Document Number: P1899R0
Date: 2019-10-07
Audience: LEWG, SG9,
Reply to: Christopher Di Bella
cjdb.ns@gmail.com

stride_view

Contents

1 Motivation [motivation]

The ability to use algorithms over an evenly-spaced subset of a range has been missed in the STL for a quarter of a century. This is, in part, due to the complexity required to use an iterator that can safely describe such a range. It also means that the following examples cannot be transformed from raw loops into algorithms, due to a lacking iterator.

```
for (auto i = 0; i < ssize(v); i += 2) {
  v[i] = 42; // fill
}

for (auto i = 0; i < ssize(v); i += 3) {
  v[i] = f(); // transform
}

for (auto i = 0; i < ssize(v); i += 3) {
  for (auto j = i; j < ssize(v); i += 3) {
    if (v[j] < v[i]) {
      ranges::swap(v[i], v[j]); // naive bubble sort, but hopefully the idea is conveyed
    }
  }
}
```

Boost introduced a range adaptor called `strided`, and range-v3's equivalent is `stride_view`, both of which make striding far easier than when using iterators:

```
ranges::fill(v | views::stride(2), 42);

auto strided_v = v | views::stride(3);
ranges::transform(strided_v, ranges::begin(strided_v) f);
ranges::stable_sort(strided_v); // order restored!
```

Given that there's no way to compose a strided range adaptor in C++20, this should be one of the earliest range adaptors put into C++23.

1.0.1 Risk of not having `stride_view`

Although it isn't possible to compose `stride_view` in C++20, someone inexperienced with the ranges design space might mistake `filter_view` as a suitable way to "compose" `stride_view`:

```
auto bad_stride = [](auto const step) {
  return views::filter([n = 0, step](auto&&) mutable { return n++ % step == 0; });
};
```

This implementation is broken for two reasons:

1. `filter_view` expects a `predicate` as its input, but the lambda we have provided does not model `predicate` (a call to `invoke` on a `predicate` mustn't modify the function object, yet we clearly are).
2. The lambda provided doesn't account for moving backward, so despite *satisfying* `bidirectional_iterator`, it does not model the concept.

For these reasons, the author regrets not proposing this in the C++20 design space.

1.1 Implementation experience [impl.experience]

The idea of a striding range adaptor has been implemented in both Boost and range-v3 for widespread use. The proposed wording has (mostly) been implemented in cmcstl2 and in a CppCon main session.

1.2 Acknowledgements [acknowledgements]

The author would like to thank Tristan Brindle for providing editorial commentary on P1899, and also those who reviewed material for, or attended the aforementioned CppCon session or post-conference class, for their input on the design of the proposed `stride_view`.

2 Proposed wording [proposed.wording]

2.1 Range Adaptors [range.adaptors]

2.1.1 Stride view [range.stride]

2.1.1.1 Overview [range.stride.overview]

1 `stride_view` presents a view of an underlying sequence, advancing over `n` elements at a time, as opposed to the usual single-step succession.

2 [*Example*:

```
auto input = stride_view{views::iota(0, 12), 3};
ranges::copy(input, ostream_iterator<int>{cout, " "}); // prints: 0 3 6 9
ranges::copy(input | views::reverse, ostream_iterator<int>{cout, " "}); // prints: 9 6 3 0
```

— *end example*]

