size_hint: Eagerly reserving memory for not-quite-sized lazy ranges

Document #: P2846R1
Date: 2023-09-15
Programming Language C++
Audience: SG-9, LEWG
Reply-to: Corentin Jabot <corentin.jabot@gmail.com>

Abstract
This proposals adds a ranges::size_hint customization point object which allows ranges::to to reserve memory for non-sized ranges whose size can be approximated.

Revisions

R1
When SG-9 reviewed R0, Tim Song pointed out that std::vector and std::string call std::distance in their constructor (rather than calling push_back and reallocating themselves). This was not mentioned or considered at all in R0. R1 fixes this blatant omission.

If anything this makes the motivation stronger but it opens some interesting design questions.

SG-9 also pointed out that R0, despite being motivated by performance considerations failed to present any benchmarks. Benchmarks are now provided.

SG-9 also suggested that it would be useful for sized_range to subsume approximately_sized_range. This change has been applied.

Add a section on naming.

Motivation
You should know this proposal is secretly about Unicode.

Consider the string
"In C++ ist es schwieriger, sich selbst in den fuß zu schießen."

Its correct upper-case transformation is
"IN C++ IST ES SCHWIERIGER, SICH SELBST IN DEN FUSS ZU SCHIESSEN".
You will have observed that the transformed string is longer. 2 codepoints longer to be exact. This is because B capitalizes as SS.

Now consider a hypothetical uppercase_view that transforms Unicode codepoints into their uppercase version. It might look like this:

```cpp
U"In C++ ist es schwieriger, sich selbst in den fuß zu schießen."sv
 | views::uppercase
 | ranges::to<std::u32string>();
```

We do not have to know anything else about views::uppercase to know that it cannot be a sized_range.

In fact, it cannot be, for the same reason, a random_access_range. It will be, at best, a bidirectional_range. This means that in the above example, we cannot reserve memory in advance when constructing the output vector in ranges::to. So the implementation will

- for a forward range, call distance, in effect going over the view twice (unless the range is random access, which is never the case in the presence of Unicode algorithms) which for a Unicode algorithm is very expensive.
- for a non forward range, call push_back in a loop

But... only a couple thousand codepoints have a casing that is not their identity and of those only a few hundred might produce more than one codepoint. That's about 1% of the total number of allocated codepoints.

Most scripts not derived from Latin (for example CJK characters) do not even have a notion of case.

So the above example is massively pessimized because of a statistically unlikely scenario.

This is not just a problem with upper-casing. All casing transformations in Unicode have the same peculiarity. It also applies to all forms of Unicode Normalization (normalization can produce output strings that are longer, or shorter than the input) and text encoding and decoding, especially between UTF forms.

It is not possible to compute the size of the conversion from a string of length L from UTF-32 to UTF-8, but we know it will be at least L. And for some scripts, including English, it will be equal or very close to L.

We need a way to advertise “This range is about yea big” so that ranges::to and all the ranges constructors added by P1206R7 [2] can reduce the number of allocations they perform.

**Design**

To that end, we propose:

- a ranges::size_hint CPO
- a approximately_sized_range concept that checks whether a range supports size_hint
the extension of many existing standard views so that they can forward the size_hint of their adapted view

ranges::size_hint

ranges::size_hint is a CPO that calls

- ranges::size for sized ranges
- the size_hint member function.
- the size_hint function found by adl.

Like ranges::size we mandate O(1) evaluation of size_hint. With this design, ranges that are sized are already approximately_sized_range, which avoids some duplication, undue complexity, and confusion. For example, we don’t have to modify existing containers and views whose sized-ness does not depend on another view.

Usage

An implementation of uppercase_view might look like this

template <input_range V>
class uppercase_view {
    constexpr const V & base() const;
    constexpr auto begin() const;
    constexpr auto end() const;

    constexpr auto size_hint() requires approximately_sized_range<View> {
        return ranges::size_hint(base());
    }

    constexpr auto size_hint() const requires approximately_sized_range<const View> {
        return ranges::size_hint(base());
    }
};

IE, we assume uppercase_view will have approximately the same size as the underlying range, even if it might be in rare cases slightly more. Note that uppercase_view would not provide a size method because it cannot determine its exact size.

