Document number: P0769R1
Date: 2018, Feb. 5
Author: Dan Raviv <dan.raviv@gmail.com>
Audience: LWG

Add shift to `<algorithm>`

I. Introduction

This paper proposes adding shift algorithms to the C++ STL which move elements forward or backward in a range of elements.

II. Motivation and Scope

Shifting elements forward or backward in a range is a basic operation which the STL should allow performing easily. An important use case is time series analysis algorithms used in scientific and financial applications.

The scope of the proposal is adding the following function templates to `<algorithm>`:

```
template<class ForwardIterator>
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<class ForwardIterator>
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
);
```

A sample implementation which uses `std::move` to implement `shift_left` for forward iterators and `std::move_backward` to implement `shift_right` for bidirectional iterators can be found in https://github.com/danra/shift_proposal, though it's possible more efficient implementations could be made, since elements are guaranteed to be moved within the same range, not between two different ranges. The sample implementation also implements a non-trivial algorithm for `shift_right` of forward, non-bidirectional iterators.

III. Possible Objections and Responses

1) **Objection:** Shifting can be done by using `std::move` (in `<algorithm>`).

Response: Which of `std::move` or `std::move_backward` must be used depends on the shift direction, which is error-prone. It also makes for less readable code; consider

```
std::shift_right(v.begin(), v.end(), 3);
vs.
std::move_backward(v.begin(), v.end() - 3, v.end());
```

In addition, `shift_right` and `shift_left` may be implemented more efficiently than `std::move` and `std::move_backward`, since elements are guaranteed to be moved within the same range, not between two different ranges.

Also, ranges of forward, non-bidirectional iterators cannot be shifted right using either `std::move` or `std::move_backward`. Such ranges are possible to shift right, though, in O(N) time and constant space, as shown in the sample implementation.

2) **Objection:** Instead of shifting a range, you can use a circular buffer.

Response: A circular buffer is a valid alternative. However, it should not be forced on the programmer, and it does have its own limitations:

- In case there are multiple indices into the buffer, all must be updated in some way.
- Similarly, in the common case where there is some mask applied to the buffer which should not cycle with the data, the mask indices need to be updated whenever the buffer is cycled.
- A programmer might need to shift elements in non-circular buffers provided by a 3rd-party library.

3) **Objection:** There's already `std::rotate` which is similar in functionality.

Response: Shifting just the desired elements would allow for both a more efficient implementation and clearer semantics in case rotation is not needed.

IV. Impact On the Standard

The only impact on the standard is adding the proposed function templates to `<algorithm>`.

V. Design Decisions

1) `shift_left` and `shift_right` are provided as separate function templates instead of just a single `shift` function template to maximize performance and minimize compiled code size. Since shifting left and shifting right may have significantly different implementations (as is the case in the sample implementation), implementing both shift directions in a single `shift`

function template would both require extra conditional logic and inline less easily than the specific direction shifts.

Given that both `shift_left` and `shift_right` are provided, it would still be possible to provide a convenience `shift` function as well, but it seems redundant.

2) `shift_left` should return an iterator to the new end of the shifted range. The beginning of the shifted range would always be equal to the beginning of the range before the shift, so there is no need to also return an iterator to the beginning of the shifted range. (This is similar to how `std::move` only returns an iterator to the end of the moved range).

Similarly, `shift_right` should return an iterator to the new beginning of the shifted range. The end of the shifted range would always be equal to the end of the range before the shift, so there is no need to also return an iterator to the end of the shifted range. (This is similar to how `std::move_backward` only returns an iterator to the end of the moved range).

3) After shifting a range by n elements, either to the right or to the left, exactly n elements would be left "empty", with their previous values having been moved to other elements but with no new values moved into them. It's been suggested to provide function template overloads which with an extra `filler` value parameter which would set all such "empty" elements to its value. However, this has been decided to be redundant, since the iterator returned from `shift_left/right` (see (2) above) can be passed to `std::fill` (along with unchanged `begin`/`end` of the range) to fill the empty values. There is no comparative advantage to implementing this extra functionality in the STL over implementing a simple helper function which calls the desired shift algorithm and then performs a fill (or any other desired action) of the "empty" elements,

4) `shift_left` without an execution policy or with the standard `std::execution::seq` execution policy moves the shifted elements (those which would still present in the range after the shift) in order, similar to how `std::move` moves elements in order.

Similarly, `shift_right` of bidirectional iterators without an execution policy or with the standard `std::execution::seq` execution policy moves the shifted elements (those which would still present in the range after the shift) in reverse order, similar to how `std::move_backward` moves elements in reverse order.

There is no guarantee what order `shift_right` of forward iterators moves elements in.