2.1.1.2 Class `stride_view` [range.stride.view]

```
namespace std::ranges {
  template<input_range R>
    requires view<R>
  class stride_view : public view_interface<stride_view<R>> {
    template<bool Const> class iterator; // exposition only
    template<bool Const> class sentinel; // exposition only
  public:
    stride_view() = default;
    constexpr explicit stride_view(R base, range_difference_t<R> stride);

    constexpr R base() const;
    constexpr range_difference_t<R> stride() const noexcept;

    constexpr iterator<false> begin() requires (!simple-view<R>);
    constexpr iterator<true> begin() const requires range<const R>;

    constexpr auto end() requires (!simple-view<R>)
    {
      if constexpr (sized_range<R> && common_range<R>) {
        return iterator<is_const_v<Self>>{
          self,
          ranges::end(self.base_),
          static_cast<range_difference_t<R>>(ranges::size(self.base_)) % self.stride_
        };
      }
      else {
        return sentinel<is_const_v<Self>>{ranges::end(self.base_)};
      }
    }

    constexpr auto end() const requires range<const R>
    {
      if constexpr (sized_range<R> && common_range<R>) {
        return iterator<is_const_v<Self>>{
          self,
          ranges::end(self.base_),
          static_cast<range_difference_t<R>>(ranges::size(self.base_)) % self.stride_
        };
      }
      else {
        return sentinel<is_const_v<Self>>{ranges::end(self.base_)};
      }
    }
```

```
    constexpr auto size() requires (sized_range<R> && !simple-view<R>)
    { return compute_distance(*this); }

    constexpr auto size() const requires sized_range<const R>
    { return compute_distance(*this); }
  private:
    R base_ = R{};                    // exposition only
    range_difference_t<R> stride_{}; // exposition only

    template<class I>
    constexpr I compute_distance(I distance) const // exposition only
    {
      const auto quotient = distance / static_cast<I>(stride_);
      const auto remainder = distance % static_cast<I>(stride_);
      return quotient + static_cast<I>(remainder > 0);
    }
  };

  template<input_range R>
    requires viewable_range<R>
  stride_view(R&&, range_difference_t<R>) -> stride_view<all_view<R>>;
}

constexpr explicit stride_view(R base, range_difference_t<R> stride);
```

1 *Effects:* Initializes `base_` with `base` and `stride_` with `stride`.

```
constexpr R base() const;
```

2 *Effects:* Equivalent to: `return base_;`

```
constexpr range_difference_t<R> stride() const;
```

3 *Effects:* Equivalent to: `return stride_;`

```
constexpr iterator<false> begin() requires (!simple-view<R>);
```

4 *Effects:* Equivalent to: `return iterator<false>{*this};`

```
constexpr iterator<true> begin() const requires range<const R>;
```

5 *Effects:* Equivalent to: `return iterator<true>{*this};`

2.1.1.3 Class `stride_view::iterator` [range.stride.iterator]

```
namespace std::ranges {
  template<class R>
  template<bool Const>
  class stride_view<R>::iterator {
    using Parent = conditional_t<Const, const stride_view, stride_view>; // exposition only
    using Base = conditional_t<Const, const R, R>;                       // exposition only

    friend iterator<!Const>;

    Parent* parent_ = nullptr;        // exposition only
    iterator_t<Base> current_{};      // exposition only
    range_difference_t<Base> step_{}; // exposition only
  public:
    using difference_type = range_difference_t<Base>;
    using value_type = range_value_t<Base>;
    using iterator_category = typename iterator_traits<iterator_t<Base>>::iterator_category;

    iterator() = default;

    constexpr explicit iterator(Parent& parent);
    constexpr explicit iterator(Parent& parent, iterator_t<Base> end, difference_type step);
    constexpr explicit iterator(const iterator<!Const>& other)
      requires Const && convertible_to<iterator_t<R>, iterator_t<Base>>;
```

```
    constexpr iterator_t<Base> base() const;

    constexpr decltype(auto) operator*() const { return *current_; }

    constexpr iterator& operator++();
    constexpr auto operator++(int);

    constexpr iterator& operator--() requires bidirectional_range<Base>;
    constexpr iterator operator--(int) requires bidirectional_range<Base>;

    constexpr iterator& operator+=(difference_type n) requires random_access_range<Base>;
    constexpr iterator& operator-=(difference_type n) requires random_access_range<Base>;

    constexpr decltype(auto) operator[](difference_type n) const
      requires random_access_range<Base>
    { return *(*this + n); }

    friend constexpr iterator operator+(const iterator& x, difference_type n)
      requires random_access_range<Base>;

    friend constexpr iterator operator+(difference_type n, const iterator& x)
      requires random_access_range<Base>;

    friend constexpr iterator operator-(const iterator& x, difference_type n)
      requires random_access_range<Base>;

    friend constexpr difference_type operator-(const iterator& x, const iterator& y)
      requires random_access_range<Base>;

    friend constexpr bool operator==(const iterator& x, const iterator& y) const
      requires equality_comparable<iterator_t<Base>>;

    friend constexpr bool operator<(const iterator& x, const iterator& y)
      requires random_access_range<Base>;
    friend constexpr bool operator>(const iterator& x, const iterator& y)
      requires random_access_range<Base>;
    friend constexpr bool operator<=(const iterator& x, const iterator& y)
      requires random_access_range<Base>;
    friend constexpr bool operator>=(const iterator& x, const iterator& y)
      requires random_access_range<Base>;

    friend constexpr compare_three_way_result_t<iterator_t<Base>>
      operator<=>(const iterator& x, const iterator& y)
        requires random_access_range<Base> && three_way_comparable<iterator_t<Base>>;

    friend constexpr range_rvalue_reference_t<R> iter_move(const iterator& x);
    friend constexpr void iter_swap(const iterator& x, const iterator& y)
      requires indirectly_swappable<iterator>;
  private:
    constexpr iterator& advance(difference_type n) // exposition only
    {
      if constexpr (!ranges::bidirectional_range<Parent>) {
        ranges::advance(current_, n * parent_->stride_, ranges::end(parent_->base_));
        return *this;
      }
      else {
        if (n > 0) {
          auto remaining = ranges::advance(current_, n * parent_->stride_, ranges::end(parent_->base_));
          step_ = parent_->stride_ - remaining;
        }
        else if (n < 0) {
          auto stride = step_ == 0 ? n * parent_->stride_
                                   : (n + 1) * parent_->stride_ - step_;
          step_ = ranges::advance(current_, stride, ranges::begin(parent_->base_));
```

```
        }

        return *this;
      }
    }
  };
}
```