Adapting existing views

Views that transform elements of their adapted view can just forward the size_hint of their underlying view. Nothing too complicated there. drop, take, adjacent chunk, slide and stride can also compute a size_hint the same way they compute their size.

join and split cannot compute their size in O(1), so they do not provide a size_hint.
Views with predicate

take_while, drop_while and filter could, in theory, expose the size of their adapted range. however, this might lead to huge overallocation so, conservatively, these things do not expose a size_hint.

zip and cartesian_product

It would be reasonable for zip's size_hint to be the smallest size_hint amongst the adapted ranges that do have a size_hint. Similarly, it would make sense that the size_hint of cartesian_product would be the product of the size_hint of the ranges that do have one.

I think the correct way to do that might be to provide 3 overloads:

```cpp
constexpr auto size_hint() requires (approximately_sized_range<Views> &&...);
constexpr auto size_hint() const requires (approximately_sized_range<const Views> &&...);
constexpr auto size_hint() const requires (!((approximately_sized_range<Views> ||
    approximately_sized_range<Views>) &&...));
```

Otherwise, the const and non-const overload could give different results.

As I'm not entirely certain what the best approach is, this paper does not make a change to zip or cartesian_product

ranges::to

ranges::to is sligtly modified to use ranges::size_hint instead of ranges::size. An implementation can also use size_hint in the various range constructors. User code could use that feature for similar purposes.

vector and string constructors

std::vector performs a single allocation on construction by calling distance first. It could only use size_hint on types that are Cpp17MoveInsertable (which is the case for Unicode algorithms), as reallocation would force items to be moved around. It may affect the performance of vector construction negatively when it is constructed from a forward range that is not sized, but for which calling distance is cheap, and size_hint is off enough that the reallocations are noticeable.

The proposed wording is currently silent on what happens to vector but we have 3 options:

- Require implementation to rely on size_hint when constructing a vector from a non-sized range of Cpp17MoveInsertable elements.

  A concern here is that custom allocators can't tell whether elements are move insertable because they are not required to be SFINAE-friendly. So, we could either only require calling size_hint for std::allocator only, or it would break vectors of non-movable types using a custom allocator.
Another solution is to check for MoveConstructible - which would still break the fringe case of types that are MoveConstructible and not Cpp17MoveInsertable. But there is consensus in SG-9 that those are fringe edge cases.

- Allowing implementations to do so without requiring it.

Note that whether allocations are performed would be observable through whether the move constructor of each element is called or not (unless the elements are of trivial types (or are otherwise trivially relocatable P2786R0 [1])). Any effect resulting from a call to std::distance would not be performed.

SG-9 had a preference for this option.

- In the presence of a approximately_sized_range, ranges::to could construct the container using reserve and insertions, instead of forwarding to the container constructor. The downside of this approach is that ranges::to would have different performance characteristics than container construction, which seems undesirable, and the optimization could not be applied to other operations such as assign_range/insert_range.

**Benchmarks**

The following graph shows 2 Unicode normalization algorithms (Upper casing and NFC) for translations of the Universal Declaration of Human Rights in English, Danish, Japanese, and Chinese. Each such Transformation is performed twice:

- By reserving the number of elements of the pre-transformation text in the output container, emulating the proposed size_hint

- By constructing the output container directly without reserve (status quo behavior of ranges::to).

The graph shows that, for a forward range, reserving is consistently about twice as fast. This is consistent with the fact the range is traversed twice.
If the input range is non-forward, there is less difference on small datasets:

Non forward ranges with and without reserve

But as the data gets bigger and the number of allocations increases being able to reserve memory gets more noticeably impactful (performance delta of 5-10%).
Existing practices and alternatives considered

Rust has a similar facility in the iterator trait.

```rust
def size_hint(&self) -> (usize, Option<usize>)
Returns the bounds on the remaining length of the iterator. Specifically, size_hint() returns a tuple where the first element is the lower bound, and the second element is the upper bound. The second half of the tuple that is returned is an Option<usize>. A None here means that either there is no known upper bound, or the upper bound is larger than usize.

It is not enforced that an iterator implementation yields the declared number of elements. A buggy iterator may yield less than the lower bound or more than the upper bound of elements. size_hint() is primarily intended to be used for optimizations such as reserving space for the elements of the iterator, but must not be trusted to e.g., omit bounds checks in unsafe code. An incorrect implementation of size_hint() should not lead to memory safety violations. That said, the implementation should provide a correct estimation, because otherwise, it would be a violation of the trait’s protocol.
```

However, that flexibility is not useful in practice: The upper bound is rarely used, and I’m told that using it in reserved was tried and led to performance regression. The complex return type seems to also create some confusion for rust users.

