Document number: D????R0
Date: 2017-08-19
Author: Dan Raviv <dan.raviv@gmail.com>
Audience: LEWG => LWG

Add shift to `<algorithm>`

I. Introduction

This paper proposes adding shift algorithms to the C++ STL which shift 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. A main 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 ForwardIt>
ForwardIt shift_left(
        ForwardIt first, ForwardIt last,
        typename std::iterator_traits<ForwardIt>::difference_type n,
        std::optional<typename std::iterator_traits<ForwardIt>::value_type> filler = std::nullopt
);

template<class ExecutionPolicy, class ForwardIt>
ForwardIt shift_left(
        ExecutionPolicy&& policy, ForwardIt first, ForwardIt last,
        typename std::iterator_traits<ForwardIt>::difference_type n,
        std::optional<typename std::iterator_traits<ForwardIt>::value_type> filler = std::nullopt
);

template<class BidirIt>
BidirIt shift_right(
        BidirIt first, BidirIt last,
        typename std::iterator_traits<BidirIt>::difference_type n,
        std::optional<typename std::iterator_traits<BidirIt>::value_type> filler = std::nullopt
);

template<class ExecutionPolicy, class BidirIt>
BidirIt shift_right(
        ExecutionPolicy&& policy, BidirIt first, BidirIt last,
        typename std::iterator_traits<BidirIt>::difference_type n,
        std::optional<typename std::iterator_traits<BidirIt>::value_type> filler = std::nullopt
);
```

A sample implementation which uses the existing `std::move` and `std::move_backward` can be found here, 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.

III. Expected 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, `std::shift_right` and `std::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.

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) Ranges of bidirectional iterators can easily be shifted either left or right.

Ranges of forward iterators can easily be shifted left. Shifting them right is possible, but inefficient, requiring either O(N) space or O(N^2) time, N denoting the size of the range.

Therefore, we propose that `std::shift_left` would support forward iterators, while `std::shift_right` would support only bidirectional iterators. This is similar to how `std::move_backward` only supports bidirectional iterators, even though it could have supported forward iterators inefficiently.

2) Following (1) above, `std::shift_left` is required for supporting forward iterators. We were now left with the following alternatives:
a) Add `std::shift_right` to support right shifts of ranges of bidirectional iterators
b) Add `std::shift` to support either left *or* right shifts of ranges of bidirectional iterators – with the `n` parameter having positive values for right shifts and negative values for left shifts.
c) Add both `std::shift_right` and `std::shift`.

We chose alternative a, with the main reason being that `std::shift_right` may be implemented more simply and perform better than `std::shift`, both because it doesn't have to check the shift direction before proceeding, and because it would have a smaller implementation, possibly inlining more easily.

Given `std::shift_right` and `std::shift_left`, adding `std::shift` doesn't seem important enough to justify declaring and implementing the necessary functions in the standard library. In case a programmer is interested in having a `shift` function to which he can hand either a positive or negative argument, it is trivial to implement given `std::shift_right` and `std::shift_left.`

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 shifted to other elements but with no new values shifted into them. We suggest providing an optional `filler` value parameter which, if given, all such

"empty" elements would be set to. In case `filler` is not given, the only guarantee is that the "empty" elements have some valid values (not necessarily the same values as before the shift).

4) `std::shift_left` without an execution policy or with the standard `sequenced_policy` execution policy shifts the elements in order, similar to how `std::move` moves elements in order.

Similarly, `std::shift_right` without an execution policy or with the standard `sequenced_policy` execution policy shifts the elements in reverse order, similar to how `std::move_backward` moves elements in reverse order.

5) `std::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, `std::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.

6) Shifting a range by more than its length (`std::distance(first, last)`) either to the left or to the right, is undefined behavior. This may simplify implementation and optimize performance.

The only benefit to making this defined behavior would be for algorithms where the shift count isn't known in advance, and when it isn't, the given filler value should fill the entire range. This doesn't seem an important enough case to justify preventing the aforementioned benefits of making the behavior undefined.

VI. Open Issues

1) Should shift by zero be undefined behavior?

Pro for undefined behavior: Could simplify implementation and optimize performance. 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.

Pro for defined behavior: It is reasonable to expect a shift by zero to do nothing, so it is programmer-error-prone to make it undefined behavior.

2) It would be preferable for `std::shift_left` and `std::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, `std::shift_backward`, `std::shift_back` and `std::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.

`std::shift_to_front` and `std::shift_to_back` come to mind. Perhaps there are better names; ideas would be welcome.

VII. Proposed Wording

TODO

VIII. Acknowledgements

Special thanks to Walter E. Brown for his comments and guidance writing the paper.
Thanks to Alexander Zaitsev, Arthur O'Dwyer, Bryce Glover, Howard Hinnant, Nicol Bolas and Ray Hamel for their helpful comments on the paper.