[Note to reviewers: `advance` is unexpectedly complex due to the fact that reverse-iteration requires knowing how many elements to skip when evaluating `ranges::end(strided_view{r, n} | views::reverse) - x`. The need for the complexity arises when the distance between the last element in the base range and the sentinel is less than that of the stride `n`.]

```
constexpr explicit iterator(Parent& parent);
```

1 *Effects:* Initializes `parent_` with `addressof(parent)` and `current_` with `ranges::begin(parent)`.

```
constexpr explicit iterator(Parent& parent, iterator_t<Base> end, difference_type step);
```

2 *Effects:* Initializes `parent_` with `addressof(parent)`, `current_` with `std::move(end)`, and `step_` with `step`.

```
constexpr explicit iterator(const iterator<!Const>& other)
  requires Const && convertible_to<iterator_t<R>, iterator_t<Base>>;
```

3 *Effects:* Initializes `parent_` with `other.parent_`, `current_` with `other.current_`, and `step_` with `other.step_`.

```
constexpr iterator_t<Base> base() const;
```

4 *Effects:* Equivalent to: `return current_;`

```
constexpr iterator& operator++();
```

5 *Effects:* Equivalent to: `return advance(1);`

```
constexpr auto operator++(int);
```

6 *Effects:* Equivalent to:

```
if constexpr (!forward_range<Base>) {
  ++*this;
  return;
}
else {
  auto temp = *this;
  ++*this;
  return temp;
}
```

```
constexpr iterator& operator--() requires bidirectional_range<Base>;
```

7 *Effects:* Equivalent to: `return advance(-1);`

```
constexpr iterator operator--(int) requires bidirectional_range<Base>;
```

8 *Effects:* Equivalent to:

```
auto temp = *this;
--*this;
return temp;
```

```
constexpr iterator& operator+=(difference_type n) requires random_access_range<Base>;
```

9 *Effects:* Equivalent to: `return advance(n);`

```
constexpr iterator& operator-=(difference_type n) requires random_access_range<Base>;
```

10 *Effects:* Equivalent to: `return advance(-n);`

```
friend constexpr iterator operator+(const iterator& x, difference_type n)
```

```
  requires random_access_range<Base>;
```