Naming

SG-9 had no strong preferences between the naming of the size_hint function. The two options considered are size_hint (which is nice because it reflects the fact that it’s a function intended as a performance hint) and approximate_size (which is nice because it reflects the name of the approximately Sized_range concept). Both options being equally cromulent, we are happy to let LEWG pick their preference.

Wording

[Editor’s note: Add the macro __cpp_lib_ranges_zip to <version> and <ranges>]
#define __cpp_lib_ranges_zip 2026XX (**placeholder**)

⚠️ Header <ranges> synopsis

```c
#include <compare>    // see ??
#include <initializer_list> // see ??
#include <iterator>    // see ??
```
namespace std::ranges {
inline namespace unspecified {

// ?, range access
inline constexpr unspecified begin = unspecified; // freestanding
inline constexpr unspecified end = unspecified;    // freestanding
inline constexpr unspecified cbegin = unspecified; // freestanding
inline constexpr unspecified cend = unspecified;   // freestanding
inline constexpr unspecified rbegin = unspecified; // freestanding
inline constexpr unspecified rend = unspecified;   // freestanding
inline constexpr unspecified crbegin = unspecified; // freestanding
inline constexpr unspecified crend = unspecified;  // freestanding

inline constexpr unspecified size = unspecified;   // freestanding
inline constexpr unspecified ssize = unspecified;  // freestanding
inline constexpr unspecified empty = unspecified;  // freestanding
inline constexpr unspecified data = unspecified;   // freestanding
inline constexpr unspecified cdata = unspecified;  // freestanding
}

template<class>
constexpr bool disable_sized_range = false; // freestanding

template<class T>
concept approximately_sized_range = see below; // freestanding

template<class T>
concept sized_range = see below; // freestanding

template<class T>
constexpr bool enable_view = see below; // freestanding
}

[Editor's note: Insert after {[range.prim.ssize]}]

ranges::size_hint
[[range.prim.size.hint]]

The name ranges::size_hint denotes a customization point object [customization.point.object].

Given a subexpression E with type T, let t be an lvalue that denotes the reified object for E. Then:

- If ranges::size(E) is a valid expression, ranges::size_hint(E) is expression-equivalent to ranges::size(E).
- Otherwise, if auto(t.size_hint()) is a valid expression of integer-like type [iterator.concept.winc], ranges::size_hint(E) is expression-equivalent to auto(t.size_hint()).
- Otherwise, if T is a class or enumeration type and auto(size_hint(t)) is a valid expression

```cpp
8
```
of integer-like type where the meaning of size_hint is established as-if by performing argument-dependent lookup only [basic.lookup.argdep], then ranges::size_hint(E) is expression-equivalent to that expression.

- Otherwise, ranges::size_hint(E) is ill-formed.

[Note: Diagnosable ill-formed cases above result in substitution failure when ranges::size_hint(E) appears in the immediate context of a template instantiation. — end note]

[Note: Whenever ranges::size_hint(E) is a valid expression, its type is integer-like. — end note]

■**Sized ranges**  
[range.sized]

■**Approximately sized ranges**  
[range.approximately.sized]

The approximately_sized_range concept refines range with the requirement that an approximation of the number of elements in the range can be determined in amortized constant time using ranges::size_hint.

```cpp
template<class T>
concept approximately_sized_range =
    range<T> && requires(T& t) { ranges::size_hint(t); };
```

Given an lvalue t of type remove_reference_t<T>, T models approximately_sized_range only if ranges::size_hint(t) is amortized \( O(1) \), and does not modify t.

