \\
&\text{Gen}> \\
\text{requires (forward_range<R> || random_access_iterator<O>) &&} \\
\text{indirectly_copyable<iterator_t<R>, O> &&} \\
\text{uniform_random_bit_generator<remove_reference_t<Gen>>} \\
\text{O ranges::sample(R&& r, O out, range_difference_t<R> n, Gen&& g)};
\end{aligned}
\]

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

\textit{Preconditions:}
\*out is not in the range \([\text{first, last}]\).

For the overload in namespace \texttt{std}:

\begin{itemize}
  \item PopulationIterator meets the \texttt{Cpp17InputIterator} requirements ([\texttt{input.iterators}]).
  \item SampleIterator meets the \texttt{Cpp17OutputIterator} requirements ([\texttt{output.iterators}]).
  \item SampleIterator meets the \texttt{Cpp17RandomAccessIterator} requirements ([\texttt{random.access.iterators}]) unless PopulationIterator satisfies the \texttt{Cpp17ForwardIterator} requirements ([\texttt{forward.iterators}]).
  \item remove_reference_t<UniformRandomBitGenerator> meets the requirements of a uniform random bit generator type ([\texttt{rand.req.urng}]).
  \item out is not in the range \([\text{first, last}]\).
\end{itemize}

[...]

\textit{Remarks:}
For the overload in namespace `std`, `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 serve as the implementation’s source of randomness.

**Shift** [alg.shift]

```cpp
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);
```

```cpp
template<permutable I, sentinel_for<I> S>
constexpr subrange<I>
    ranges::shift_left(I first, S last, iter_difference_t<I> n);
```

```cpp
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`, the 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.

```cpp
template<class ForwardIterator>
constexpr ForwardIterator
    shift_right(ForwardIterator first, ForwardIterator last,
                ...);```
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: $n \geq 0$ is true. For the overloads in namespace std, the type of *first meets the Cpp17MoveAssignble requirements 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:
- 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 \((\text{last} - \text{first}) - n\) assignments or swaps.

**Bounded value [alg.clamp]**

```cpp
template<class T>
    constexpr const T& clamp(const T& v, const T& lo, const T& hi);
```

```cpp
template<class T, class Compare>
    constexpr const T& clamp(const T& v, const T& lo, const T& hi, Compare comp);
```

```cpp
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.