11 *Effects:* Equivalent to: `return x += n;`

```
friend constexpr iterator operator+(difference_type n, const iterator& x)
  requires random_access_range<Base>;
```

12 *Effects:* Equivalent to: `return x += n;`

```
friend constexpr iterator operator-(const iterator& x, difference_type n)
  requires random_access_range<Base>;
```

13 *Effects:* Equivalent to: `return x -= n;`

```
friend constexpr difference_type operator-(const iterator& x, const iterator& y)
  requires random_access_range<Base>;
```

14 *Effects:* Equivalent to: `return x.parent_->compute_distance(x.current_ - y.current_);`

```
constexpr bool operator==(const iterator& x, const iterator& y) const
  requires equality_comparable<iterator_t<Base>>;
```

15 *Effects:* Equivalent to: `return x.current_ == y.current;`

```
friend constexpr bool operator<(const iterator& x, const iterator& y)
  requires random_access_range<Base>;
```

16 *Effects:* Equivalent to: `return x.current_ < y.current;`

```
friend constexpr bool operator>(const iterator& x, const iterator& y)
  requires random_access_range<Base>;
```

17 *Effects:* Equivalent to: `return y < x;`

```
friend constexpr bool operator<=(const iterator& x, const iterator& y)
  requires random_access_range<Base>;
```

18 *Effects:* Equivalent to: `return !(y < x);`

```
friend constexpr bool operator>=(const iterator& x, const iterator& y)
  requires random_access_range<Base>;
```

19 *Effects:* Equivalent to: `return !(x < y);`

```
friend constexpr compare_three_way_result_t<iterator_t<Base>>
  operator<=>(const iterator& x, const iterator& y)
    requires random_access_range<Base> && three_way_comparable<iterator_t<Base>>;
```

20 *Effects:* Equivalent to: `return x.current_ <=> other.current_;`

```
friend constexpr range_rvalue_reference_t<R> iter_move(const iterator& x);
```

21 *Effects:* Equivalent to: `return ranges::iter_move(x);`

```
friend constexpr void iter_swap(const iterator& x, const iterator& y)
  requires indirectly_swappable<iterator>;
```

22 *Effects:* Equivalent to: `ranges::iter_swap(x.current_, y.current_);`

2.1.1.4 Class template `stride_view::sentinel` [range.stride.sentinel]

```
namespace std::ranges {
  template<class R>
  template<bool Const>
  class stride_view<R>::sentinel {
    using Parent = conditional_t<Const, const stride_view, stride_view>; // exposition only
    using Base = conditional_t<Const, const R, R>;                       // exposition only

    sentinel_t<Base> end_ = sentinel_t<Base>{}; // exposition only
  public:
    sentinel() = default;
    constexpr explicit sentinel(sentinel_t<Base> end);
```

```
      constexpr sentinel(sentinel<!Const> other)
        requires Const && convertible_to<sentinel_t<R>, sentinel_t<Base>>;

      constexpr sentinel_t<Base> base() const;

      constexpr bool operator==(const iterator<Const>& x, const sentinel& y) const;
    };
  }

constexpr explicit sentinel(sentinel_t<Base> end);
```

1 *Effects:* Initializes `end_` with `end`.

```
constexpr sentinel(sentinel<!Const> other)
  requires Const && convertible_to<sentinel_t<R>, sentinel_t<Base>>;
```

2 *Effects:* Initializes `end_` with `std::move(other.end_)`.

```
constexpr sentinel_t<Base> base() const;
```

3 *Effects:* Equivalent to: `return end_;`

```
friend constexpr bool operator==(const iterator<Const>& x, const sentinel& y) const;
```

4 *Effects:* Equivalent to: `return x.current_ == y.end_;`

2.1.2 `views::stride` [range.stride.view]

1 The name `views::stride` denotes a range adaptor object ([range.adaptor.object]). For subexpressions `E` and `N`, the expression `views::stride(E, N)` is expression-equivalent to `stride_view{E, N}`.