The sized_range concept refines `range` approximately_sized_range with the requirement that the number of elements in the range can be determined in amortized constant time using ranges::size.

```cpp
template<class T>
concept sized_range =
    range approximately_sized_range<T> && requires(T& t) { ranges::size(t); };
```

Given an lvalue t of type remove_reference_t<T>, T models sized_range only if

- ranges::size(t) is amortized \( O(1) \), does not modify t, and is equal to ranges::distance(ranges::begin(t), ranges::end(t)), and

- if iterator_t<T> models forward_iterator, ranges::size(t) is well-defined regardless of the evaluation of ranges::begin(t). [Note: ranges::size(t) is otherwise not required to be well-defined after evaluating ranges::begin(t). For example, it is possible for ranges::size(t) to be well-defined for a sized_range whose iterator type does not model forward_iterator only if evaluated before the first call to ranges::begin(t). — end note]

■ **ranges::to**  
[range.utility.conv.to]

```cpp
template<class C, input_range R, class... Args> requires (!view<C>)
constexpr C to(R&& r, Args&&... args);

*Mandates:* C is a cv-unqualified class type.

*Returns:* An object of type C constructed from the elements of r in the following manner:

- If C does not satisfy input_range or convertible_to<range_reference_t<R>, range_value_t<C>> is true:
  - If constructible_from<C, R, Args...> is true:
    
    
    ```cpp
    C(std::forward<R>(r), std::forward<Args>(args)...)  
    ```

  - Otherwise, if constructible_from<C, from_range_t, R, Args...> is true:
    
    ```cpp
    C(from_range, std::forward<R>(r), std::forward<Args>(args)...)  
    ```

- Otherwise, if
  - common_range<R> is true,
  - the qualified-id iterator_traits<iterator_t<R>>::iterator_category is valid and denotes a type that models derived_from<input_iterator_tag>, and
  - constructible_from<C, iterator_t<R>, sentinel_t<R>, Args...> is true:
    
    ```cpp
    C(ranges::begin(r), ranges::end(r), std::forward<Args>(args)...)  
    ```

- Otherwise, if
  - constructible_from<C, Args...> is true, and
  - container-insertable<C, range_reference_t<R>> is true:
    
    ```cpp
    C c(std::forward<Args>(args)...);
    if constexpr (approximately_sized_range<R> && reservable-container<C>)
    c.reserve(static_cast<range_size_t<C>>(ranges::size_hint(r)));
    ranges::copy(r, container inserter<range_reference_t<R>>(c));  
    ```

- Otherwise, if input_range<range_reference_t<R>> is true:
  
  ```cpp
  to<C>(r | views::transform([](auto&& elem) {
    return to<range_value_t<C>>(std::forward<decltype(elem)>(elem));
  })), std::forward<Args>(args)...);  
  ```

- Otherwise, the program is ill-formed.

---

**Class template ref_view**

`ref_view` is a view of the elements of some other range.
namespace std::ranges {
template<range R>
requires is_object_v<R>
class ref_view : public view_interface<ref_view<R>> {
    //...

cconstexpr auto size() const requires sized_range<R>
{ return ranges::size(*r_); }

cconstexpr auto size_hint() const requires approximately_sized_range<R>
{ return ranges::size_hint(*r_); }

cconstexpr auto data() const requires contiguous_range<R>
{ return ranges::data(*r_); }
};

Class template owning_view

owning_view is a move-only view of the elements of some other range.

namespace std::ranges {
template<range R>
requires movable<R> && (!is-initializer-list<R>) // see ??
class owning_view : public view_interface<owning_view<R>> {
    private:
    R r_ = R(); // exposition only

    public:
    owning_view() requires default_initializable<R> = default;
    constexpr owning_view(R&& t);
    owning_view(owning_view&&) = default;
    owning_view& operator=(owning_view&&) = default;

    //...

cconstexpr auto size() requires sized_range<R>
{ return ranges::size(r_); }
cconstexpr auto size() const requires sized_range<const R>
{ return ranges::size(r_); }

cconstexpr auto size_hint() requires approximately_sized_range<R>
{ return ranges::size_hint(r_); }

cconstexpr auto size_hint() const requires approximately_sized_range<const R>
{ return ranges::size_hint(r_); }

cconstexpr auto data() requires contiguous_range<R>
{ return ranges::data(r_); }
cconstexpr auto data() const requires contiguous_range<const R>
};
Class template as_rvalue_view

namespace std::ranges {
    template<input_range V>
    requires input_range<V>
    class as_rvalue_view : public view_interface<as_rvalue_view<V>> {
        V base_ = V();  // exposition only

        public:
            as_rvalue_view() requires default_initializable<V> = default;
            constexpr explicit as_rvalue_view(V base);

            //...
            constexpr auto size() requires sized_range<V> { return ranges::size(base_); }
            constexpr auto size() const requires sized_range<const V> { return ranges::size(base_); }
            constexpr auto size_hint() requires approximately_sized_range<V> {
                return ranges::size_hint(base_);
            }
            constexpr auto size_hint() const requires approximately_sized_range<const V> {
                return ranges::size_hint(base_);
            }
    };

    template<class R>
    as_rvalue_view(R&&) -> as_rvalue_view<views::all_t<R>>;
}

Class template transform_view

namespace std::ranges {
    template<input_range V, move_constructible F>