5) Shifting a range by more than its length (`std::distance(first, last)`), either to the left or to the right, has the effect of shifting out all of the elements, the same as shifting a range by exactly the length of the range. This could be defined as undefined behavior instead, but there would probably be no extra cost to handle larger shifts, seeing as an implementation would have to handle a shift by exactly the length of the range anyway.

6) Shifting a range by zero does nothing. This might have some minimal extra run-time cost: for example, in the sample implementation, since both `std::move` and `std::move_backward`

have undefined behavior when moving a range exactly onto itself, an extra `n==0` condition check must be done before performing either of them for a shift by zero to have defined behavior.

However, the alternative of making a shift by zero undefined behavior is too unexpected and user-unfriendly, with a high potential for causing bugs.

7) Shifting a range by a negative `n` is does nothing. While not very important (could be undefined behavior instead), checking for `n<=0` in implementations would probably have no extra cost over checking for `n==0` which is required anyway (see (6) above).

VI. Open Issues

1) It would be preferable for `shift_left` and `shift_right` to have more generic names; the fact that the first element in a range is the left-most is a matter of convention which is not specified in the standard, and some programmers may think of the first element as the right most, or maybe the top-most, etc.

However, `shift_backward`, `shift_back` and `shift_forward` are probably all out of the question, since other algorithms exist, e.g., `std::move_backward` and `std::copy_backward`, in which 'backward' means performing the operation starting from the back of the range, instead of from its front.

`shift_to_front` and `shift_to_back` come to mind; more ideas would be welcome.

VII. Proposed Wording

- In [algorithm.syn], after the declaration of `shuffle`, add:

```
 // [<link to alg.shift>], shift
template<class ForwardIterator>
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 [<link to algorithms.parallel.overloads>]
      ForwardIterator first, ForwardIterator last,
      typename iterator_traits<ForwardIterator>::difference_type n
);
template<class ForwardIterator>
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 [<link to algorithms.parallel.overloads>]
      ForwardIterator first, ForwardIterator last,
```

```
      typename iterator_traits<ForwardIterator>::difference_type n
);
```

- In [alg.modifying.operations], after [alg.shuffle], add a new [alg.shift] section:

28.6.?? Shift [alg.shift]

```
template<class ForwardIterator>
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
);
```

1 Requires: The type of `*first` shall satisfy the `MoveConstructible` and `MoveAssignable` requirements.
2 Effects: If `n <= 0` or `n >= last – first`, does nothing. Otherwise, moves elements in the range `[first + n, last)` into the range `[first, first + (last – first) – n)`, performing `*(first + i) = std::move(*(first + n + i))` for each non-negative integer `i < (last-first) – n`. Does so starting from `first` and proceeding to `first + (last – first) – n` in the first overload case.

3 Returns: `first + (last – first) – n` if `n` is positive and `n < last – first`, otherwise `first` if `n` is positive, otherwise `last`.
4 *Complexity:* Exactly `(last – first) – n` assignments if `n` is positive and `n < last – first`, otherwise 0 assignments.

```
template<class ForwardIterator>
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
);
```

1 Requires: The type of `*first` shall satisfy the `MoveConstructible` and `MoveAssignable` requirements. `ForwardIterator` shall meet the `BidirectionalIterator` or `ValueSwappable` requirements.

2 Effects: If `n <= 0` or `n >= last – first`, does nothing. Otherwise, moves elements in the range `[first, first + (last – first) - n)` into the range `[first + n, last)`, performing `*(first + n + i) = std::move(*(first + i))` for each non-negative integer `i < (last-`

`first) – n`. If `ForwardIterator` satisfies the `BidirectionalIterator` requirements, does so starting from `last – 1` and proceeding to `first + n` in the first overload case.

3 Returns: `first + n` if `n` is positive and `n < last – first`, otherwise `last` if `n` is positive, otherwise `first`.

4 *Complexity:* If `n <= 0` or `n >= last – first`, exactly 0 assignments/swaps. Otherwise:

- If `ForwardIterator` meets the `BidirectionalIterator` requirements, exactly `(last – first) – n` assignments.

- Otherwise, at most `(last – first) – n` swaps.

VIII. Acknowledgements

- Special thanks to Walter E. Brown for his comments and guidance writing the paper.
- Special thanks to Casey Carter for his comments, for advancing the paper in the standards committee, and for his generous code contribution, including a `shift_right` implementation for forward, non-bidirectional iterators.
- Thanks to Alexander Zaitsev, Arthur O'Dwyer, Bryce Glover, Howard Hinnant, Nicol Bolas and Ray Hamel for their helpful comments on the paper.