    requires view<V> && is_object_v<F> &&
    regular_invocable<F&, range_reference_t<V>> &&
    can_reference<invoke_result_t<F&, range_reference_t<V>>> class transform_view : public view_interface<transform_view<V, F>> {
        //...
        constexpr auto size() requires sized_range<V> { return ranges::size(base_); }
        constexpr auto size() const requires sized_range<const V> {
            return ranges::size(base_); }
        constexpr auto size_hint() requires approximately_sized_range<V> {
            return ranges::size_hint(base_);
        }
        constexpr auto size_hint() const requires approximately_sized_range<const V> {
            return ranges::size_hint(base_);
        }
    };
}
Class template take_view

namespace std::ranges {
    template<view V>
    class take_view : public view_interface<take_view<V>> {
        // ...
        constexpr auto size() requires sized_range<V> {
            auto n = ranges::size(base_);
            return ranges::min(n, static_cast<decltype(n)>(count_));
        }

        constexpr auto size() const requires sized_range<const V> {
            auto n = ranges::size(base_);
            return ranges::min(n, static_cast<decltype(n)>(count_));
        }

        constexpr auto size_hint() requires approximately_sized_range<V> {
            auto n = ranges::size_hint(base_);
            return ranges::min(n, static_cast<decltype(n)>(count_));
        }

        constexpr auto size_hint() const requires approximately_sized_range<const V> {
            auto n = ranges::size_hint(base_);
            return ranges::min(n, static_cast<decltype(n)>(count_));
        }
    };
}

Class template drop_view

namespace std::ranges {
    template<view V>
    // ...
    constexpr auto size() requires sized_range<V> {
        const auto s = ranges::size(base_);
        const auto c = static_cast<decltype(s)>(count_);
        return s < c ? 0 : s - c;
    }

    constexpr auto size() const requires sized_range<const V> {
        const auto s = ranges::size(base_);
        const auto c = static_cast<decltype(s)>(count_);
        return s < c ? 0 : s - c;
    }
}
constexpr auto size_hint() requires approximately_sized_range<V> {
    const auto s = ranges::size_hint(base_);
    const auto c = static_cast<decltype(s)>(count_);
    return s < c ? 0 : s - c;
}

constexpr auto size_hint() const requires approximately_sized_range<const V> {
    const auto s = ranges::size_hint(base_);
    const auto c = static_cast<decltype(s)>(count_);
    return s < c ? 0 : s - c;
}

private:
V base_ = V(); // exposition only
range_difference_t<V> count_ = 0; // exposition only
};

◆ Class template common_view

namespace std::ranges {
    template<view V>
    requires (!common_range<V> && copyable<iterator_t<V>>)
    class common_view : public view_interface<common_view<V>> {
    // ...
        constexpr auto size() requires sized_range<V> {
            return ranges::size(base_);
        }
        constexpr auto size() const requires sized_range<const V> {
            return ranges::size(base_);
        }
        constexpr auto size_hint() requires approximately_sized_range<V> {
            return ranges::size_hint(base_);
        }
        constexpr auto size_hint() const requires approximately_sized_range<const V> {
            return ranges::size_hint(base_);
        }
    }
}

◆ Class template reverse_view

namespace std::ranges {
    template<view V>
    requires bidirectional_range<V>
    class reverse_view : public view_interface<reverse_view<V>> {
    // ...
        constexpr auto size() requires sized_range<V> {
            return ranges::size(base_);
        }
    }
}
constexpr auto size() const requires sized_range<const V> {  
    return ranges::size(base_);  
}  

constexpr auto size_hint() requires approximately_sized_range<V>  
{ return ranges::size_hint(base_); }  

constexpr auto size_hint() const requires approximately_sized_range<const V>  
{ return ranges::size_hint(base_); }  
};

class template as_const_view

namespace std::ranges {
    template<view V>
    requires input_range<V>
    class as_const_view : public view_interface<as_const_view<V>> {  
        //...
        constexpr auto size() requires sized_range<V> { return ranges::size(base_); }  
        constexpr auto size() const requires sized_range<const V> { return ranges::size(base_); }  
        constexpr auto size_hint() requires approximately_sized_range<V>  
        { return ranges::size_hint(base_); }  
        constexpr auto size_hint() const requires approximately_sized_range<const V>  
        { return ranges::size_hint(base_); }  
    }  
}

class template elements_view

namespace std::ranges {
    template<class T, size_t N>
    concept has_tuple_element =  
        tuple-like<T> && N < tuple_size_v<T>;  

    template<class T, size_t N>
    concept returnable_element =  
        is_reference_v<T> || move_constructible<tuple_element_t<N, T>>;  

    template<input_range V, size_t N>
    requires view<V> && has_tuple_element<range_value_t<V>, N> &&  
        has_tuple_element<remove_reference_t<range_reference_t<V>>, N> &&  
        returnable_element<range_reference_t<V>, N>
    class elements_view : public view_interface<elements_view<V, N>> {  
        //...
        constexpr auto size() requires sized_range<V>
```cpp
{ return ranges::size(base_); }

constexpr auto size() const requires sized_range<const V>
{ return ranges::size(base_); }

constexpr auto size_hint() requires approximately_sized_range<V>
{ return ranges::size_hint(base_); }

constexpr auto size_hint() const requires approximately_sized_range<const V>
{ return ranges::size_hint(base_); }

private:
  // ???, class template elements_view::iterator
template<bool> class iterator; // exposition only

  // ???, class template elements_view::sentinel
template<bool> class sentinel; // exposition only

V base_ = V(); // exposition only
};

Class template enumerate_view

namespace std::ranges {
  template<view V>
  requires range-with-movable-references<V>
  class enumerate_view : public view_interface<enumerate_view<V>> {
    // ...

    constexpr auto size()
    requires sized_range<V>
    { return ranges::size(base_); }

    constexpr auto size() const
    requires sized_range<const V>
    { return ranges::size(base_); }

    constexpr auto size_hint() requires approximately_sized_range<V>
    { return ranges::size_hint(base_); }

    constexpr auto size_hint() const requires approximately_sized_range<const V>
    { return ranges::size_hint(base_); }

    constexpr V base() const & requires copy_constructible<V> { return base_; }
    constexpr V base() && { return std::move(base_); }
  };

  template<class R>
  enumerate_view(R&&) -> enumerate_view<typename views::all_t<R>>;
}
Class template `adjacent_view` [range.adjacent.view]

namespace std::ranges {
    template<forward_range V, size_t N>
    requires view<V> && (N > 0)
    class adjacent_view : public view_interface<adjacent_view<V, N>> {
        //...
        constexpr auto size() requires sized_range<V>;
        constexpr auto size() const requires sized_range<const V>;
        constexpr auto size_hint() requires approximately_sized_range<V>;
        constexpr auto size_hint() const requires approximately_sized_range<const V>;
    }
}

```cpp
constexpr auto size() requires sized_range<V>;
constexpr auto size() const requires sized_range<const V>;
Effects: Equivalent to:
using ST = decltype(ranges::size(base_));
using CT = common_type_t<ST, size_t>;
auto sz = static_cast<CT>(ranges::size(base_));
sz -= std::min<CT>(sz, N - 1);
return static_cast<ST>(sz);
```

```cpp
constexpr auto size_hint() requires approximately_sized_range<V>;
constexpr auto size_hint() const requires approximately_sized_range<const V>;
Effects: Equivalent to:
using ST = decltype(ranges::size_hint(base_));
using CT = common_type_t<ST, size_t>;
auto sz = static_cast<CT>(ranges::size_hint(base_));
sz -= std::min<CT>(sz, N - 1);
return static_cast<ST>(sz);
```

Class template `adjacent_transform_view` [range.adjacent.transform.view]

namespace std::ranges {
    template<forward_range V, move_constructible F, size_t N>
    requires view<V> && (N > 0) && is_object_v<F> &&
    regular_invocable<F&, REPEAT(range_reference_t<V>, N)...> &&
    can-reference<invoke_result_t<F&>, REPEAT(range_reference_t<V>, N)...>>
    class adjacent_transform_view : public view_interface<adjacent_transform_view<V, F, N>> {
        //...
        constexpr auto size() requires sized_range<InnerView> { 
            return inner_.size(); 
        }
    }
}
constexpr auto size() const requires sized_range<const InnerView> {
    return inner_.size();
}

constexpr auto size_hint() requires approximately_sized_range<InnerView> {
    return inner_.size_hint();
}

constexpr auto size_hint() const requires approximately_sized_range<InnerView> {
    return inner_.size_hint();
}

Class template chunk_view for input ranges

class chunk_view : public view_interface<chunk_view<V>> {
    //...
    constexpr auto size() requires sized_range<V>;
    constexpr auto size() const requires sized_range<const V>;
    constexpr auto size_hint() requires approximately_sized_range<V>;
    constexpr auto size_hint() const requires approximately_sized_range<const V>;
};

Effects: Equivalent to:

    return to_unsigned-like(div-ceil(ranges::distance(base_, n_));

constexpr auto size_hint() requires approximately_sized_range<V>;
constexpr auto size_hint() const requires approximately_sized_range<const V>;

Effects: Equivalent to:

    return to_unsigned-like(div-ceil(ranges::size_hint(base_, n_));

Class template chunk_view for forward ranges

namespace std::ranges {
    template<view V>
    requires forward_range<V>
    class chunk_view<V> : public view_interface<chunk_view<V>> {
        //...
        constexpr auto size() requires sized_range<V>;
        constexpr auto size() const requires sized_range<const V>;
        constexpr auto size_hint() requires approximately_sized_range<V>;
        constexpr auto size_hint() const requires approximately_sized_range<const V>;
    };
}
constexpr auto size() requires sized_range<V>;
constexpr auto size() const requires sized_range<const V>;

**Effects:** Equivalent to:

```cpp
return to-unsigned-like(div-ceil(ranges::distance(base_, n_));
```

constexpr auto size_hint() requires approximately_sized_range<V>;
constexpr auto size_hint() const requires approximately_sized_range<const V>;

**Effects:** Equivalent to:

```cpp
auto sz = ranges::distance(base_, base_) - n_ + 1;
if (sz < 0) sz = 0;
return to-unsigned-like(sz);
```

---

**Class template slide_view**

```cpp
namespace std::ranges {

template<forward_range V>
requires view<V>
class slide_view : public view_interface<slide_view<V>> {

//...

constexpr auto size() requires sized_range<V>;
constexpr auto size() const requires sized_range<const V>;
constexpr auto size_hint() requires approximately_sized_range<V>;
constexpr auto size_hint() const requires approximately_sized_range<const V>;

};
}
```

constexpr auto size() requires sized_range<V>;
constexpr auto size() const requires sized_range<const V>;

**Effects:** Equivalent to:

```cpp
auto sz = ranges::distance(base_, n_ + 1;
if (sz < 0) sz = 0;
return to-unsigned-like(sz);
```

constexpr auto size_hint() requires approximately_sized_range<V>;
constexpr auto size_hint() const requires approximately_sized_range<const V>;

**Effects:** Equivalent to:

```cpp
auto sz = static_cast<range_difference_t<R>>(ranges::size_hint(r)) - n_ + 1;
if (sz < 0) sz = 0;
return to-unsigned-like(sz);
```
namespace std::ranges {
    template<input_range V>
    requires view<V>
    class stride_view : public view_interface<stride_view<V>> {
        
        constexpr auto size() requires sized_range<V>;
        constexpr auto size() const requires sized_range<const V>;
        constexpr auto size_hint() requires approximately_sized_range<V>;
        constexpr auto size_hint() const requires approximately_sized_range<const V>;
    };
}

constexpr auto size() requires sized_range<V>;
constexpr auto size() const requires sized_range<const V>;

Effects: Equivalent to:

    return to_unsigned-like(div-ceil(ranges::distance(base_, stride_))); 

constexpr auto size_hint() requires approximately_sized_range<V>;
constexpr auto size_hint() const requires approximately_sized_range<const V>;

Effects: Equivalent to:

    return toUnsigned-like(div-ceil(ranges::size_hint(base_), stride_));

Acknowledgments

Thanks to Zach Laine and Barry Revzin for providing valuable feedback on an early draft of this paper.

References
